Collection EXPERT IT

Java Server Faces (JSF) avec Eclipse

Conception d'applications web exploitant des composants JSF

François-Xavier SENNESAL

eni Editions

Java Server Faces (JSF) avec Eclipse

Conception d'applications web

François-Xavier SENNESAL

Résumé

Ce livre sur JSF traite de la mise en œuvre de la technologie **Java Server Faces** avec l'environnement de développement **Eclipse**. Les aspects théoriques, **étayés par de nombreux exemples**, montrent comment l'usage de composants JSF permet de faciliter la conception et la maintenance des applications web, tout en offrant aux utilisateurs des services plus adaptés à leurs attentes.
Le livre s'adresse tout particulièrement à des lecteurs **maîtrisant le langage de programmation Java** et familiarisés par ailleurs avec le développement d'applications web basées sur la technologie JSP.
Les principaux points développés dans ce livre couvrent la **validation** et la **conversion des données**, la **gestion évènementielle**, la conception de **composants JSF personnalisés**, ainsi que **l'internationalisation** des applications.
Après un **rappel des notions essentielles** liées à la conception d'applications web en environnement J2EE, l'ouvrage présente dans le détail le **langage d'évaluation d'expressions** propre à la technologie JSF, ainsi que les **bibliothèques de composants** utilisées. Les chapitres suivants montrent comment exploiter efficacement les composants en question, tout en s'attardant sur les méthodes utilisables pour définir les règles de navigation des applications web.

L'auteur

François-Xavier Sennesal est responsable informatique au sein d'un organisme de recherche publique. Il assure par ailleurs des enseignements dans le domaine des nouvelles technologies à l'École Supérieure d'Ingénieurs Léonard de Vinci à Paris, ainsi qu'à l'université de Versailles St Quentin. À travers ce livre, le lecteur profite d'une riche expérience professionnelle axée sur le développement d'applications web avec JSF et Eclipse.

Ce livre numérique a été conçu et est diffusé dans le respect des droits d'auteur. Toutes les marques citées ont été déposées par leur éditeur respectif. La loi du 11 Mars 1957 n'autorisant aux termes des alinéas 2 et 3 de l'article 41, d'une part, que les "copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective", et, d'autre part, que les analyses et les courtes citations dans un but d'exemple et d'illustration, "toute représentation ou reproduction intégrale, ou partielle, faite sans le consentement de l'auteur ou de ses ayants droit ou ayant cause, est illicite" (alinéa 1er de l'article 40). Cette représentation ou reproduction, par quelque procédé que ce soit, constituerait donc une contrefaçon sanctionnée par les articles 425 et suivants du Code Pénal. Copyright Editions ENI

Introduction

Ce chapitre tient lieu de rappel concernant les notions essentielles mises en œuvre dans le cadre de la conception d'applications web basée sur la technologie Java. Il présente de façon sommaire les notions d'application web, de servlets et de pages JSP. Il traite également de l'installation de l'environnement de développement Eclipse pour permettre la création d'applications web J2EE.

© ENI Editions - All rigths reserved

Rappels sur la notion d’application web

1. Éléments constitutifs d’une application

Une application web est composée d’un descripteur de déploiement, et d’un ensemble de ressources. Ces ressources peuvent être de types variés : les plus caractéristiques sont les pages web (statiques ou dynamiques), les classes correspondant à des servlets, et les classes personnalisées.

Le descripteur de déploiement est un fichier conventionnellement nommé web.xml. Il regroupe diverses informations concernant l’application, comme la déclaration des servlets, la définition des paramètres initiaux ou la déclaration de listeners.

2. Vie d’une application

Le fonctionnement d’une application web basée sur la technologie J2EE requiert l’utilisation d’un moteur de servlets tenu de la prendre en charge. Le moteur de servlets le plus populaire est Apache Tomcat.

Lorsqu’une page JSP de l’application web est appelée pour la première fois par un internaute, celle-ci est dans un premier temps traduite en servlet par le moteur de servlets. Une tentative de compilation de cette servlet est ensuite assurée. Si la compilation réussit, la servlet est exécutée : cela a le plus souvent pour effet de restituer un contenu dans le navigateur de l’internaute. Si la compilation de la servlet aboutit à un échec, un message d’erreur est généré par le moteur de servlets, puis restitué à l’internaute. Ce processus de traduction/compilation est par ailleurs réalisé chaque fois qu’une modification est apportée dans le code source de la page JSP.

© ENI Editions - All rigths reserved

Les servlets

1. Présentation

Une servlet est une classe implémentant l'interface `javax.servlet.Servlet`. Elle fonctionne par l'intermédiaire du moteur de servlets, en réceptionnant les requêtes émises par les clients web, et en y répondant.

Il existe actuellement deux classes standards implémentant Servlet : il s'agit de classes abstraites nommées `GenericServlet` et `HttpServlet`. La première caractérise une servlet indépendante de tout protocole, alors que la seconde est spécifique au protocole HTTP. Enfin, l'interface `Servlet` est également implémentée en standard par une classe non abstraite nommée `FacesServlet` : celle-ci gère le cycle de vie des applications web exploitant des composants Java Server Faces.

2. Cycle de vie

Une servlet assure le traitement des requêtes émises par les clients par l'intermédiaire de deux objets, issus des classes `ServletRequest` et `ServletResponse`. Ces objets sont fournis en paramètre d'une méthode abstraite nommée `service`, qu'il convient d'implémenter pour permettre la prise en charge des requêtes clientes.

Outre cette méthode `service`, dont l'invocation est assurée à chaque fois qu'une requête cliente est émise, une servlet implémente deux méthodes entrant spécifiquement en jeu dans son cycle de vie. Il s'agit des méthodes `init` et `destroy`. La première est appelée par le moteur de servlets juste après la création de la servlet, et uniquement à ce moment. La seconde de ces deux méthodes est invoquée juste avant la destruction de la servlet, et uniquement dans ce cas.

L'exemple suivant illustre la conception d'une servlet basée sur la classe `GenericServlet`. Sa méthode `service` exploite l'objet de type `ServletResponse` pour retourner une chaîne de caractères vers le flux de sortie.

```
package perso;
import java.io.IOException;

import javax.servlet.GenericServlet;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;

public class MaGenericServlet extends GenericServlet {

     @Override
     public void service(ServletRequest arg0, ServletResponse arg1)
                throws ServletException, IOException {
          arg1.getWriter().println("Bienvenue");

     }
}
```

3. Cas particulier des servlets répondant spécifiquement au protocole HTTP

Le meilleur moyen de concevoir une servlet capable de prendre en charge les requêtes HTTP consiste à créer une classe dérivée de `HttpServlet`.

Cette classe réalise, au travers de sa méthode `init`, un traitement répartissant les requêtes compte tenu de leur nature. Ainsi, une requête GET pourra être prise en charge au travers de la méthode `doGet`, alors qu'une requête POST le sera grâce à la méthode `doPost`.

Il convient alors de surcharger l'une de ces deux méthodes, compte tenu de la nature de la requête réalisée, pour tirer convenablement profit des informations fournies par le client. Ces méthodes acceptent en argument un objet de type `HttpServletRequest` contenant toutes les caractéristiques de la requête initiale, ainsi qu'un objet de type `HttpServletResponse` utilisable pour renvoyer une réponse au client. L'objet de type `HttpServletRequest` peut en particulier être exploité pour prendre connaissance des informations passées dans la chaîne requête au moment de la soumission d'un formulaire de saisie, grâce à l'une ou l'autre des méthodes `getParameter`, `getParameterNames`, et `getParameterValues`.

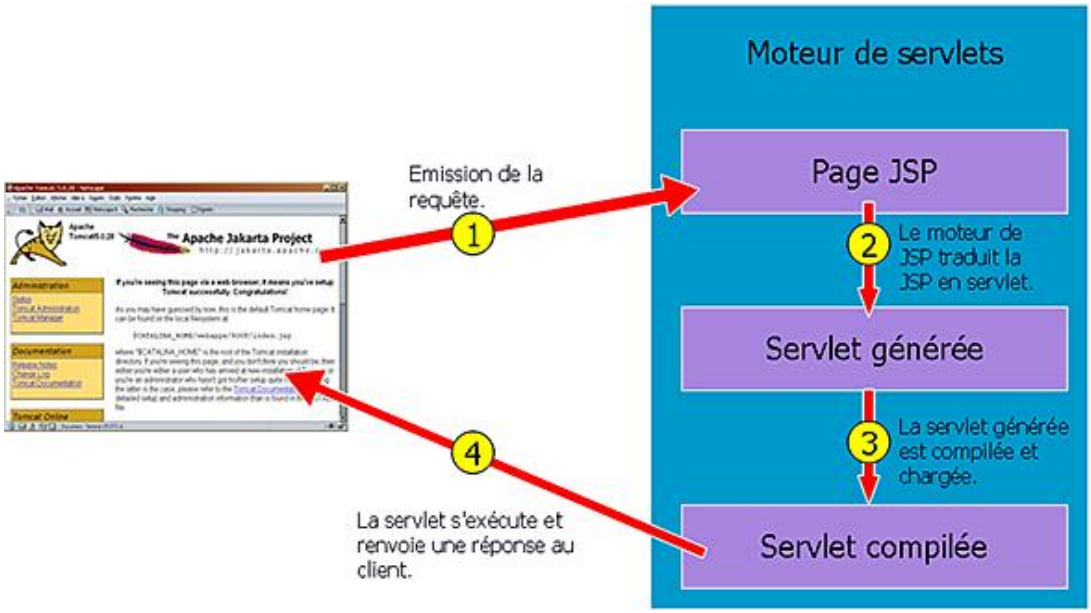
© ENI Editions - All rigths reserved

La technologie Java Server Pages (JSP)

1. Une brève description du mode de fonctionnement

La technologie JSP est conçue pour faciliter la génération de pages web dynamiques s'appuyant sur Java. Elle est directement associée à la notion de servlets, et permet d'obtenir une séparation entre la logique métier et la couche de présentation, principalement par l'usage des JavaBeans.

Lorsqu'une page JSP située dans une application web particulière est demandée par un client web, le conteneur qui la gère (moteur de servlets) capture et analyse la requête. Si la page en question n'a jamais été appelée, ou si son code source a été modifié, le conteneur la traduit sous la forme d'une servlet. Cette servlet est alors compilée puis exécutée pour fournir une réponse à la requête du client web. Par contre, si la page a déjà été utilisée et si son code source n'a pas été modifié, la servlet qui lui correspond existe déjà : il suffit alors au conteneur d'exécuter cette servlet pour répondre au client.



2. Les composants d'une page JSP

Le code source le plus simple d'une page JSP peut correspondre à celui-ci :

```
<html>
...
<body>
<%
out.println("Ceci est un cours de développement web avec les JSP");
%>
...
</body>
</html>
```

D'une manière générale, une page JSP se compose d'éléments statiques, caractérisés par des balises HTML, et d'éléments dynamiques. Ces derniers peuvent être classés en quatre catégories : les directives, les actions, les objets implicites, et les zones de scripting.

a. Les directives

Ces éléments fournissent des informations globales concernant la page JSP. La forme générale de la syntaxe utilisée pour définir une directive est la suivante :

```
<%@ nom_directive {nom_attribut1="valeur_attribut1"}
{nom_attributN="valeur_attributN"}%>
```

© ENI Editions - All rigths reserved

Une directive est donc caractérisée par un ou plusieurs attributs. L'illustration la plus couramment répandue consiste à mentionner le langage à utiliser pour la compilation de la page JSP.

```
<%@ page language="java" %>
```

Il convient de préciser qu'une page JSP particulière peut comporter plusieurs directives.

Les spécifications JSP proposent trois types de directives, nommées `page`, `include` et `taglib`. Chaque type de directive dispose d'un jeu d'attributs spécifiques, dont voici un bref récapitulatif :

Principaux attributs de la directive page

- `language` : langage utilisé pour la compilation de la JSP. Actuellement, uniquement Java.
- `import` : ensemble des paquetages Java à mettre à disposition de la page JSP.
- `session` : indique si les données de session sont disponibles pour la page (valeur par défaut : `true`).
- `error_page` : URL de la JSP prenant en charge les exceptions rencontrées.
- `extends` : classe parente dont dérivera la servlet générée.
- `isErrorPage` : indique si la page courante est une *errorPage*. Default=false.

L'instruction suivante illustre l'importation de paquetages au sein d'une page JSP :

```
<%@page import="java.sql.*, java.io.*" session="false" %>
```

Attribut de la directive include

La directive `include` ne possède qu'un unique attribut dénommé `file`. Celui-ci indique le nom d'un document HTML ou d'une page JSP à inclure automatiquement dans la page courante.

```
<%@ include file="../../piedDePage.html" %>
<%@ include file="formulaireInscription.jsp" %>
```

Principaux attributs de la directive taglib

- `uri` : identifie de manière unique le descripteur d'une bibliothèque de balises personnalisées à utiliser dans la page JSP.
- `prefix` : chaîne utilisée pour distinguer des instances de balises personnalisées.

```
<%@ taglib uri="../hello.tld" prefix="exemple" %>
<%@ taglib uri="http://www.monserveur.fr/hello.tld"
          prefix="test" %>
<exemple:balise1 name="jacques">…</exemple:balise1>
<test:balise2 />
```

La directive `taglib` permet donc d'étendre le jeu de balises JSP standard par l'emploi de bibliothèques de balises personnalisées.

b. Les actions

Les actions standards constituent un moyen technique destiné à encapsuler les tâches réalisées les plus couramment. Elles se présentent sous la forme de balises incorporées dans les pages JSP. Au cours de la compilation de la servlet, le conteneur rencontre ces balises et les remplace par le code Java correspondant.

Voici les différents types d'actions standards :

L'action <jsp:useBean>

© ENI Editions - All rigths reserved

Elle est utilisée pour instancier un JavaBean, ou en localiser une instance existante. Cette méthode est très efficace pour séparer la présentation web de la logique métier, dont le code Java est encapsulé dans un JavaBean.

Les attributs de cette action permettent de faire correspondre un nom de variable au JavaBean, et de lui affecter une durée de vie grâce à la notion de portée (page, session, application). Suivant la portée choisie, le JavaBean peut ou non être accessible à partir d'autres pages JSP.

```
<jsp:useBean id="nomBean" scope="session" class="classeBean"/>
```

L'action <jsp:setProperty>

Cette action est utilisée conjointement à `<jsp:useBean>`. Elle permet de définir les valeurs des propriétés d'un JavaBean.

La définition des propriétés d'un bean peut être assurée de plusieurs manières :

- À l'aide des paramètres de requêtes. Toutes les propriétés du bean sont spécifiées grâce aux informations fournies :

```
http://www.monserveur.fr/maPage.jsp?nom=dupond&prenom=paul
<jsp:setProperty name="nomBean" property="*" />
```

- Grâce à l'utilisation explicite d'un nom de propriété :

```
<jsp:setProperty name="nomBean" property="nom" value="dupond" />
```

L'action <jsp:getProperty>

Cette action est également utilisée conjointement à `<jsp:useBean>`. Elle permet d'accéder aux valeurs des propriétés d'un JavaBean.

```
<jsp:getProperty name="nomBean" property="nom" />
```

L'exemple suivant illustre l'utilisation conjointe des actions `<jsp:useBean>`, `<jsp:getProperty>` et `<jsp:setProperty>` dans une page JSP :

```
<jsp:useBean id="nomBean" scope="session"  class="classeBean"/>
<jsp:setProperty name="nomBean" property="nom" value="dupond" />
<html><head><title>
Exemple d'utilisation d'un JavaBean dans une JSP
</title></head><body>
La valeur du nom est :
<jsp:getProperty name="nomBean" property="nom" />
</body></html>
```

L'action <jsp:forward>

Elle permet la redirection de la requête, au moment de l'exécution, vers une ressource statique, une servlet, ou une autre JSP. Son usage interrompt l'exécution de la page courante. Cette action peut elle-même contenir des actions `<jsp:param>`.

```
<jsp:forward page="pageSuivante.jsp">
     {<jsp:param .../>}
</jsp:forward>
```

L'action <jsp:include>

Cette action fournit un mécanisme pour inclure dans la page des ressources statiques ou dynamiques externes (par exemple, autres pages JSP ou HTML). Elle peut elle-même contenir des actions `<jsp:param>`, constituant les paramètres à passer à la ressource à inclure.

```
<jsp:include page="hautPage.jsp">
     {<jsp:param ... />}
</jsp:include>
```

L'action <jsp:plugin>

© ENI Editions - All rigths reserved

Elle permet la génération de documents HTML contenant des plug-ins qui seront exécutés par exemple sous la forme d'applets Java, ou sous la forme d'un composant JavaBean. Au moment de l'exécution de la page, cette action est remplacée soit par une balise `<embed>`, soit par une balise `<object>`, suivant la nature du navigateur client. L'action `<jsp:plugin>` peut elle-même contenir des actions `<jsp:param>`.

```
<jsp:plugin type="type de plugin (ex:applet)"
         code="nom de la classe exécutée par le plugin"
         codebase="chemin d'accès à la classe" >
     {<jsp:param ... />}
</jsp:plugin>
```

L'action <jsp:param>

Cette action est utilisée uniquement conjointement à `<jsp:include>`, `<jsp:forward>` ou `<jsp:plugin>` : elle permet de créer des paires nom/valeur, qui tiennent lieu de paramètres passés aux actions dans lesquelles elles sont placées.

```
<jsp:params>
     <jsp:param name="nom" value="DUPONT" />
</jsp:params>
```

c. Les objets implicites

Les objets implicites sont des objets accessibles dans toute page JSP, sans qu'il soit nécessaire de les instancier au préalable. Les spécifications JSP précisent que tous les langages de scripts pouvant servir à créer des pages JSP doivent permettre d'utiliser ces objets.

Ces objets implicites sont bien sûr instanciés à partir de classes ou interfaces disponibles dans le JDK (*Java Development Kit*) ou dans le JSDK (*Java Servlet Development Kit*) - Package `javax.servlet`.

Voici un récapitulatif des principaux objets implicites :

application

Représente le contexte d'exécution des servlets. Il s'agit d'un objet de type `javax.servlet.ServletContext` disposant de la portée session.

out

Correspond au `JspWriter` rattaché au flux de sortie. Il dispose de la portée page.

page

Correspond à l'objet "this" représentant la page en cours. Il dispose de la portée page.

pageContext

Représente le contexte de la page en cours. Sa portée est page.

request

Représente la requête soumise au moteur de servlet par le client (navigateur). Cet objet est issu de `javax.servlet.ServletRequest`. Sa portée est `request`.

response

Représente la réponse retournée par le moteur de servlet au client. Il s'agit d'un objet implémentant l'interface `javax.servlet.http.HttpServletResponse`, dont la portée est `page`.

session

Représente la session propre au client. De type `javax.servlet.http.HttpSession`, cet objet dispose de la portée `session`.

d. Les scripting JSP

© ENI Editions - All rigths reserved

Les scripting JSP constituent un mécanisme permettant d'inclure du code Java dans des pages JSP. Ils sont composés de trois types d'éléments : les déclarations, les expressions, et les scriptlets.

Les déclarations

Les déclarations sont utilisées pour déclarer des variables et des méthodes au sein d'une page JSP. Elles sont initialisées lors de l'initialisation de la page JSP. Dès que cette opération est réalisée, elles sont disponibles pour toutes les autres déclarations et expressions, ainsi que pour les scriptlets.

```
<%! String nom=new String("DUPONT"); %>
<%! Public String getNom() { return nom;} %>
```

Les expressions

Les expressions permettent d'évaluer le résultat d'une instruction sous la forme d'une chaîne de caractères, et de l'inscrire immédiatement sur le flux de sortie. Lorsque le résultat de l'instruction ne peut pas être converti en chaîne de caractères, une exception est levée.

```
<HTML><BODY>
...
Bonjour Monsieur <%= getNom()%>
</BODY></HTML>
```

Les scriptlets

Les scriptlets servent à regrouper tous les éléments de scripts entre les balises `<%` et `%>`. Ils peuvent contenir toute instruction de code compatible avec le langage mentionné dans l'attribut `language` de la directive `page`.

© ENI Editions - All rigths reserved

Installation de l'environnement Eclipse et configuration d'une application web

Cette section présente les étapes nécessaires à l'installation de l'environnement de développement Eclipse en vue de concevoir des applications web basées sur J2EE et exploitant des composants JSF. Les informations suivantes supposent qu'un moteur de servlets est déjà installé et correctement configuré.

L'installation de l'environnement se fait en deux étapes :

1. Choix d'une implémentation JSF

Il existe différentes implémentations permettant d'exploiter la technologie Java Server Faces, notamment MyFaces et Sun JSF 1.2 RI. Les exemples et illustrations présentés dans cet ouvrage ont été réalisés à partir de cette dernière implémentation, à laquelle ont été joints quelques autres composants destinés à faciliter la conception des applications web. Au final, les fichiers suivants ont été utilisés pour implémenter JSF dans le cadre des exemples présentés dans cet ouvrage :

```
common-annotations.jar
commons-beanutils.jar
commons-collections.jar
commons-digester.jar
commons-logging.jar
jsf-api.jar
jsf-impl.jar
jstl.jar
standard.jar
```

Ces fichiers peuvent être obtenus sur le site des Éditions ENI : www.editions-eni.fr

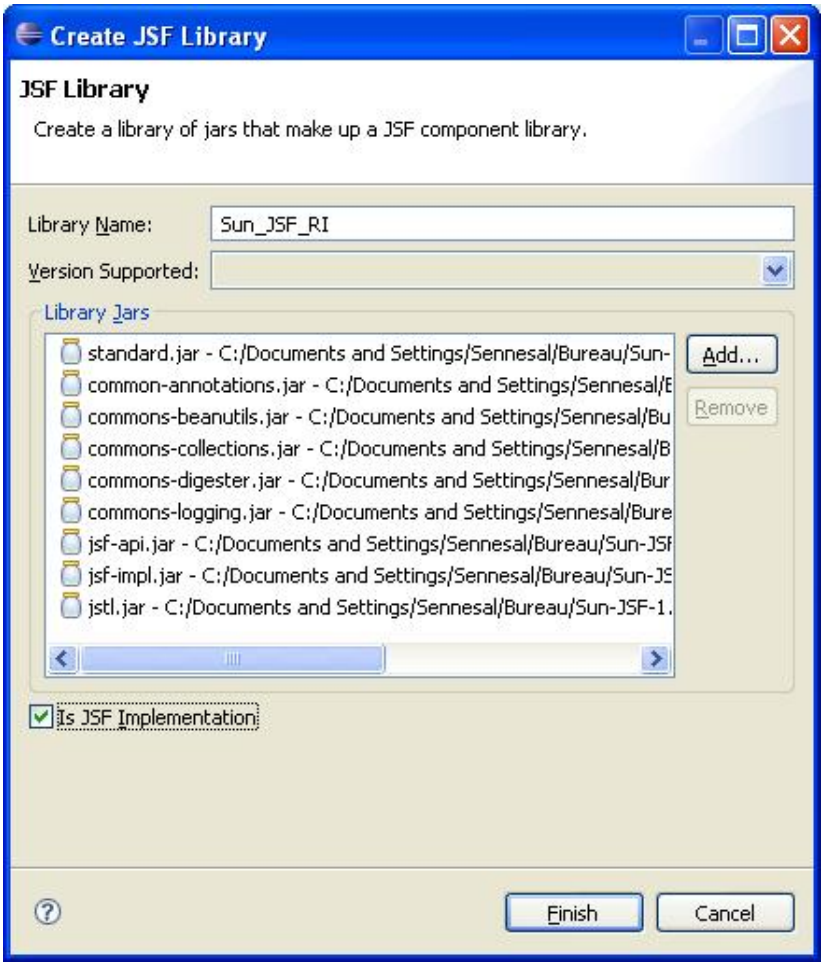
2. Installation et configuration de Eclipse et de la Web Tool Platform

Pour installer l'IDE Eclipse, il suffit de télécharger l'une des versions de ce produit prenant en charge J2EE. La version utilisée dans le cadre de cet ouvrage est Eclipse Europa (Eclipse v3.3.2) disponible à l'adresse http://www.eclipse.org/downloads/packages/release/europa/winter

Une fois le produit installé et activé, il est possible de créer un premier projet exploitant la technologie JSF en suivant les étapes suivantes :

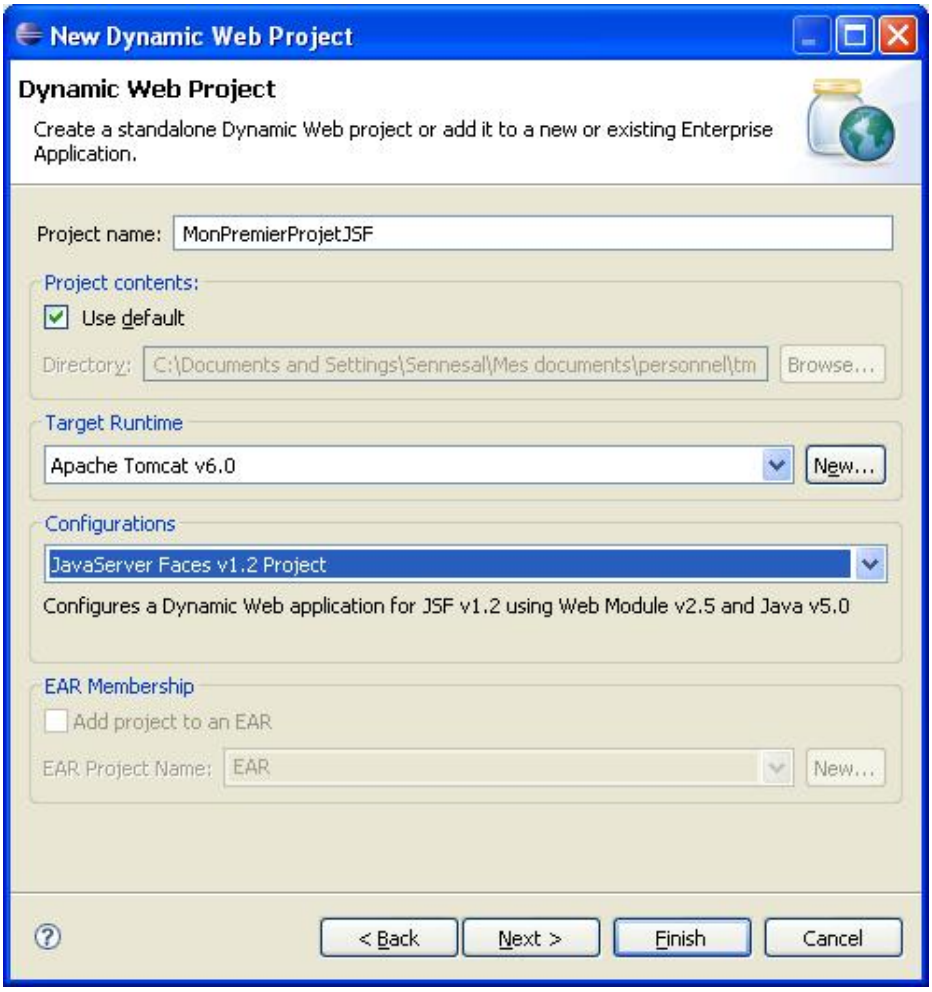
- Déclarer l'implémentation JSF, évoquée précédemment, à partir du menu **Window - Preferences**. En déployant le nœud **Web and Xml** de l'arborescence, on accède successivement aux nœuds **Java Server Faces Tools**, puis **Libraries**. Il est alors possible de définir la bibliothèque d'archives Jar mettant en œuvre les composants JSF. Le nom de cette bibliothèque peut être choisi librement. Par contre, il est indispensable de cocher la case intitulée **Is JSF Implementation**.

© ENI Editions - All rigths reserved



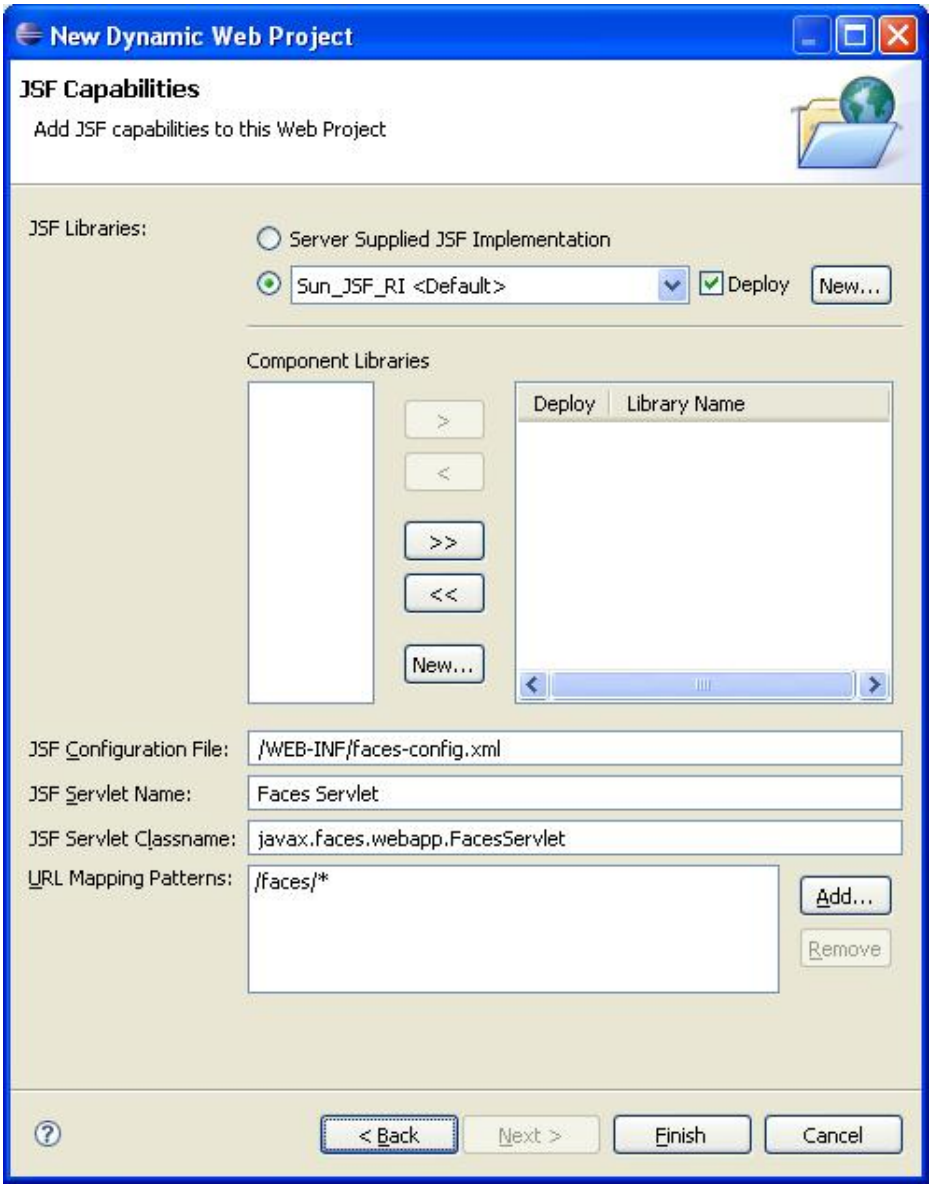
- La création d'un projet web dynamique se fait ensuite par utilisation du menu **File - New Project**. La fenêtre qui apparaît permet de sélectionner le choix **Dynamic Web Project** proposé dans l'arborescence du nœud **Web**. Il faut alors mentionner un nom de projet, préciser le nom et la localisation du moteur de servlets dans la zone **Target Runtime**, et sélectionner la configuration **JavaServer Faces v1.2 Project** depuis la zone **Configurations**.

© ENI Editions - All rigths reserved



Il est ensuite possible d'accepter toutes les options proposées dans les deux écrans suivants (visibles en cliquant successivement deux fois sur le bouton **Next**). Lorsque la rubrique d'options dénommée **JSF Capabilities** est affichée, il est indispensable de préciser l'implémentation JSF à utiliser dans le cadre de ce nouveau projet : cela se fait grâce aux boutons radio **JSF Libraries**. Attention, il est impératif de cocher la case **Deploy** avant de cliquer sur le bouton **Finish**.

© ENI Editions - All rigths reserved



- Une fois ces opérations réalisées, le nouveau projet web dynamique est convenablement configuré. Il est donc possible de lui adjoindre différentes ressources, telles que des classes, des pages JSP ou HTML.

© ENI Editions - All rigths reserved

La technologie Java Server Faces

Avec l'émergence du web 2.0 et l'accroissement de la puissance des ordinateurs, de nouveaux besoins sont apparus vis-à-vis des applications web : les utilisateurs souhaitent désormais une interaction plus forte entre le client qu'ils exploitent (navigateur, le plus souvent) et le serveur web. À l'image de la réactivité dont ils bénéficient lors de l'utilisation de logiciels non orientés web, les internautes attendent maintenant d'un site web qu'il réagisse quasiment instantanément aux actions réalisées dans l'interface. Quant aux développeurs d'applications web, ils veulent aujourd'hui pouvoir concevoir rapidement de nouveaux outils, en exploitant le concept de composants.

De telles exigences ne peuvent pas trouver de réponse adaptée au travers de la seule technologie Java Server Pages, ni même de l'usage des servlets : sans connaissances techniques particulièrement poussées, ces deux moyens ne permettent pas un développement rapide d'applications web riches en fonctionnalités. Ils sont également dépourvus de dispositifs simples à utiliser en vue de la création de composants graphiques élaborés, éventuellement personnalisables. La création de tels composants requiert, là encore, beaucoup de temps et d'expertise.

Conscient de ces manques, Sun Microsystems propose la technologie JSF (*Java Server Faces*) dont le but principal est de fournir aux concepteurs d'applications web Java un framework de composants spécifiques. JSF permet la création d'applications s'exécutant côté serveur. Les interfaces graphiques restituées aux clients s'appuient sur un modèle de composants particulièrement élaboré, permettant la gestion des événements et une restitution variable de l'apparence des composants graphiques suivant le contexte.

Le cœur de la technologie JSF repose principalement sur les éléments suivants :

- Un langage d'expressions unifié facilitant le travail du designer web. Celui-ci permet d'accéder rapidement en lecture comme en écriture aux propriétés des composants. Les méthodes de ces composants peuvent également être invoquées à l'aide du langage d'expressions.

- Deux bibliothèques de balises spécifiques, étendant le jeu de balises standards JSP. Celles-ci sont conçues pour faciliter l'intégration des composants graphiques dans les pages JSP, ainsi que pour permettre leur liaison avec les composants côté serveur.

- Une API spécifique permettant de disposer d'une représentation des composants graphiques côté serveur, d'assurer la gestion des événements et la validation des données saisies. Cette API offre également un support pour l'internationalisation des applications web.

© ENI Editions - All rigths reserved

Principe du langage d’évaluation d’expressions

Les spécifications JSP2.0 de la technologie Java Server Pages proposent un langage d’évaluation d’expressions. Son principal intérêt est de faciliter le travail des designers web en leur permettant d’accéder simplement et rapidement, mais en lecture seulement, aux propriétés de composants JavaBeans depuis une page JSP. Cet artifice technologique a longtemps suffi pour répondre au fonctionnement des applications web, dans la mesure où celles-ci ne faisaient intervenir que des échanges simples entre client et serveur.

Grâce à une évolution des spécifications JSP et l’introduction d’un langage d’expressions spécifique à JSF, il est aujourd’hui possible de développer une application web Java dont l’interface prend instantanément en charge les événements générés par les actions de l’internaute. Alors qu’une application JSP standard ne supporte qu’un unique cycle de vie de type requête/réponse, une application JSF s’appuie sur un cycle de vie à phases multiples permettant la gestion événementielle, la liaison de composants visuels avec des JavaBeans, ainsi que la mise en place de processus automatiques de conversion et de validation des données saisies.

L’exploitation de ces fonctionnalités est particulièrement facilitée par l’usage du langage d’expressions propre à JSF, unifié au langage d’expressions initialement conçu pour JSP. Là où le langage ne permettait auparavant que l’accès en lecture seule des propriétés de JavaBeans, il est maintenant possible d’accéder en lecture comme en écriture à ces propriétés. En outre, ce langage unifié permet d’invoquer les méthodes de JavaBeans et d’évaluer les expressions en instantané comme en différé.

© ENI Editions - All rigths reserved

Le cycle de vie à phases multiples d'une page JSF

Après une présentation rapide de la notion de cycle de vie à phases multiples d'une page JSF, cette partie décrit chacune des phases en question. Elle apporte des informations purement théoriques, qui intéresseront naturellement le lecteur souhaitant comprendre avec précision comment une application JSF assure la prise en charge de la gestion des événements ainsi que la conversion et la validation des données. Mais l'acquisition de ces apports théoriques n'est cependant pas indispensable à la réalisation de projets basés sur JSF.

1. Principe

Globalement, il est possible de considérer le cycle de vie d'une page JSF comme semblable à celui d'une page JSP : un client web émet une requête HTTP vers la page en question, puis le serveur qui l'héberge répond en renvoyant une page traduite en HTML. Mais en réalité, le cycle de vie d'une page JSF est partitionné en plusieurs phases, dans le but de pouvoir répondre efficacement aux attentes liées à cette technologie : capture d'événements au niveau des composants graphiques, validation ou conversion de composants, liaison entre composants d'interface et objets s'exécutant côté serveur.

Deux types de requêtes doivent être distingués :

- la requête initiale au cours de laquelle l'internaute accède pour la première fois à la page JSF,
- la requête dite postback correspondant le plus souvent à la validation d'un formulaire précédemment chargé dans le navigateur du client.

Lorsque l'application web traite une requête initiale, seules les phases de restitution de la vue et de traduction de la réponse sont exécutées. Le traitement d'une requête postback requiert quant à lui le passage par chacune des phases du cycle de vie de la page JSF, sauf en cas d'erreur, dans le but de réaliser toutes les conversions et validations nécessaires. La page est ensuite restituée au client.

2. Présentation des différentes phases

a. Phase Restitution de la vue (Restore view)

Dès qu'un internaute tente d'atteindre une page JSF, la phase de restitution de vue est exécutée. Son but est d'associer une vue à la page visitée. L'application web crée une vue vierge dans le cas d'une requête initiale ; elle restitue la vue précédemment associée à la page dans le cas d'une requête postback. Tous les gestionnaires d'événements ainsi que les validateurs requis par les composants graphiques de la page sont liés à la vue. Un arbre de composants est constitué. La vue est ensuite sauvegardée dans l'objet prédéfini `FacesContext`.

Dans le cas d'une requête initiale, le déroulement du cycle de vie se poursuit par la phase de traduction de la réponse : un appel de la méthode `renderResponse` sur l'objet `FacesContext` provoque la traduction de la vue, puis sa restitution au client.

Si l'accès à la page JSF se fait par l'intermédiaire d'une requête postback, la restitution de la vue se fait compte tenu des informations fournies dans la requête. Celle-ci prend également en charge les données éventuellement stockées sur le serveur à l'occasion d'une précédente consultation de la page. JSF assure en effet la conservation des informations de requêtes sur le serveur : ce mécanisme permet, par exemple, de restituer facilement à un internaute les données que celui-ci a déjà saisies dans un formulaire.

b. Phase Application des paramètres de requête (Apply request values)

À l'issue de la phase précédente, une vue est associée à la page JSF visitée. Cette vue dispose obligatoirement d'un arbre de composants. Ces composants sont alors parcourus successivement pour invoquer leur méthode `decode()`. Celle-ci est chargée d'analyser la requête pour attribuer au composant la valeur qui doit lui être affectée.

Il est courant que cette affectation de valeur nécessite une conversion préalable : c'est le cas notamment lorsqu'une zone de texte de formulaire est associée à une propriété de composant dont le type est numérique. Si cette conversion échoue, un message d'erreur est automatiquement attribué au composant en question, puis placé en attente de traitement dans l'objet prédéfini `FacesContext`. Ce message est ensuite pris en charge durant la phase de traduction de la réponse pour être présenté à l'internaute, en même temps que les éventuels messages de validation produits lors de la phase *Processus de validation* (voir ci-dessous). Un comportement similaire se produit si l'affectation d'une valeur au composant est soumise à une validation préalable, par exemple pour s'assurer qu'une valeur numérique saisie dans une zone de texte est bien comprise entre deux valeurs V1 et V2.

© ENI Editions - All rigths reserved

Si certains composants disposent d'un attribut `immediate` positionné sur la valeur `true`, la validation et la conversion éventuellement requises sont directement prises en charge dans cette phase du cycle de vie.

En plus de permettre l'analyse de la requête et l'attribution de valeurs aux composants, la phase *Application des paramètres de requête* transmet les événements générés au niveau des composants graphiques présentés dans la page JSF au contexte JSF.

À la fin de cette phase, tous les composants associés à la vue disposent d'une nouvelle valeur. Les messages et les événements éventuels sont en attente de traitement par l'objet `FacesContext`.

c. Phase Processus de validation (Process Validations)

Dans cette étape, l'application JSF effectue toutes les validations attendues pour chacun des composants de l'arbre associé à la vue.

Pour cela, les attributs de composant définissant les règles de validation sont examinés, puis comparés à la valeur du composant. Lorsque cette valeur est effectivement incorrecte, un message d'erreur est ajouté à l'objet `FacesContext` et le cycle de vie se poursuit directement par la phase de traduction de la réponse : la page web est alors restituée à l'internaute, accompagnée du message d'erreur issu du processus de validation.

d. Phase Mise à jour du modèle (Update Model Values)

Chaque composant d'interface JSF peut être associé à un JavaBean côté serveur. Après s'être assurée de la validité des données saisies lors de la phase *Processus de validation*, l'application JSF parcourt à nouveau l'arbre de composants pour affecter la valeur de chaque composant graphique au JavaBean qui lui est associé. Là encore, si un problème de conversion survient au moment de la mise à jour de la propriété du JavaBean, le cycle de vie de la page se poursuit directement par la phase de traduction de la réponse et un message d'erreur adapté est présenté à l'internaute.

e. Phase Appel de l'application (Invoke application)

Lorsque l'internaute valide un formulaire, ou clique sur un lien hypertexte, l'application JSF génère respectivement un objet de type "événement de formulaire" ou un objet de type "événement de commande". Ces objets sont qualifiés d'événements de niveau application : ils sont pris en charge par des gestionnaires spécifiques au cours de cette phase dont le rôle est de mentionner une URL vers laquelle la navigation est dirigée.

f. Phase Restitution de la réponse (Render response)

Cette phase correspond au moment où l'implémentation Java Server Faces rend la main au conteneur en charge des pages JSP de l'application web. Tous les composants sont alors présentés dans l'interface utilisateur, dans l'état qui est le leur au moment où survient la phase *Render response*.

© ENI Editions - All rigths reserved

Deux types d'évaluation d'expressions

Le langage d'expressions unifié est conçu pour permettre l'évaluation des expressions de deux manières distinctes.

Cette évaluation peut se produire de manière immédiate, instantanée : dans ce cas, son résultat est renvoyé à l'internaute dès la phase de restitution de la page JSP.

L'évaluation peut également avoir lieu de manière différée, c'est-à-dire qu'elle peut survenir à n'importe quel moment du cycle de vie de la page. Ce mode d'évaluation s'avère indispensable dans le cadre de la mise en œuvre de la technologie JSF, dans la mesure où le cycle de vie à phases multiples des pages web doit rendre possible l'évaluation des expressions au moment le plus opportun (notamment lors de la capture d'un événement, d'une demande de validation ou de conversion de la valeur d'un composant). La détermination du meilleur moment pour évaluer une expression reste à la charge de la technologie implémentant le langage d'expressions unifié.

1. Évaluation instantanée d'une expression

Une demande d'évaluation instantanée d'expression au sein d'une page JSP se fait à l'aide de la syntaxe `${}`. Elle peut être utilisée directement dans le texte brut de la page, ou être représentée en tant valeur de l'attribut `value` d'une balise. Il faut alors, dans ce deuxième cas, que la TLD (*Tag Library Description*) de la balise en question autorise effectivement l'utilisation d'une expression en tant que valeur de l'attribut `value`.

Le code source ci-dessous est un exemple dans lequel sont présentés ces deux modes de prise en charge de l'évaluation instantanée :

```
<%@taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<%@taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
<%@taglib uri="http://java.sun.com/jstl/core" prefix="c"%>
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c1"%>
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Test de l'évaluation instantanée</title>
</head>
<jsp:useBean id="laPersonne" class="premierProjetJSF.Personne"
scope="session">
</jsp:useBean>
<body>
<c:set var="leNom" value="SENNESAL" scope="session"></c:set>
Le nom déclaré dans la session est ${sessionScope.leNom}.
<br>

Contenu de la propriété "nom" du JavaBean "laPersonne":
<c1:out value="${laPersonne.nom}"></c1:out>
<br>
Le même résultat s'obtient en écrivant simplement ${laPersonne.nom}.
</body>
</html>
```

Dans cet exemple, une première utilisation de l'évaluation instantanée dans un texte brut est effectuée par l'instruction `${sessionScope.leNom}` : l'expression à évaluer concerne la variable de session `leNom`, instanciée par le biais de une balise `<c:set>`, dont l'accès est rendu possible par l'exploitation de l'objet implicite `sessionScope`. Une seconde utilisation de l'évaluation instantanée dans un texte brut est présentée à la fin du code source au travers de l'instruction `${laPersonne.nom}` : cette fois, l'expression à évaluer concerne la propriété `nom` d'un JavaBean instanciée dans la page par l'intermédiaire d'une balise `<jsp:useBean>`.

Cette expression est par ailleurs également utilisée en tant que valeur de l'attribut `value` d'une balise `<c1.out>`.

Qu'une demande d'évaluation instantanée d'expression soit directement faite dans un texte brut, ou qu'elle le soit au travers de l'attribut `value` d'une balise particulière, elle ne peut concerner que des tentatives d'accès en lecture uniquement. Il n'est notamment pas possible d'utiliser une évaluation instantanée d'expression pour mettre à jour une propriété spécifique d'un JavaBean.

© ENI Editions - All rigths reserved

2. Évaluation différée d'une expression

Ce mode d'évaluation, introduit avec la technologie Java Server Faces, peut être exploité à n'importe quel moment du cycle de vie d'une page web. Une expression à évaluation différée est symbolisée dans le code source d'une page par la syntaxe `#{}`. Elle ne peut pas être utilisée directement dans le texte brut d'une page web, contrairement à l'expression à évaluation immédiate. Dans l'exemple ci-après, l'attribut `value` d'une balise `<h:inputText>` représentant un champ de saisie de type `HtmlInputText` possède la valeur `#{laPersonne.nom}` : il s'agit d'une expression à évaluation différée permettant d'accéder, en lecture comme en écriture, à la propriété `nom` d'un JavaBean nommé `laPersonne`.

L'utilisation d'une expression à évaluation différée permet toujours d'accéder en lecture et en écriture à une donnée particulière.

```
<html><head><title>Test de<html>
<head>
<title>Test de l'évaluation différée</title>
</head>
<jsp:useBean id="laPersonne" class="premierProjetJSF.Personne"
scope="session"></jsp:useBean>
<body>
<f:view>
     <h:form>
          <h:panelGrid border="0" columns="1">
               <h:panelGroup>
                    <h:panelGrid border="0" columns="2">
                    <h:outputText value="Votre nom:" id="idotxt1">
                    </h:outputText>
                    <h:inputText id="itxtNom"
                                   value="#{laPersonne.nom}">
                    </h:inputText>
                    </h:panelGrid>
               </h:panelGroup>
               <h:commandButton id="btnValider" value="Valider">
               </h:commandButton>
               </h:panelGrid>
          </h:form>
</f:view></body>
</html>"
```

Lorsque la page correspondant au code source précédent est invoquée pour la première fois, l'évaluation de l'expression `#{laPersonne.nom}` se fait au moment la phase de restitution de la réponse. Un accès en mode lecture à la propriété `nom` du JavaBean est alors assuré : si cette propriété possède effectivement une valeur, celle-ci est présentée à l'internaute dans le champ de saisie.

En dehors de l'accès initial à la page web, en particulier lors de la soumission du formulaire, l'expression est évaluée à différents stades du cycle de vie, afin de permettre la mise à jour de la propriété `nom` du JavaBean en fonction de la saisie réalisée par l'internaute. Cette opération est en particulier assurée lors des processus de validation et de conversion de l'information.

Dans l'exemple proposé, l'évaluation différée d'expression est utilisée pour accéder aux propriétés d'un objet particulier. Mais l'évaluation différée peut également s'appliquer à une expression permettant d'invoquer une méthode particulière d'un objet. Ces deux notions sont abordées plus précisément dans les paragraphes qui suivent.

© ENI Editions - All rigths reserved

Utilisation d'expressions pour accéder aux propriétés d'un objet

Avant de voir comment accéder aux propriétés d'un objet, il peut être intéressant de s'attarder sur les différents types d'objets exploitables et de comprendre le mécanisme permettant à une application web de trouver ces objets.

Les objets utilisables dans les expressions (que leurs évaluations soient immédiates ou différées) sont de quatre types : ils peuvent être des JavaBeans, des collections, des énumérations, ou l'un des objets implicites définis par la spécification. Définir une expression faisant appel à un objet particulier revient à mentionner l'identifiant de l'objet en question, tel qu'il a été déclaré d'une manière ou d'une autre auprès de l'application web. Ainsi, les expressions `${laPersonne}` et `#{laPersonne}` sont deux expressions valables pour accéder à un objet dont l'identifiant est `laPersonne`. Cet objet a par exemple pu être préalablement instancié en tant que JavaBean grâce à une balise `<jsp:useBean>` située dans une page JSP, ou déclaré en tant que bean managé de l'application web, par l'intermédiaire des lignes suivantes définies dans le fichier de configuration de l'application (en général nommé **faces-config.xml**).

```
<managed-bean>
          <managed-bean-name>
          laPersonne</managed-bean-name>
          <managed-bean-class>
          premierProjetJSF.Personne
          </managed-bean-class>
          <managed-bean-scope>
          Session
          </managed-bean-scope>
     </managed-bean>
```

Concrètement, l'évaluation d'une expression comportant uniquement l'identifiant d'un objet est assurée par l'application web au travers de l'invocation de la méthode `findAttribute` sur l'objet représentant le contexte d'application lui-même. L'identifiant de l'objet est passé en paramètre de cette méthode. Celle-ci se charge alors de rechercher successivement l'objet en question dans les différentes portées : page, request, session et enfin application. Si l'objet est trouvé, sa valeur est convertie en chaîne de caractères, puis renvoyée à l'internaute. Dans le cas où l'objet n'existe pas, la valeur `null` est renvoyée.

Comme cela a déjà été évoqué dans les exemples proposés précédemment, l'accès aux propriétés d'un JavaBean se fait par l'intermédiaire de l'opérateur *point* ("."). Il est également possible de faire appel à la notation par crochets ("[…]") pour obtenir le même résultat. Bien sûr, pour faire écho à la remarque faite plus haut à propos des types d'objets exploitables, ces notations peuvent également servir à obtenir une instance particulière d'une énumération, ou un élément spécifique d'une collection. Ainsi, les deux expressions qui suivent sont équivalentes : elles permettent toutes deux d'atteindre la propriété `nom` d'un JavaBean dont l'identifiant est `laPersonne` :

```
${laPersonne.nom}
${laPersonne["nom"]}
```

Lorsqu'une propriété particulière correspond elle-même à une instance de classe, il est également possible d'atteindre les propriétés de cette instance. L'exemple suivant illustre deux moyens d'afficher le nom de la ville du lieu de résidence d'une personne : l'objet `laPersonne` dispose d'une propriété `adresse`, issue d'une classe personnalisée possédant une propriété `ville`.

```
${laPersonne.adresse["ville"]}
${laPersonne.adresse.ville}
```

Dans le cas de l'utilisation d'expressions en tant que valeur de l'attribut `value` d'un composant JSF, il est possible de combiner de multiples expressions, en les associant éventuellement avec du texte brut, comme le montre l'exemple suivant :

```
<h:outputText
           value="#{laPersonne.nom} (#{laPersonne.age} ans)"
           id="idotxt2">
</h:outputText>
```

© ENI Editions - All rigths reserved

Utilisation d'expressions pour accéder aux méthodes d'un objet

L'invocation de méthodes grâce au langage d'expressions est rendu possible avec la technologie Java Server Faces. Ces expressions doivent être utilisées qu'en tant que valeurs d'attributs de balises représentant des composants JSF spécifiques. L'exécution des méthodes concernées engendre un traitement particulier agissant sur le composant lui-même : il s'agit généralement d'assurer la gestion événementielle (sélection d'un élément dans une zone de liste déroulante, clic sur un bouton, etc.) ou d'engager un processus de validation de l'information saisie dans le composant.

```
<f:view>
     <h:form>
          <h:inputText
                    id="txtCommentaire"
                    validator="#{beanAttributValidator.checkName}"
                    label="Commentaire"
                    required="true"
                    requiredMessage="Saisie obligatoire!">
          </h:inputText>
          <h:commandButton
                    id="btnValidation"
                    value="Valider"
                    actionListener="#{beanAction.traitementClic}">
          </h:commandButton>
     </h:form>
</f:view>
```

Le code source ci-dessus montre qu'une balise `<h:inputText>`, représentant un champ de saisie de type zone de texte, possède un attribut `validator`. La valeur de celui-ci correspond à une expression de méthode, dont l'interprétation provoque l'appel de la méthode `checkName` d'un bean managé nommé `beanAttributValidator`. Cette méthode peut par exemple être chargée de vérifier que l'information saisie dans la zone de texte contient au moins une occurrence de la lettre F.

De même, le code source présente une balise `<h:commandButton>`. Celle-ci est restituée graphiquement sous la forme d'un bouton de soumission de formulaire. Elle dispose d'un attribut `actionListener`, dont la valeur correspond à une expression de méthode. La méthode en question se nomme `traitementClic` et est définie dans une classe dont le bean managé `beanAction` est une instance. Le rôle de la méthode est d'assurer la gestion des clics réalisés par l'utilisateur sur le bouton de soumission.

L'évaluation des expressions de méthodes indiquées a lieu de manière différée : cela doit systématiquement être le cas, dans la mesure où les traitements demandés peuvent être assurés à différentes étapes du cycle de vie de la page JSP. Par exemple, la méthode mentionnée dans l'attribut `validator` de la balise `<h:inputText>` est appelée durant la phase Process Validation, alors que la méthode `traitementClic` est invoquée pendant la phase Invoke Application.

Dans l'exemple proposé, les beans managés évoqués peuvent être d'un type quelconque. Par contre, les méthodes mentionnées dans les expressions doivent se conformer aux contraintes indiquées dans la TLD (*Tag Library Description*). En l'occurrence, la signature de la méthode utilisée comme valeur de l'attribut `actionListener` d'une balise `<h:commandButton>` doit obligatoirement posséder un argument de type `ActionEvents`. De même, une expression de méthode utilisée comme valeur de l'attribut `validator` d'une balise `<h:inputText>` doit impérativement faire référence à une méthode possédant trois arguments : le premier de type `FacesContext`, le second de type `UIComponent` et le troisième de type `Object`.

D'une manière générale, la signature d'une méthode utilisée dans une expression de méthode dépend donc de l'attribut de balise dans lequel elle est placée. L'ensemble des contraintes à respecter peut être retrouvé sur le web, dans les pages de documentation concernant les TLD (par exemple, à l'adresse http://java.sun.com/javaee/javaserverfaces/1.2/docs/tlddocs/index.html).

© ENI Editions - All rigths reserved

Expressions avec opérateurs

Les expressions dont l'évaluation est demandée au sein des pages JSP d'une application web peuvent faire usage de différents opérateurs, afin de faciliter l'obtention de résultats adaptés aux attentes des internautes. Parmi les types d'opérateurs utilisables, on retrouve les habituels opérateurs arithmétiques, logiques et relationnels. Ces opérateurs peuvent être indifféremment utilisés dans des expressions faisant appel à des constantes, des valeurs de propriétés d'objets, ou des résultats issus de l'exécution de méthodes de classes. La liste suivante représente les opérateurs les plus couramment utilisés, répartis par catégorie :

Opérateurs arithmétiques : `+`, `-`, `*`, `/`, `%`, `mod` et `div`.

Opérateurs logiques : `and`, `&&`, `or`, `||`, `not`, `!`

Opérateurs relationnels : `==`, `eq`, `!=`, `ne`, `<`, `lt`, `>`, `gt`, `<=`, `ge`, `>=`, `le`.

Il existe en outre un opérateur particulier noté `empty`, permettant de savoir si la valeur d'une propriété ou le résultat d'une méthode est `null`.

Voici un exemple de page JSP, illustrant l'exploitation de ces différents opérateurs dans des expressions :

```
<html><head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-
8859-1">
<title>Test des opérateurs dans les expressions</title>
</head>
<jsp:useBean id="laPersonne1" class="premierProjetJSF.Personne"
scope="session"></jsp:useBean>
<jsp:useBean id="laPersonne2" class="premierProjetJSF.Personne"
scope="session"></jsp:useBean>

<body>
La personne n°1 est-elle un client de la boutique ayant commandé
dans le mois en cours?
${laPersonne1.clientBoutique and laPersonne1.achatMoisCourant}
<br>
La personne n°2 a-t-elle le même âge que la personne n°1?
${laPersonne1.age == laPersonne2.age}
<br>
La personne n°2 habite-t-elle dans la même ville que la personne
n°1?
${laPersonne1.adresse.ville ne laPersonne2.adresse.ville}
<br>
La personne n°2 dispose-t-elle d'une adresse email ?
${!empty laPersonne2.email}
</body>
</html>
```

© ENI Editions - All rigths reserved

Principaux éléments de la bibliothèque HTML

Après une présentation des principaux attributs communs à l'ensemble des balises de la bibliothèque HTML, ce paragraphe passe en revue les principales balises représentant les composants graphiques Java Server Faces, et mentionne le nom et le rôle de leurs attributs spécifiques les plus importants.

1. Principaux attributs communs aux balises de la bibliothèque HTML

Attribut binding

Sert à indiquer une expression permettant de lier le composant représenté par la balise avec une propriété de JavaBean.

Attribut id

Permet d'indiquer l'identifiant associé au composant représenté par la balise.

2. Présentation des balises et de leurs principaux attributs spécifiques

a. Balise column

Cette balise restitue une colonne particulière d'un composant de type `UIData`. Ses principaux attributs sont :

footerClass

Contient une liste de classes de styles appliquées au pied de colonne. Les noms de classes de styles sont séparés les uns des autres par le caractère espace.

headerClass

Contient une liste de classes de styles appliquées à l'en-tête de colonne. Les noms de classes de styles sont séparés les uns des autres par le caractère espace.

b. Balise commandButton

Cette balise sert à représenter un bouton de soumission de formulaire ou un bouton de réinitialisation des champs d'un formulaire. Voici ses principaux attributs spécifiques :

action

Permet de définir le cas de navigation utilisé lorsque l'internaute clique sur le composant. S'il est nécessaire d'obtenir une génération dynamique du cas de navigation, la valeur de l'attribut `action` doit référencer une méthode capable de renvoyer un objet : l'invocation de la méthode `toString()` de cet objet doit alors correspondre à l'un des cas de navigation prévus dans l'application web.

actionListener

Permet de mentionner une méthode d'expression identifiant un gestionnaire d'événements capable de prendre en charge le clic sur le composant. Ce gestionnaire d'événements est obligatoirement une méthode de type `void` acceptant en argument un objet de type `ActionEvent`.

value

Sert à indiquer le texte affiché sur le composant. La valeur utilisée pour cet attribut peut correspondre à une expression pointant sur une propriété de bean managé.

disabled

La valeur de cet attribut est un booléen indiquant si le composant peut, ou non, être utilisé par l'internaute. Dans le cas où le composant n'est pas utilisable, il est graphiquement représenté par un bouton grisé.

© ENI Editions - All rigths reserved

type

Sert à préciser le type du composant. Les valeurs possibles sont `submit`, pour représenter un bouton de soumission de formulaire, et `reset`, pour obtenir un bouton d'effacement des champs de formulaire. La valeur par défaut est `submit`.

c. Balise form

Cette balise restitue une balise HTML correspondant à un formulaire de saisie. Elle accepte les principaux arguments suivants :

styleClass

Liste des classes de styles appliquées lorsque ce composant est restitué graphiquement dans l'interface web. Les noms des classes de styles sont séparés les uns des autres par le caractère espace.

target

Indique la frame dans laquelle le résultat de la soumission du formulaire sera affiché.

d. Balise inputHidden

Cette balise permet de représenter un champ de formulaire caché. Parmi ces attributs, les plus remarquables sont les suivants :

converter

Cet attribut sert à associer un convertisseur spécifique au composant. Le convertisseur en question doit être défini dans le fichier de configuration des ressources de l'application web, ou correspondre à l'un des convertisseurs standards.

converterMessage

La valeur de cet attribut correspond à un message à présenter à l'utilisateur en cas d'échec de la conversion. Ce message vient surcharger celui qui est éventuellement défini par le convertisseur associé au composant.

required

Cet attribut est un flag permettant de préciser si le composant doit, ou non, posséder une valeur au moment de la soumission du formulaire.

requiredMessage

Cet attribut permet d'indiquer le message à présenter à l'internaute lorsque le composant, dont l'attribut `required` est positionné sur `true`, ne possède pas de valeur au moment de la soumission du formulaire.

validator

Cet attribut sert à associer un validateur spécifique au composant. Le validateur en question doit être défini dans le fichier de configuration des ressources de l'application web, ou correspondre à l'un des validateurs standards.

validatorMessage

La valeur de cet attribut correspond à un message à présenter à l'utilisateur en cas d'échec de la validation. Ce message vient surcharger celui qui est éventuellement défini par le validateur associé au composant.

value

Cet attribut permet de spécifier la valeur du composant.

valueChangeListener

La valeur de cet attribut doit contenir une méthode d'expression référençant un gestionnaire d'événement capable

© ENI Editions - All rigths reserved

de prendre en charge les changements de valeurs réalisés dans le composant. Le gestionnaire d'événement doit obligatoirement être une méthode de type void et accepter un argument de type `ValueChangeEvent`.

e. Balise inputSecret

Cette balise est utilisée pour restituer une zone de texte de type mot de passe. Outre ses attributs `converter`, `converterMessage`, `required`, `requiredMessage`, `validator`, `validatorMessage`, `value`, `valueChangeListener`, dont le rôle est identique à celui qu'ils jouent dans le cadre de la balise `inputHidden`, la balise `inputSecret` peut exploiter les attributs suivants :

disabled

Cet attribut accepte une valeur booléenne chargée d'indiquer si le composant est utilisable ou non par l'internaute.

maxlength

La valeur de cet attribut correspond au nombre maximum de caractères pouvant être saisis dans ce composant.

redisplay

Cet attribut est utilisé pour indiquer que le mot de passe éventuellement précédemment saisi dans le composant doit être réaffiché à chaque nouvelle présentation du formulaire. Pour des raisons évidentes de sécurité, la valeur par défaut de cet attribut est `false`.

f. Balise inputText

Cette balise représente un champ de saisie de type texte. Ses attributs sont identiques à ceux de la balise `inputSecret`.

g. Balise inputTextArea

Cette balise représente une zone de texte multilignes, et dispose des attributs cités précédemment concernant la balise `inputHidden`. Elle présente en outre les deux attributs suivants :

cols

Cet attribut accepte une valeur numérique correspondant au nombre de colonnes caractérisant le composant.

rows

Cet attribut accepte une valeur numérique correspondant au nombre de lignes caractérisant le composant.

h. Balise message

Cette balise représente un message individuel, obligatoirement associé à un composant graphique situé dans la page web. Elle permet notamment de présenter les raisons qui ont provoqué une erreur de conversion ou de validation de la valeur saisie dans le composant associé. Les attributs de cette balise les plus couramment utilisés sont :

for

Cet attribut sert à préciser l'identifiant du composant JSF concerné par les messages.

errorClass, fatalClass, infoClass, warnclass

Ces quatre attributs sont utilisés pour désigner les classes de style CSS à appliquer pour des messages de niveau de sévérité "ERROR", "FATAL", "INFO" et "WARN".

i. Balise messages

La balise `messages` est utilisée pour l'affichage des messages d'erreur de conversion ou de validation de la valeur saisie dans l'un des composants JSF présent sur la page web. La plupart des attributs de cette balise sont les

© ENI Editions - All rigths reserved

mêmes que ceux de la balise message, à l'exception de l'attribut for, non disponible pour la balise messages dans la mesure où celle-ci ne concerne pas un composant JSF particulier.

j. Balise outputLink

Cette balise sert à restituer une ancre HTML. Son attribut le plus important est value : il permet de définir la valeur de l'attribut href de l'ancre. Les principaux autres attributs sont :

converter

Cet attribut est utilisé pour affecter une instance de la classe Converter au composant.

disabled

L'utilisation de cet attribut est nécessaire pour indiquer que le composant ne doit jamais être en mesure de recevoir le focus, ou être pris en compte au moment de la soumission du formulaire.

target

Comme c'est le cas pour la balise HTML représentant une ancre, cet attribut target permet de spécifier le nom de la frame dans laquelle le document ciblé par le composant devra s'afficher.

k. Balise outputText

Cette balise sert à représenter un texte brut au sein d'une page web. Dans le cas où l'un des attributs styleClass, style, dir ou lang est utilisé, le texte brut en question est restitué dans la page en étant encadré par des balises et .

converter

Cet attribut est utilisé pour associer un convertisseur au composant.

value

Permet de spécifier la chaîne correspondant au texte brut à présenter dans la page.

l. Balise panelGrid

Cette balise est chargée de restituer un tableau HTML composé d'un certain nombre de colonnes. Le nombre de lignes est déterminé dynamiquement compte tenu du nombre de composants à positionner dans le tableau. Dans l'exemple suivant, la balise panelGrid rassemble quatre composants, représentés par des balises outputText.

```
<h:panelGrid border="1" columns="2" bgcolor="red" width="30%">
     <h:outputText value="item1"></h:outputText>
     <h:outputText value="item2"></h:outputText>
     <h:outputText value="item3"></h:outputText>
     <h:outputText value="item4"></h:outputText>
</h:panelGrid>
```

La restitution correspondante est un tableau HTML de deux lignes et deux colonnes.

```
<table bgcolor="red" border="1" width="30%"><tbody>
<tr><td>item1</td><td>item2</td></tr>
<tr><td>item3</td><td>item4</td></tr>
</tbody></table>
```

Voici les principaux attributs disponibles pour cette balise :

bgcolor

Cet attribut sert à indiquer la couleur de fond du tableau.

border

© ENI Editions - All rigths reserved

La valeur de cet attribut indique l'épaisseur des bordures du tableau.

width

Cet attribut permet de spécifier, comme en HTML, la largeur du tableau.

footerClass, **headerClass**

Ces attributs servent à mentionner la liste des classes de styles applicables aux en-têtes et pieds de colonnes.

m. Balise selectBooleanCheckbox

Cette balise permet de restituer un champ de saisie de type case à cocher. Elle dispose des attributs `converter`, `converterMessage`, `required`, `requiredMessage`, `validator`, `validatorMessage`, `value`, `valueChangeListener`, déjà évoqués pour la balise `inputHidden`.

n. Balise selectManyCheckbox

Ce composant permet de restituer un tableau HTML contenant un ensemble de cases à cocher. Chaque case à cocher individuelle peut être représentée par le biais d'une balise `<f:selectItem>`, issue de la bibliothèque Core.

Si de nombreuses cases à cocher doivent être présentées, il est alors préférable d'utiliser une unique balise `<f:selectItems>`, plutôt qu'un ensemble de balises `<f:selectItem>`. La présentation de la balise `selectManyListBox`, située plus loin dans ce paragraphe, montre un usage possible de la balise `<f:selectItems>`.

```
<h:selectManyCheckbox border="1" id="mesChoix">
          <f:selectItem itemLabel="choix n°1" itemValue="1"/>
          <f:selectItem itemLabel="choix n°2" itemValue="2"/>
          <f:selectItem itemLabel="choix n°3" itemValue="3"/>
</h:selectManyCheckbox>
```

La restitution dans une page HTML peut correspondre au code source suivant :

```
<table border="1" id="j_id_jsp_1719144863_7:mesChoix">
<tr>
<td>
<input name="j_id_jsp_1719144863_7:mesChoix"
      id="j_id_jsp_1719144863_7:mesChoix:0"
     value="1"
     type="checkbox" />
<label for="j_id_jsp_1719144863_7:mesChoix:0">choix n&deg;1</label>
</td>
<td>
<input name="j_id_jsp_1719144863_7:mesChoix"
      id="j_id_jsp_1719144863_7:mesChoix:1"
     value="2"
     type="checkbox" />
<label for="j_id_jsp_1719144863_7:mesChoix:1">
choix n&deg;2</label>
</td>
<td>
<input name="j_id_jsp_1719144863_7:mesChoix"
      id="j_id_jsp_1719144863_7:mesChoix:2"
     value="3"
     type="checkbox" />
<label for="j_id_jsp_1719144863_7:mesChoix:2">choix n&deg;3</label>
</td>
     </tr>
</table>
```

Les attributs de la balise `selectManyCheckbox` sont identiques à ceux de la balise `selectBooleanCheckbox`. Il est néanmoins possible d'indiquer, grâce à un attribut `layout`, le sens d'affichage des cases à cocher : si la valeur de cet attribut est égale à `pageDirection`, alors les cases seront présentées en colonne. Si l'attribut en question n'est

© ENI Editions - All rigths reserved

pas mentionné, ou si sa valeur est lineDirection, alors les cases à cocher sont affichées sur une ligne unique.

o. Balise selectOneListbox

Cette balise sert à représenter une zone de liste non déroulante à choix unique, correspondant à la balise HTML <select>. Chacun des éléments sélectionnables est identifié par une balise <f:selectItem>, dont les attributs permettent de spécifier la valeur et le label de l'élément en question. Il est aussi possible de restituer tous les éléments au travers d'une unique balise <f:selectItems>.

Les principaux attributs de la balise selectOneListbox sont les mêmes que ceux de la balise selectBooleanCheckbox. Un attribut size permet toutefois de définir le nombre d'éléments visibles dans la zone de liste.

```
<h:selectOneListbox id="mesChoix" >
          <f:selectItem itemValue="1" itemLabel="choix n°1"/>
          <f:selectItem itemValue="2" itemLabel="choix n°2"/>
          <f:selectItem itemValue="3" itemLabel="choix n°3"/>
</h:selectOneListbox>
```

p. Balise selectManyListbox

Cette balise permet de générer une zone de liste non déroulante à choix multiples. Elle est donc restituée par une balise HTML <select> disposant d'un attribut multiple. Comme c'est le cas pour la balise selectOneListbox, les éléments sélectionnables peuvent être représentés soit par plusieurs balises <f:selectItem>, soit par une unique <f:selectItems>.

```
<h:selectManyListbox size="12">
     <f:selectItems value="#{mesPrix}"/>
</h:selectManyListbox>
```

Dans l'exemple ci-dessus, les éléments de la zone de liste sont définis par une balise <f:selectItems>, dont l'attribut value référence un bean managé nommé mesPrix. Ce bean peut, par exemple, être issu d'une classe dérivée de ArrayList et contenir la totalité des éléments sélectionnables, qui sont en réalité des instances de la classejavax.faces.model.SelectItem.

```
package premierProjetJSF;

import java.util.ArrayList;
import javax.faces.model.SelectItem;

public class Prix extends ArrayList {

     public Prix()
     {
          SelectItem element;
          for (int i=1;i<=100;i++)
          {
               element=new SelectItem(
                          new Integer(i),
                          i+".00 euros",
                          "Produit à "+i+".00 euros");
               add(element);
          }
     }
}
```

En dehors, notamment, de l'attribut size qui permet d'indiquer le nombre d'éléments visibles dans la zone de liste, les principaux attributs de la balise selectManyListbox sont les mêmes que ceux de la balise selectBooleanCheckbox.

q. Balise selectOneMenu

Cette balise s'utilise pour restituer une zone de liste déroulante à choix unique. Elle ne dispose d'aucun attribut permettant de préciser le nombre d'éléments visibles : celui-ci est automatiquement fixé à 1 au moment de la restitution. De manière identique à ce qui a été précisé pour les balises selectOneListBox et selectManyListBox, les éléments de la zone de liste déroulante à choix unique peuvent être représentés par un groupe de balises

© ENI Editions - All rigths reserved

<f:selectItem> ou par une unique balise <f:selectItems>.

Les principaux attributs de la balise selectOneMenu sont les mêmes que ceux de la balise selectBooleanCheckbox.

r. Balise selectManyMenu

Tout comme la balise selectManyListBox, la balise selectManyMenu permet de restituer une zone de liste non déroulante à choix multiples. Elle dispose cependant de la particularité de ne permettre la visualisation que d'un seul élément à la fois. La balise HTML restituée est donc <select>, avec un attribut multiple, ainsi qu'un attribut size de valeur égale à 1.

s. Balise selectOneRadio

Cette balise permet de représenter un groupe de boutons radio. Les choix possibles peuvent être définis à l'aide d'un ensemble de balises <f:selectItem> ou par l'intermédiaire d'une unique balise <f:selectIems>.

Les principaux attributs de la balise selectOneRadio sont les mêmes que ceux de la balise selectBooleanCheckbox. Il existe toutefois un attribut layout permettant d'indiquer le sens de l'affichage des boutons radio (en ligne ou en colonne).

```
<h:selectOneRadio id="choix5">
     <f:selectItem itemValue="1" itemLabel="choix n°1"/>
     <f:selectItem itemValue="2" itemLabel="choix n°2"/>
     <f:selectItem itemValue="3" itemLabel="choix n°3"/>
</h:selectOneRadio>
```

© ENI Editions - All rigths reserved

Principaux éléments de la bibliothèque Core

Ce paragraphe présente les principales balises contenues dans la bibliothèque Core, ainsi que leurs attributs les plus caractéristiques.

1. Balise actionListener

Cette balise sert à affecter une instance d'une classe implémentant `ActionListener` à un composant JSF particulier, dans le but de gérer les événements de type `ActionEvent` survenant sur le composant en question. L'exemple suivant illustre l'utilisation d'une balise `actionListener` associée à une balise `commandButton`.

```
<h:commandButton id="btnEnregistrer" value="Enregistrer" >
<f:actionListenertype="premierProjetJSF.listeners.BeanActionListener"/>
</h:commandButton>
```

L'attribut `type` permet de mentionner le nom complet de la classe chargée de gérer l'`ActionEvent`. Cette classe doit obligatoirement implémenter l'interface `javax.faces.event.ActionListener`.

2. Balise convertDateTime

La balise `convertDateTime` est utilisée pour affecter un convertisseur de type `DateTimeConverter` à un composant particulier. Cela peut permettre, par exemple, de s'assurer que la valeur saisie dans un champ correspond bien à une date spécifiée dans un format donné. L'exemple suivant montre l'association d'une balise `convertDateTime` avec une balise `inputText`.

```
<h:inputText id="itxtDateDuJour" >
<f:convertDateTime pattern="dd/MM/yyyy" />
</h:inputText>
```

a. Attribut dateStyle

Cet attribut permet de spécifier comment la date doit être formatée. Les valeurs autorisées sont `default`, `short`, `medium`, `long`, et `full`. Cet attribut n'est pris en compte que si l'attribut `type` est positionné sur `date` ou `both`.

b. Attribut pattern

L'attribut `pattern` est employé pour mentionner un format de date personnalisé. Ce format doit respecter la syntaxe mentionnée dans la classe `java.text.SimpleDateFormat`.

c. Attribut timeStyle

Cet attribut sert à préciser comment l'heure doit être formatée. Cet attribut n'est pris en compte que si l'attribut `type` vaut `time` ou `both`. Les valeurs autorisées pour l'attribut `timeStyle` sont `default`, `short`, `medium`, `long`, et `full`.

d. Attribut type

Permet de préciser le type de conversion : date uniquement, heure uniquement, ou les deux. Les valeurs que peut prendre cet attribut sont `date`, `time` ou `both`.

3. Balise convertNumber

Cette balise permet d'associer un convertisseur de type `NumberConverter` à un composant JSF particulier, pour assurer un formatage correct des nombres, pourcentages et devises. Le code source suivant montre l'association d'une balise `convertNumber` avec une balise `inputText`.

© ENI Editions - All rigths reserved

```
<h:inputText
     id="itxtValeurNumerique"
     converterMessage="Format non respecté">
<f:convertNumber pattern="0.00%"/>
</h:inputText>
```

Les principaux attributs de cette balise sont les suivants :

a. Attribut pattern

Cet attribut permet de personnaliser le format à utiliser pour convertir la valeur du composant JSF.

b. Attribut type

Cet attribut sert à indiquer le type de la valeur du composant. Les valeurs autorisées sont `number`, `currency` et `percentage`.

4. Balise converter

La balise `converter` est utilisée pour associer un convertisseur à un composant JSF particulier.

```
<h:inputText id="txtSaisie">
<f:converter converterId="monConvertisseurNombrePairPositif"/>
</h:inputText>
```

Le principal attribut de cette balise est `converterId`. Celui-ci référence un convertisseur personnalisé déclaré dans le fichier de configuration des ressources de l'application web. Ce convertisseur est obligatoirement issu d'une classe implémentant l'interface `javax.faces.conver.Converter`.

5. Balise loadBundle

Cette balise permet d'instancier un `ResourceBundle` particulier au sein d'une page web donnée.

```
<f:view>
<f:loadBundle
basename="premierProjetJSF.messages"
var="mesMessagesPerso"/>
...
</f:view>
```

Cette balise ne possède que les deux attributs suivants :

a. Attribut baseName

Cet attribut sert à préciser le nom de la famille du `ResourceBundle` à instancier.

b. Attribut var

Il est utilisé pour attribuer un identifiant spécifique au `ResourceBundle` créé, de manière à le rendre utilisable dans la page JSP.

6. Balise selectItem

Cette balise permet d'associer une instance de la classe `UISelectItem` à un composant JSF particulier.

Comme cela a été expliqué précédemment dans la section Principaux éléments de la bibliothèque HTML de ce chapitre, cette balise peut notamment être associée aux balises `selectManyCheckbox`, `selectManyListbox`,

© ENI Editions - All rigths reserved

selectManyMenu, selectOneListbox et selectOneRadio. De telles utilisations servent à définir les éléments sélectionnables de zones de liste (déroulantes ou non), de cases à cocher, de boutons radios.

a. Attribut itemDisabled

Cet attribut permet d'indiquer si l'élément sélectionnable, représenté par la balise selectItem, peut ou non être utilisable par l'internaute.

b. Attribut itemLabel

La valeur de cet attribut correspond au label présenté à l'utilisateur dans l'interface web.

c. Attribut itemValue

La valeur de cet attribut correspond à la valeur renvoyée au serveur si l'élément en question est sélectionné au moment de la soumission du formulaire.

7. Balise selectItems

Cette balise permet d'associer une instance de la classe UISelectItems à un composant JSF particulier. Elle peut donc notamment servir à définir l'ensemble des éléments sélectionnables d'une zone de liste (déroulante ou non), cases à cocher ou boutons radio. Un exemple d'utilisation de cette balise est présenté à la section Principaux éléments de la bibliothèque HTML de ce chapitre, dans la rubrique relative à la balise selectManyListbox.

L'attribut le plus important de cette balise est value : il est utilisé pour identifier une List ou une ArrayList dans laquelle est regroupée la totalité des éléments sélectionnables du composant JSF associé.

8. Balise validateDoubleRange

Cette balise permet d'associer une instance de la classe DoubleRangeValidator à un composant JSF particulier. Cette action est destinée à contrôler la valeur de ce composant, qui peut notamment être un nombre décimal. L'exemple suivant correspond à l'association de la balise validateDoubleRange avec une balise inputText.

```
<h:inputText id="txtDuree">
<f:validateDoubleRange minimum="10" maximum="100">
</f:validateDoubleRange>
</h:inputText>
```

Les deux principaux attributs de cette balise sont minimum et maximum. Ils permettent de définir respectivement les valeurs minimales et maximales autorisées pour le composant JSF associé.

9. Balise validateLength

Cette balise permet d'associer une instance de la classe LengthValidator à un composant JSF particulier, dans le but de contrôler la longueur de la chaîne de caractères saisie dans le composant en question.

```
<h:inputText id="txtCommentaire"
label="Commentaire impératif"
required="true">
<f:validateLengthminimum="5"
     maximum="10">
</f:validateLength>
</h:inputText>
```

Les deux principaux attributs de cette balise sont minimum et maximum. Ils permettent de définir respectivement les longueurs de chaîne minimale et maximale autorisées pour le composant JSF associé.

© ENI Editions - All rigths reserved

10. Balise validateLongRange

Cette balise joue un rôle similaire à celui de la balise validateDoubleRange : associée à un composant JSF particulier, elle sert à définir un validateur chargé de contrôler la valeur du composant. Ce validateur est de type LongRangeValidator : si celui-ci est capable d'assurer le contrôle des valeurs numériques entières, il ne peut en aucun cas prendre en charge le traitement de valeurs numériques décimales, contrairement au DoubleRangeValidator.

Les deux principaux attributs de la balise validateLongRange sont minimum et maximum. Ils permettent de définir respectivement les valeurs minimales et maximales autorisées pour le composant JSF associé.

11. Balise validator

Cette balise permet d'associer un validateur personnalisé à un composant JSF spécifique. Dans l'exemple suivant, la balise validator est associée à une balise inputSecret : le validateur personnalisé peut par exemple être chargé de vérifier la conformité d'un mot de passe compte tenu d'un certain nombre de contraintes.

```
<h:inputSecret value="#{loginBean.password}" id="champMotPasse">
<f:validator validatorId="monValidateurDeMotDePasse" />
</h:inputSecret>
```

Le principal attribut de cette balise est validatorId. Celui-ci identifie le validateur personnalisé, tel qu'il a été défini dans le fichier de configuration des ressources de l'application web. Ce validateur doit obligatoirement être issu d'une classe implémentant l'interface javax.faces.validator.Validator.

12. Balise valueChangeListener

Cette balise sert à associer une instance de classe implémentant l'interface ValueChangeListener à un composant JSF particulier, dans le but de capturer les événements de type ValueChangeEvent générés par ce composant. Le code suivant montre comment une balise valueChangeListener peut être associée à une balise inputTextarea.

```
<h:inputTextarea id="itaCommentaires">
<f:valueChangeListener type="premierProjetJSF.listeners.BeanValueChange
Listener"/>
</h:inputTextarea>
```

Le principal attribut de cette balise est type : il sert à indiquer le nom complet de la classe représentant le ValueChangeListener.

13. Balise view

Cette balise tient lieu de conteneur pour l'ensemble des balises issues des bibliothèques HTML et Core.

Elle présente deux attributs principaux : beforePhase et afterPhase.

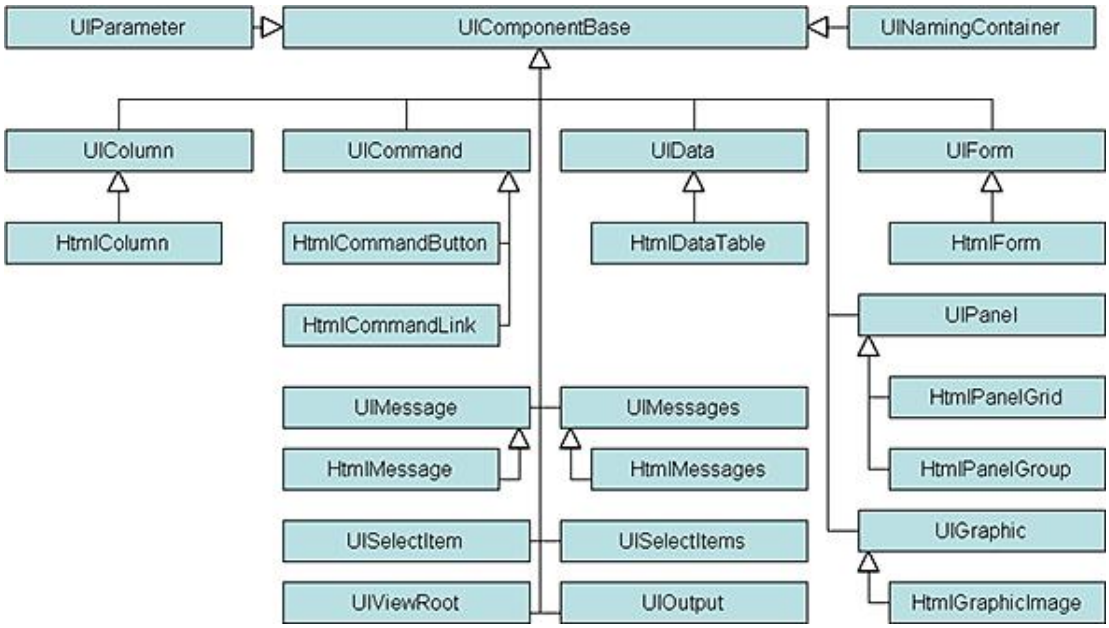
L'attribut beforePhase permet de spécifier une méthode dont l'invocation se produit systématiquement avant chaque changement de phase (hormis la phase de restauration de la vue). La méthode en question doit obligatoirement être de type void et accepter un argument de type javax.faces.event.PhaseEvent.

L'attribut afterPhase permet quant à lui de mentionner une méthode destinée à être exécutée à l'issue de chaque changement de phase (hormis la phase de restauration de la vue). Cette méthode doit nécessairement présenter les mêmes caractéristiques que celle évoquée dans le cas de l'attribut beforePhase.

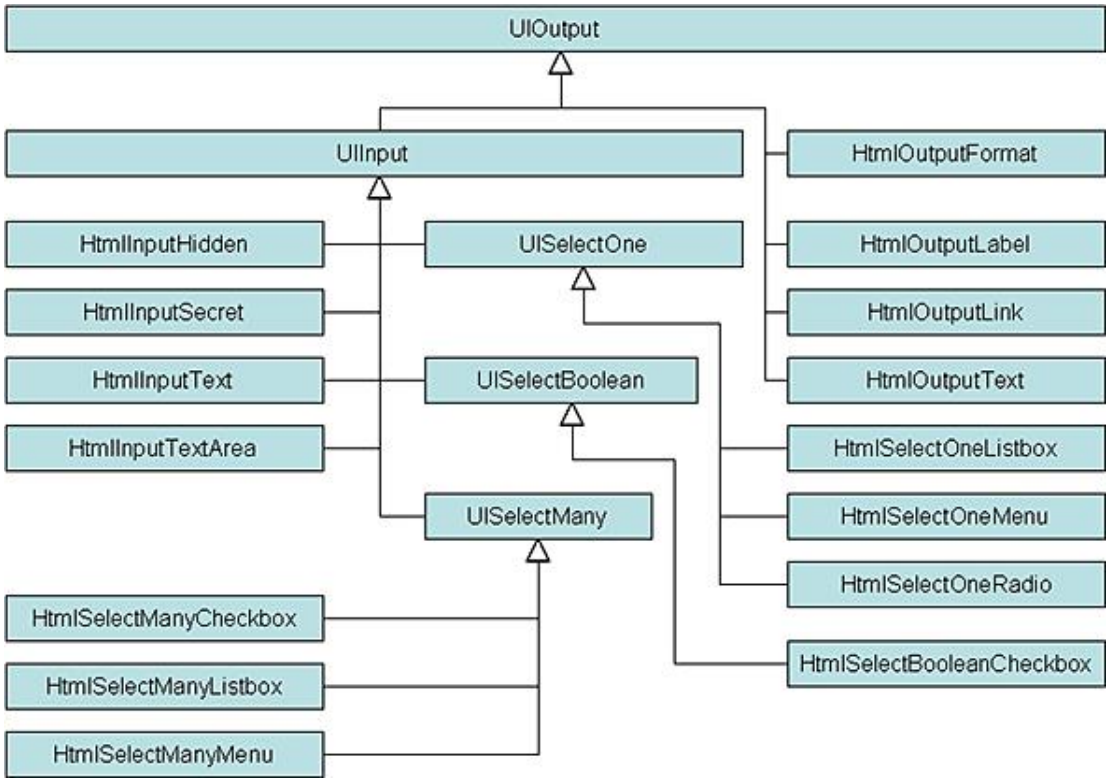
© ENI Editions - All rigths reserved

Diagramme de classes relatif aux composants standards JSF

Voici les principales classes issues de `UIComponentBase`.



Il existe également de nombreux composants JSF instanciés à partir de classes dérivées de `UIOutput`. Les classes en question sont représentées dans le diagramme suivant :



© ENI Editions - All rigths reserved

Introduction

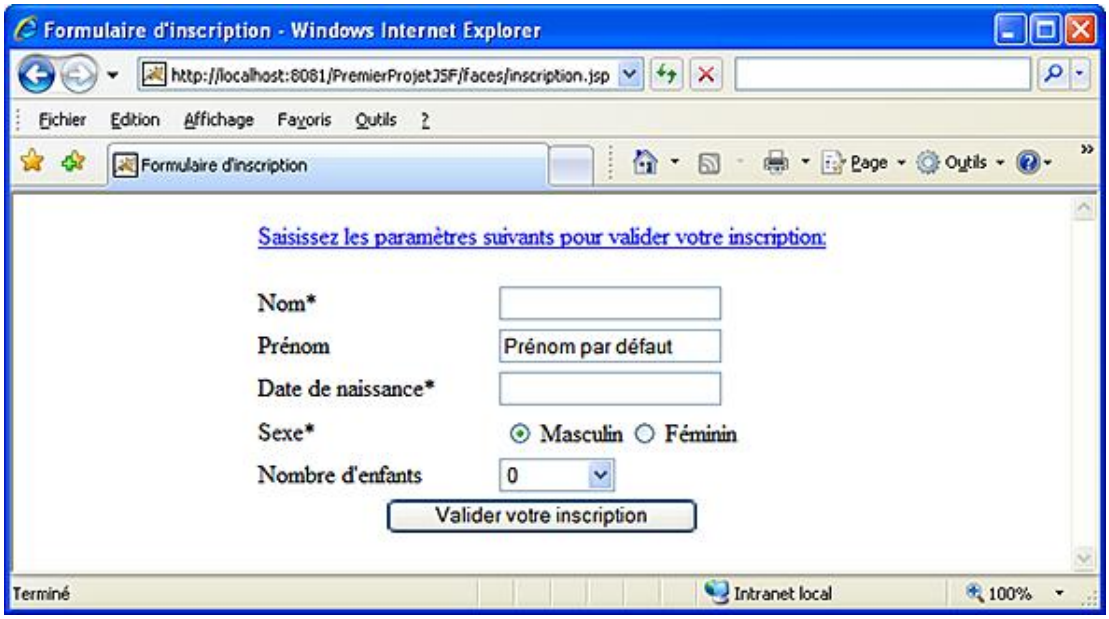
Ce chapitre présente les aspects théoriques mis en œuvre dans la technologie Java Server Faces. À partir d'un exemple simple, servant d'illustration pour l'ensemble du chapitre, les étapes de création d'une page web exploitant des composants JSF seront abordées. Après une présentation des multiples composants graphiques mis à la disposition du développeur, les processus de validation et de conversion des données saisies sont expliqués. Les principes généraux mis en œuvre dans la liaison de composants à des JavaBeans sont également vus, ainsi que l'étude du modèle de navigation exploité par JSF. Enfin, ce chapitre se termine par une présentation des différentes phases constituant le cycle de vie d'une page.

© ENI Editions - All rigths reserved

Description de l'application JSF à développer

Le but de ce paragraphe est de présenter l'application web qui servira d'illustration aux différents points abordés dans ce chapitre. Au travers d'un exemple simple sur le plan conceptuel, il sera possible de traiter les aspects essentiels de la technologie Java Server Faces, afin d'en avoir rapidement une vue assez claire. Pour le moment, le travail dans l'environnement de développement Eclipse n'est pas abordé : cela fera l'objet des chapitres suivants de cet ouvrage. L'accent est plutôt mis ici sur les concepts JSF, qui peuvent naturellement être appliqués à d'autres outils, tels que NetBeans et Java Sun Studio Creator.

L'application web à développer dans ce chapitre est constituée de trois pages JSP. La première d'entre elles, la plus importante, propose un formulaire destiné à permettre la saisie d'informations liées à l'identité d'une personne. La figure présentée ci-dessous en est une illustration. Les deux autres pages de l'application web, dont les codes source ne seront pas présentés compte tenu de leur simplicité, sont les pages vers lesquelles la navigation pourra être orientée au moment de la validation du formulaire. La première de ces deux pages devra être présentée à l'internaute en cas de réussite de l'inscription, la seconde devra l'être dans le cas contraire.

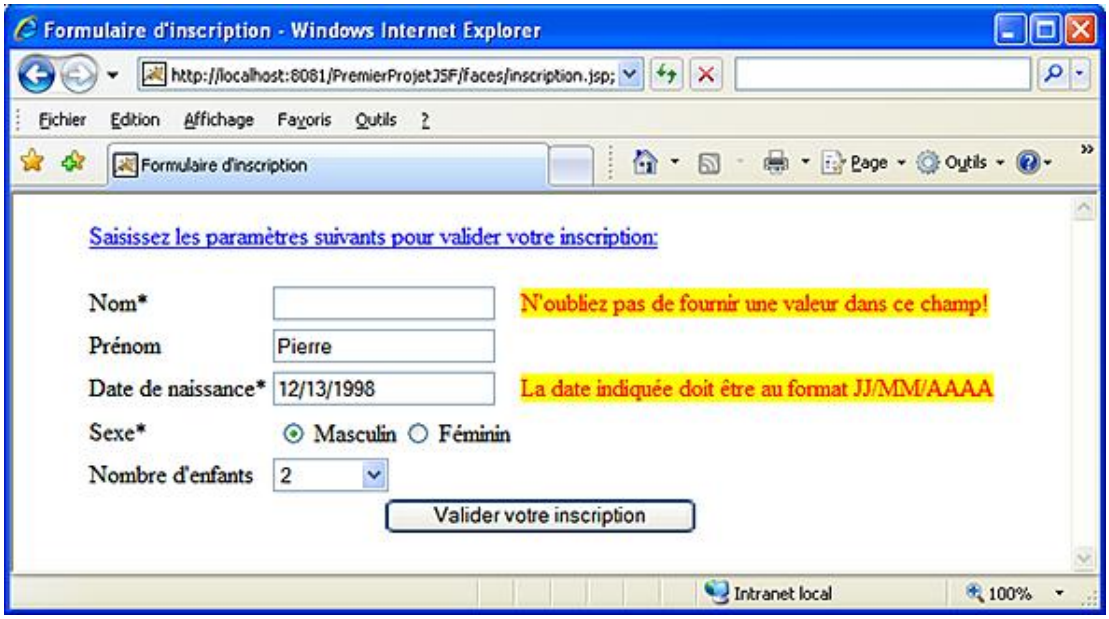


Outre les habituels éléments de formulaire (zones de texte, cases à cocher, zone de liste, etc.), la page principale de l'application comporte des contraintes destinées à garantir la fiabilité des données saisies. En l'occurrence :

- Les champs **Nom**, **Date de naissance** et **Sexe** doivent obligatoirement être renseignés.
- Le champ **Nom** doit contenir un texte de longueur comprise entre 5 et 100 caractères. Une telle contrainte est, par exemple, tout à fait envisageable dans le cas où le nom doit être par la suite stocké dans un champ de base de données dont la longueur ne peut pas excéder 100 caractères.
- Le champ **Date de naissance** doit obligatoirement être au format JJ/MM/AAAA. Par exemple, 17/07/2008.

En cas de non-respect des contraintes, des messages d'erreurs explicites devront être présentés à l'internaute, en regard des champs de saisie concernés, dès la validation du formulaire.

© ENI Editions - All rigths reserved



© ENI Editions - All rigths reserved

Architecture et fichiers de configuration

1. Architecture d'une application JSF

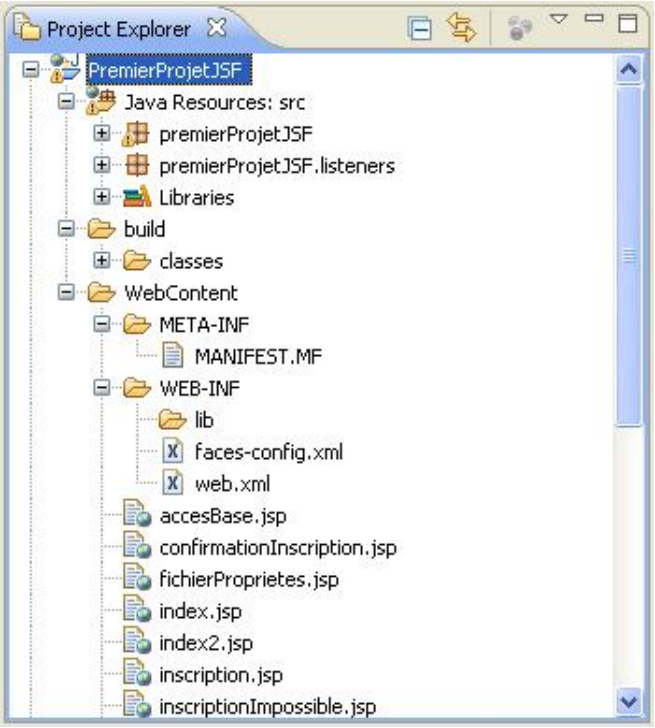
Une application Java Server Faces doit obligatoirement respecter la spécification relative aux servlets (version 2.3 ou plus récente), ainsi que la spécification Java Server Pages (version 1.2 ou plus récente). En conséquence, l'architecture d'une application JSF est similaire à celle d'une application web classique mise en œuvre par un conteneur de servlets : l'arborescence est constituée de trois dossiers :

- Le premier, par défaut nommé **src**, est destiné au stockage du code source des classes Java utilisées par l'application (JavaBean, écouteurs, validateurs personnalisés, convertisseurs personnalisés, etc.). Il sert aussi à l'archivage des fichiers d'extension **.properties**, exploités notamment par l'application pour faciliter son internationalisation.

- Le second regroupe le code compilé des classes Java évoquées précédemment. Ce dossier se nomme **build**.

- Le dernier dossier, appelé par défaut **WebContent**, contient les ressources destinées à être publiées par le moteur de servlets, principalement les pages JSP/JSF. Ce répertoire a également pour rôle de regrouper les fichiers de configuration de l'application web, à savoir le descripteur de déploiement, ainsi que le(s) fichier(s) de configuration des ressources de l'application (voir ci-dessous).

La figure suivante montre l'architecture d'une application JSF, telle qu'elle se présente dans l'explorateur de projets de l'environnement de développement Eclipse.



Le nom des répertoires **src** et **WebContent** peut être choisi librement au moment de la création de l'application. Dans le cas où les valeurs par défaut ne sont pas utilisées, les choix réalisés sont mémorisés dans le fichier de configuration du projet.

De même, lorsque le nom du fichier de configuration des ressources de l'application JSF est personnalisé (son nom par défaut est **faces-config.xml**), ou s'il existe plusieurs fichiers de configuration de ressources, les changements sont répertoriés au sein du descripteur de déploiement par l'intermédiaire d'une balise `<context-param>`, comme le montre le code source suivant :

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
```

© ENI Editions - All rigths reserved

```
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
id="WebApp_ID"
version="2.5">
  <display-name>deuxiemeJSF</display-name>
...
<context-param>
    <param-name>javax.faces.CONFIG_FILES</param-name>
    <param-value>/maConfig/faces2-config2.xml</param-value>
  </context-param>
</web-app>
```

2. Fichiers de configuration des ressources d'une application JSF

La technologie Java Server Faces est conçue pour permettre la configuration des ressources exploitables par une application web au moyen d'un ou de plusieurs fichiers de configuration. Ces fichiers sont appelés fichiers de configuration des ressources : ils servent à répertorier et initialiser les ressources telles que les JavaBeans et les règles de navigation.

a. Localisation du/des fichier(s) de configuration des ressources

Comme cela a été précisé dans le paragraphe précédent, une application JSF de base utilise un unique fichier de configuration des ressources dénommé **faces-config.xml**, mais il est possible d'en définir davantage. Ces fichiers XML contiennent tous une balise racine `<faces-config>` encadrant d'autres balises enfants représentant les ressources gérées. Un exemple de contenu de fichier de configuration de ressources est présenté plus loin dans ce chapitre.

Au moment du démarrage de l'application web, le ou les fichiers de configuration des ressources sont recherchés dans l'arborescence selon l'un des procédés suivants :

- Recherche d'un fichier **/META-INF/faces-config.xml** dans l'un des fichiers JAR éventuellement présents dans le répertoire **/WEB-INF/lib** de l'application web. Si ce fichier est trouvé, il est exploité en tant que fichier de configuration des ressources.

- Recherche d'un paramètre initial nommé javax.faces.application.CONFIG_FILES dans le descripteur de déploiement de l'application web. Lorsque ce paramètre est trouvé, sa valeur contient la liste des noms de fichiers de configuration des ressources, séparés par des virgules. Ce mode de déclaration est pratique lorsque l'application JSF est de taille importante et que son architecture est confiée à plusieurs personnes.

- Recherche d'un fichier **faces-config.xml** dans le répertoire **/WEB-INF** de l'application. C'est le moyen le plus simple de configurer une application JSF ; c'est aussi le choix retenu par l'environnement Eclipse lorsque les paramètres proposés par défaut au moment de la création d'un projet JSF sont retenus.

b. Mode d'exploitation

Toutes les applications JSF sont représentées en mémoire sous la forme d'une instance de la classe `javax.faces.application.Application`. Cet objet est utilisé notamment pour regrouper toutes les ressources utilisables.

Au démarrage de l'application, le(s) fichier(s) de configuration des ressources sont parcourus pour déterminer la liste des objets à créer puis ajouter à l'objet `Application`. Plusieurs méthodes distinctes de la classe `Application` sont utilisées pour réaliser ces opérations. Voici leurs signatures :

```
public abstract Converter createConverter(String converterId)
public abstract Converter createConverter(Class targetClass)

public abstract Validator createValidator(String validatorId)
                                  throws FacesException


public abstract UIComponent createComponent(String componentType)
                                    throws FacesException
public abstract UIComponent createComponent(
                                     ValueBinding componentBinding,
```


© ENI Editions - All rigths reserved

```
                                        FacesContext context,
                                        String componentType)
                                       throws FacesException
public UIComponent createComponent(
                                ValueExpression componentExpression,
                                FacesContext context,
                                String componentType)
                             throws FacesException
```

Les méthodes nommées `createConverter(…)` sont utilisées pour instancier les convertisseurs personnalisés, alors que la méthode `createValidator(…)` sert à mettre à la disposition de l'application les validateurs personnalisés. Quant aux méthodes `createComponent(…)`, elles sont destinées à assurer la création des autres types de composants JSF personnalisés.

Le code source ci-dessous est un extrait de fichier de configuration de ressources, dans lequel sont déclarés un convertisseur personnalisé, un validateur personnalisé, un bean managé et une règle de navigation. Ces notions seront étudiées plus en détail dans la suite de ce chapitre, section Navigation entre les pages, ainsi qu'au chapitre Convertisseurs et validateurs.

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

<faces-config
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd"
    version="1.2">
     <converter>
          <display-name>
          convertisseurNombrePairPositif</display-name>
          <converter-id>
          monConvertisseurNombrePairPositif</converter-id>
          <converter-class>
          premierProjetJSF.NombrePairPositifConverter</converter-class>
     </converter>

     <validator>
          <display-name>
          Validation du mot de passe</display-name>
          <validator-id>
          monValidateurDeMotDePasse</validator-id>
          <validator-class>
          premierProjetJSF.ValidatorPassword</validator-class>
     </validator>

     <managed-bean>
          <managed-bean-name>
          tirage</managed-bean-name>
          <managed-bean-class>
          premierProjetJSF.Tirage</managed-bean-class>
          <managed-bean-scope>
          session</managed-bean-scope>
          <managed-property>
               <property-name>
               maximum</property-name>
               <property-class>
               int</property-class>
               <value>
               10</value>
          </managed-property>
          <managed-property>
               <property-name>
               minimum</property-name>
               <property-class>
               int</property-class>
               <value>
               0</value>
          </managed-property>
     </managed-bean>
```

© ENI Editions - All rigths reserved

```
    <navigation-rule>
        <display-name>
        inscription</display-name>
        <from-view-id>
        /inscription.jsp</from-view-id>
        <navigation-case>
            <from-outcome>succesInscription</from-outcome>
            <to-view-id>
            /confirmationInscription.jsp</to-view-id>
        </navigation-case>
    </navigation-rule>
</faces-config>
```

c. Types de ressources configurables

Dans un souci de clarté du texte dans le reste de l'ouvrage, nous considérerons que les ressources des applications web évoquées sont configurées grâce à un unique fichier de configuration des ressources.

Javabeans

La déclaration d'un JavaBean ou d'un bean managé dans le fichier de configuration des ressources se fait à l'aide de la balise `<managed-bean>`. Celle-ci regroupe un ensemble de balises enfants dont le nom et le rôle sont précisés ci-dessous :

`<managed-bean-name>`

Définit la clé attribuée au JavaBean. Cette clé, valable sur la portée prévue pour le JavaBean, pourra être utilisée dans toutes les pages web de l'application JSF pour référencer ce JavaBean en particulier.

`<managed-bean-class>`

Indique le nom de la classe à partir de laquelle doit être instancié le JavaBean. Cette classe doit bien sûr être rendue accessible par l'application.

`<managed-bean-scope>`

Précise la portée dans laquelle le JavaBean est instancié. Les valeurs possibles pour ce paramètre sont `none`, `request`, `session` et `application`. L'utilisation de la portée `none` permet d'instancier systématiquement un nouveau JavaBean à chaque fois que celui-ci est référencé dans la page. Dans ce cas bien sûr, le JavaBean n'est rattaché à aucune portée particulière.

`<managed-property>`

Sert à définir la valeur d'initialisation d'une propriété particulière du JavaBean.

La définition d'un JavaBean peut contenir plusieurs définitions de propriétés.

La valeur de la propriété est déclarée grâce à une balise enfant nommée `<value>`. Son type est indiqué par la balise enfant `<property-class>`, alors que le nom même de la propriété est mentionné par la balise enfant `<property-name>`.

Le fait de déclarer les JavaBeans au sein du fichier de configuration des ressources présente plusieurs avantages. Notamment la possibilité de faire les déclarations dans un unique fichier utilisable par l'ensemble de l'application, et la possibilité de préciser facilement la valeur des propriétés sans écrire de lignes de code Java supplémentaires.

De plus amples détails concernant l'usage de l'élément `<managed-bean>` sont donnés dans le chapitre Exploitation des composants standards JSF.

Messages et labels personnalisés

Comme présenté plus loin dans ce chapitre, les applications JSF sont conçues pour faciliter l'affichage des messages d'erreurs survenant au cours des processus de validation ou de conversion des données. Les contenus de ces messages, qui peuvent être exposés à l'internaute par l'intermédiaire des balises `<h:message>` ou `<h:messages>`, peuvent être définis de manière dynamique compte tenu de la logique métier de l'application. Ils peuvent aussi être définis de manière statique au sein d'un ou plusieurs fichiers de propriétés, dont les références sont mentionnées dans le fichier de configuration des ressources.

© ENI Editions - All rigths reserved

Ce même type de fichier de propriétés peut également être utilisé pour spécifier un ensemble d'étiquettes, utilisables par exemple en tant que labels de champs de saisie de formulaires web.

Un fichier de propriétés est un simple fichier texte dont le nom possède l'extension **.properties**. Généralement placé dans le sous-répertoire **src** de l'application web, il regroupe un ensemble de lignes ; chacune d'elles représentant un message particulier. Un message spécifique est indiqué grâce à son identifiant et à sa valeur. Voici un exemple de fichier de propriétés dont le contenu peut être utilisé pour fournir des étiquettes aux champs de saisie de formulaires :

```
name=Nom
password=Mot de passe
clientcode=Code client
welcome=Bienvenue!
submit=Validez
login=Se connecter
```

Pour que ce fichier de propriétés soit utilisable par l'application JSF, celle-ci doit en connaître la référence par l'intermédiaire du fichier de configuration des ressources. Par exemple, le fichier de propriétés dont le contenu est présenté ci-dessus peut être déclaré de la façon suivante :

```
<faces-config    xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd"
   version="1.2">
       ...
       <application>
             <message-bundle>
             premierProjetJSF.messages
             </message-bundle>
             ...
       </application>
       ...
</faces-config>
```

La déclaration se fait donc à l'aide d'une balise `<message-bundle>`, elle-même contenue dans un élément `<application>`, directement placé sous `<faces-config>`. Le fichier de propriétés se nomme **messages.properties** et se trouve placé dans le sous-répertoire **src/premierProjetJSF** de l'application web.

Pour information (ce point est abordé plus en détail au chapitre Internationalisation d'une application web), l'usage de ces messages dans une page web de l'application peut se faire en chargeant la ressource grâce à une balise `<f:loadBundle>`, puis en utilisant la référence de cette ressource pour afficher une étiquette de champ de saisie. Le code source suivant illustre cette situation :

```
%@taglib uri="http://java.sun.com/jsf/html" prefix="h"%
<%@taglib     uri=http://java.sun.com/jsf/core  prefix="f"%>
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html><head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Usage d'une ressource regroupant des messages personnalisés</title>
</head>
<body>
<f:view>
      <f:loadBundle basename="premierProjetJSF.messages"
      var="mesMessagesPerso"/>
      <h:form>
            <h:outputText value="#{mesMessagesPerso.clientcode}">
            </h:outputText>
      </h:form>
</f:view>
</body>
</html>
```

Malgré l'intérêt que peut présenter cette forme d'utilisation des messages personnalisés, l'emploi de la balise `<f:loadBundle>` dans chaque page susceptible d'exploiter le fichier de propriétés contenant les messages peut paraître fort contraignant, en particulier lorsque le nombre des pages concernées est important. Pour lever cette contrainte, la référence aux fichiers de propriétés peut se faire d'une autre manière dans le fichier de configuration

© ENI Editions - All rigths reserved

des ressources, comme le montre le code suivant :

```
<application>
     ...
     <ressource-bundle>
          <base-name>
               premierProjetJSF.messages
          </base-name>
          <var>
               mesMessagesPerso
          </var>
     </ressource-bundle>
     ...
</application>
```

L'usage initial de l'élément `<message-bundle>` a été remplacé par celui de la balise `<ressource-bundle>`, à laquelle sont associées les éléments enfants `<base-name>` et `<var>`. Le contenu de ce dernier élément précise l'identifiant du fichier de propriétés, tel qu'il pourra être employé dans les différentes pages web de l'application sans qu'il soit fait usage de la balise `<f:loadBundle>`.

Validateurs personnalisés

L'implémentation JSF fournit un jeu de validateurs par défaut, qui peuvent être utilisés pour répondre aux besoins les plus courants. Cependant, ces validateurs peuvent s'avérer insuffisants dans le cadre d'applications professionnelles où la logique métier peut imposer la mise en place de contrôles plus poussés. JSF offre donc la possibilité de définir des validateurs personnalisés, par la création de classes Java implémentant l'interface `javax.faces.validator.Validator`.

Le caractère non standard de ce type de ressource implique obligatoirement sa déclaration au sein du fichier de configuration des ressources de l'application, grâce à l'élément `<validator>` directement placé sous `<faces-config>` :

```
<validator>
      <display-name>
      Validation du mot de passe</display-name>
      <validator-id>
      monValidateurDeMotDePasse</validator-id>
      <validator-class>
      premierProjetJSF.ValidatorPassword</validator-class>
      <attribute>
            ...
            <attribute-name>longueurMinimale</attributeName>
            <attribute-class>java.lang.Integer</attribute-class>
            <default-value>5</default-value>

            <attribute-name>nombreCaracteresSpeciaux</attributeName>
            <attribute-class>java.lang.Integer</attribute-class>

      </attribute>
</validator>
```

L'exemple ci-dessus présente la déclaration d'un validateur personnalisé nommé `monValidateurDeMotDePasse`. Cet identifiant est celui qui doit être utilisé dans les pages de l'application web pour exploiter le validateur en question. La classe à partir de laquelle est instancié le validateur est mentionnée grâce à l'élément `<validator-class>`. Son nom doit être indiqué intégralement, package compris. Si le validateur requiert des attributs, ceux-ci sont précisés par leur nom et leur type grâce aux éléments `<attribute>`, `<attribute-name>` et `<attribute-class>`.

Pour information, un tel validateur personnalisé peut être constitué pour vérifier que la saisie d'un mot de passe répond à certains critères précis, comme le respect d'un nombre de caractères minimal, ainsi que l'obligation de contenir un certain nombre de caractères spéciaux (%, #, $, etc.).

L'étude des validateurs personnalisés est assurée de manière plus approfondie dans le chapitre Convertisseurs et validateurs de cet ouvrage.

Convertisseurs personnalisés

Pour les mêmes raisons que celles évoquées à propos des validateurs personnalisés, il est indispensable de déclarer les convertisseurs non standards dans le fichier de configuration des ressources, afin qu'ils soient disponibles dans l'ensemble de l'application. Un convertisseur est une classe Java implémentant l'interface `javax.faces.convert.Converter`, dont les références sont précisées à l'aide d'une balise `<converter>`. Le code ci-

© ENI Editions - All rigths reserved

dessous présente la déclaration d'un convertisseur personnalisé.

```
<converter><description>
           Utilisé pour s'assurer qu'un nombre précis peut être
           Stocké dans une propriété de JavaBean ne pouvant
           Accueillir qu'un nombre pair positif.
     </description>
     <display-name>
     convertisseurNombrePairPositif</display-name>
     <converter-id>
     monConvertisseurNombrePairPositif</converter-id>
     <converter-class>
     premierProjetJSF.NombrePairPositifConverter</converter-class>
</converter>
```

Là encore, les propriétés principales du convertisseur personnalisé se limitent à son identifiant, tel qu'il sera exploité au sein des pages de l'application, et la classe à partir de laquelle il est instancié. Ces deux informations sont données respectivement par l'usage des balises `<converter-id>` et `<converter-class>`.

Règles de navigation

Le fichier de configuration des ressources est aussi exploité par l'architecte de l'application pour mentionner le flot de navigation. La navigation au sein d'une application JSF se définit à l'aide de règles de navigation qui spécifient comment il est possible d'aller d'une page web vers une autre lorsque l'internaute active un bouton ou un lien hypertexte.

Concrètement, une règle de navigation concerne une page source en particulier, et indique les différentes situations qui peuvent provoquer la navigation de cette page source vers d'autres pages de l'application. Ces différentes situations, nommées cas de navigation dans la terminologie JSF, sont le plus souvent caractérisées par un identifiant (chaîne de caractères) et le nom de la page cible correspondante.

Voici comment se déclare une règle de navigation simple, dans le fichier de configuration des ressources :

```
<navigation-rule>
      <display-name>
      inscription</display-name>
      <from-view-id>
      /inscription.jsp</from-view-id>
      <navigation-case>
            <from-outcome>succesInscription</from-outcome>
            <to-view-id>
            /confirmationInscription.jsp</to-view-id>
      </navigation-case>
      <navigation-case>
            <from-outcome>echecInscription</from-outcome>
            <to-view-id>
            /inscriptionImpossible.jsp</to-view-id>
      </navigation-case>
</navigation-rule>
```

Une règle de navigation spécifique se définit donc à l'aide d'une balise `<navigation-rule>` directement placée sous l'élément `<faces-config>`. La page source concernée par la règle est indiquée par le contenu de l'élément `<from-view-id>`. Dans le cas où cet élément n'est pas précisé, la règle s'applique à toutes les pages de l'application. Si une chaîne générique est utilisée comme valeur de cet élément (par exemple, **/administration/***), seules les pages qui y correspondent sont concernées par la règle de navigation (ici, toutes les pages situées dans le sous-répertoire **/administration** de l'application).

Les cas de situations répondant à la règle correspondent aux balises `<navigation-case>`, auxquelles sont le plus souvent associées les balises `<from-outcome>` et `<to-view-id>`. La première contient la chaîne de caractères identifiant le cas de navigation, alors que la seconde mentionne la page cible correspondante.

L'exemple présenté ci-dessus correspond à une unique règle de navigation concernant la page source **inscription.jsp**. Cette règle est composée de deux cas de navigation permettant de passer de la page source vers la page cible **confirmationInscription.jsp** lorsque le cas nommé **succesInscription** est employé et de passer vers la page **inscriptionImpossible.jsp** en utilisant le cas **echecInscription**.

D'autres détails concernant la mise en œuvre du modèle de navigation JSF sont présentés plus loin dans ce chapitre.

© ENI Editions - All rigths reserved

Étapes à respecter pour la création d'une application JSF

Le développement d'une application JSF implique le respect des étapes suivantes :

- création d'une instance de la `FacesServlet` ;
- conception des pages web en utilisant les différents composants graphiques ainsi que les éléments de la bibliothèque Core ;
- spécification de la navigation au sein de l'application web ;
- développement des beans nécessaires au bon fonctionnement de l'application.

L'un des intérêts de la technologie Java Server Faces est qu'elle permet une répartition des tâches de développement entre trois types d'intervenants. En l'occurrence :

- Le designer web, qui prend en charge la création des pages JSP/JSF et l'ergonomie générale de l'application. Son travail est largement facilité par l'emploi des bibliothèques de balises JSF.
- Le développeur de l'application, dont le rôle consiste à créer les différents beans. Celui-ci est également amené à mettre en place les éventuels composants personnalisés non graphiques, tels que les convertisseurs, les validateurs et les écouteurs d'événements.
- Le concepteur de composants graphiques personnalisés. Naturellement, sa tâche comprend également la gestion du rendu de ces composants graphiques.

1. Création d'une instance de la FacesServlet

Toutes les applications web basées sur JSF exploitent une instance de la `FacesServlet`. Celle-ci est utilisée pour prendre en charge les requêtes entrantes, les analyser et les insérer dans le cycle de vie de l'application web. Elle assure également l'initialisation des multiples ressources utilisées dans l'application.

Pour qu'une application Java Server Faces soit en mesure d'exécuter une première page JSP, le conteneur web invoque l'instance de `FacesServlet`. Cela a pour effet d'enclencher le cycle de vie de l'application. L'instance en question est identifiée au niveau de l'application par un mapping qui peut être :

- un préfixe : par exemple, /faces/* ;
- une extension : par exemple, *.faces.

Ce mapping est utilisé pour repérer les pages JSP susceptibles de contenir des éléments Java Server Faces.

Le code ci-dessous présente les lignes correspondant à la déclaration d'un mapping de la `FacesServlet`, au sein du descripteur de déploiement d'une application web :

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

Dans cet exemple, toutes les pages JSP situées dans le sous-répertoire **faces** de l'application web sont susceptibles de contenir des composants Java Server Faces. Il s'agit donc d'un mapping par préfixe qui peut, par exemple, être facilement exploité depuis la barre d'adresse d'un navigateur web : http://localhost:8080/monApplicationJSF/faces/index.jsp

L'exemple suivant illustre le mapping de la `FacesServlet` par l'extension ***.faces** :

© ENI Editions - All rigths reserved

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

Lorsque le conteneur web récupère une requête adressée à une page JSP dont le nom se termine par l'extension **faces**, il la transmet à l'instance de `FacesServlet`. Celle-ci recherche alors dans l'arborescence de l'application web la page JSP correspondant au nom mentionné, en remplaçant l'extension faces par l'extension jsp. Par exemple, si l'utilisateur demande la ressource http://localhost:8080/monApplicationJSF/index.faces, c'est la page index.jsp qui sera recherchée puis prise en charge par l'instance de `FacesServlet`.

2. Création des pages web

Cette étape consiste à réaliser les opérations suivantes :

- Positionner les composants Java Server Faces sur la page.
- Lier les composants à des propriétés de beans, lorsque c'est nécessaire.
- Référencer les éventuels convertisseurs, validateurs et écouteurs auprès des composants graphiques qui requièrent leur existence.

a. Déclaration des bibliothèques de balises Java Server Faces

Ces actions, théoriquement à la charge du designer web, imposent la déclaration des deux bibliothèques standards de balises JSF en en-tête de chaque page web. En effet, l'exploitation de composants graphiques Java Server Faces dans les pages JSP ne peut se faire qu'à condition de savoir comment restituer à l'écran l'apparence de ces composants : la bibliothèque HTML fournit cette information. Il est également impératif de savoir comment gérer les actions réalisées au niveau de chaque composant graphique : c'est le rôle de la bibliothèque Core. Il faut donc systématiquement mentionner l'accès à ces deux bibliothèques dans les pages JSP, grâce aux directives `taglib` suivantes, dès lors que les pages en question font usage des balises JSF.

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

L'attribut `uri` de chacune de ces directives référence la TLD décrivant les balises contenues dans la bibliothèque concernée. L'attribut `prefix` identifie le préfixe utilisé dans la page JSP pour faire référence à l'une ou l'autre des bibliothèques. Le nom de ce préfixe peut être choisi librement.

b. Positionnement des balises view et form

Tous les composants JSF d'une page web sont regroupés dans un arbre de composants. Dans la nomenclature Java Server Faces, cet arbre est nommé vue. La racine de cet arbre est un composant JSF issu de la classe `UIViewRoot`, dont la représentation au sein des pages web se fait grâce à une balise `view`. Cette balise est déclarée dans la bibliothèque Core. Au moment du développement des pages de l'application, la balise `view` doit impérativement être positionnée en tant qu'élément parent de toute autre balise JSF, et en particulier de celles correspondant aux composants graphiques.

Par ailleurs, si les pages web présentent des composants permettant aux utilisateurs de réaliser des saisies (zones de texte, zones de liste déroulantes, cases à cocher, boutons d'options…), il est impératif d'intercaler une balise `form` entre la balise `view` et les différentes balises représentant les composants de saisie. Cette balise `form` est issue de la bibliothèque HTML et peut comporter un attribut `id` dont le rôle consiste à identifier de manière unique le formulaire au sein de la page web.

Voici le contenu type d'une page JSP exploitant JSF, avant insertion de tout composant autre que `view` et `form`.

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<html>
<head><title>Formulaire d'inscription</title></head>
<body><f:view>
          <h:form id="formulaireInscription"></h:form>
     </f:view>
</body>
```


© ENI Editions - All rigths reserved

```
</html>
```

c. Ajout de composants graphiques dans une page web

Les éléments principaux des pages web sont, bien entendu, les composants graphiques à partir desquels l'internaute est amené à interagir avec l'application web. Ces composants, décrits plus en détail dans la section Les composants graphiques de ce chapitre, sont des objets issus de classes dérivées de la classe parente `javax.faces.component.UIComponentBase`. À chacun d'eux est associée une balise spécifique de la bibliothèque HTML. L'ajout de composants graphiques JSF à une page web consiste donc simplement à positionner dans le code source de cette page les balises correspondant aux éléments que l'on souhaite visualiser. Il faut toutefois s'assurer que ces balises sont bien insérées entre les éléments `<f:view>` et `</f:view>`, et que les attributs impérativement requis sont bien renseignés.

Dans le formulaire de saisie utilisé pour illustrer ce chapitre, plusieurs composants graphiques sont utilisés :

- Les champs texte de sortie, qui servent à afficher des informations textuelles sur l'interface web. Ce type de composant graphique est utilisé ici pour présenter l'invite située en en-tête de formulaire, ainsi que les différents labels des champs de saisie. La balise à utiliser pour les champs texte de sortie est `<h:outputText>`.

- Les champs de saisie de type texte : la balise qui leur correspond est `<h:inputText>`.

- Les zones de liste déroulantes à choix unique, représentées par la balise `<h:selectOneListBox>`. Cette balise dispose de balises enfants `<f:selectItem>`, qui correspondent aux éléments de listes.

- Les boutons radio, générés grâce à une balise `<h:selectOneRadio>`. À cette balise sont associées une ou plusieurs balises enfants `<f:selectItem>` pour représenter chacun des choix possibles.

- Les boutons de commande. La balise qui leur correspond est `<f:commandButton>`.

- Les messages qui apparaissent lorsque la validation du formulaire échoue, du fait d'une contrainte non respectée (nombre de caractères insuffisant, mauvais format de chaîne, etc.). Ces messages sont insérés dans la page web sous la forme d'une balise `<h:message>`.

Le code source de la page JSP représentant le formulaire à concevoir est situé ci-dessous. Tous les composants JSF qui viennent d'être cités y sont mentionnés : ils sont accompagnés d'attributs obligatoires ou optionnels, dont la signification sera expliquée plus loin dans ce chapitre. Notez que tous ces composants graphiques JSF sont intercalés avec du code HTML classique, pour permettre notamment un affichage sous la forme de tableau.

```
<%@taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
<%@taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8859-1"%>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Formulaire d'inscription</title>
</head>
<body>

<f:view>
<h:form id="formulaireInscription">

<table align="center">
<tr>
<td colspan="3">
<h:outputText value="Saisissez les paramètres suivants pour valider
votre inscription:" style="color: #0000FF; text-decoration:
underline">
</h:outputText>
<br><br>
</td>
</tr>

<tr>
```

© ENI Editions - All rigths reserved

```
<td>
<h:outputText id="otxtNom" value="Nom*">
</h:outputText>
</td>
<td>
<h:inputText id="itxtNom" >
</h:inputText>
</td>
<td>
<h:message id="msgNom" for="itxtNom" style="color: #FF0000;
background-color: #FFFF00">
</h:message>
</td>
</tr>

<tr>
<td>
<h:outputText id="otxtPrenom" value="Prénom">
</h:outputText>
</td>
<td>
<h:inputText id="itxtPrenom" value="Prénom par défaut">
</h:inputText>
</td>
<td>
<h:message id="msgPrenom" for="itxtPrenom" style="color: #FF0000;
background-color: #FFFF00">
</h:message>
</td>
</tr>

<tr>
<td>
<h:outputText id="otxtDateNaissance" value="Date de naissance*">
</h:outputText>
</td>
<td>
<h:inputText id="itxtDateNaissance" >
</h:inputText>
</td>
<td>
<h:message id="msgDateNaissance" for="itxtDateNaissance"
style="color: #FF0000; background-color: #FFFF00">
</h:message>
</td>
</tr>
<tr>
<td>
<h:outputText id="otxtSexe" value="Sexe*">
</h:outputText>
</td>
<td>
<h:selectOneRadio id="radioSx">
<f:selectItem id="sexeChoix1" itemLabel="Masculin" itemValue="Masc" />
<f:selectItem id="sexeChoix2" itemLabel="Féminin" itemValue="Fem" />
</h:selectOneRadio>
</td>
<td>
<h:message id="msgSexe" for="radioSx" style="color: #FF0000;
background-color: #FFFF00">
</h:message>
</td>
</tr>
<tr>
<td>
<h:outputText id="otxtNombreEnfants" value="Nombre d’enfants">
</h:outputText>
</td>
<td>
```


© ENI Editions - All rigths reserved

```
<h:selectOneListbox id="listeAge" size="1">
<f:selectItem id="itemZero" itemLabel="0" itemValue="0" />
<f:selectItem id="itemUn" itemLabel="1" itemValue="1" />
<f:selectItem id="itemDeux" itemLabel="2" itemValue="2" />
<f:selectItem id="itemTrois" itemLabel="3" itemValue="3" />
<f:selectItem id="itemQuatre" itemLabel="4" itemValue="4" />
<f:selectItem id="itemCinqPlus" itemLabel="5 et plus" itemValue="5" />
</h:selectOneListbox>
</td>
<td></td>
</tr>

<tr>
<td colspan="3" align="center">
<h:commandButton id="cmdValider" value="Valider votre inscription">
</h:commandButton>
</td>
</tr>

</table>

</h:form>
</f:view>

</body>
</html>
```

© ENI Editions - All rigths reserved

Les composants graphiques

Ce paragraphe décrit les différents composants d'interfaces proposés en standard par la technologie Java Server Faces.

Les composants graphiques sont constitués côté serveur par des classes spécifiques, dérivées de la classe parente `javax.faces.component.UIComponentBase`. Ces composants peuvent être associés à diverses interfaces permettant notamment :

- le maintien de l'état entre les pages web ;
- la prise en charge de la gestion événementielle ;
- la restitution de rendus visuels spécifiques, variables suivant le composant graphique concerné.

1. Classes correspondant aux composants graphiques

Comme toutes les classes Java, la classe `UIComponentBase` a la possibilité d'être dérivée dans le but de permettre la constitution de composants graphiques entièrement personnalisés. Ce point fait l'objet de l'intégralité du chapitre Composants graphiques personnalisés de cet ouvrage.

La liste suivante présente les principales classes correspondant à des composants graphiques standards JSF. Le rôle de ces composants y est précisé, ainsi que celui de leurs principaux attributs, modifiables par appel de méthode (ou par précision d'un attribut spécifique dans la balise représentant le composant). Les classes sont présentées ici selon l'ordre alphabétique.

`UIColumn`

Représente une colonne unique du composant `UIData`.

Attributs caractéristiques éventuels :

contenus de l'en-tête et du pied de colonne (méthodes `setHeader(…)` et `setFooter(…)`).

`UICommand`

Représente un composant pouvant être utilisé au niveau de l'interface web pour provoquer une action particulière. En général, ce type de composant est restitué visuellement par un bouton, un élément de menu ou un lien hypertexte.

La méthode `decode()` de cette classe assure la détection de l'activation du contrôle (clic de souris, par exemple) et place un événement de type `ActionEvent` en file d'attente de traitement.

La méthode `broadcast()` assure la diffusion de cet événement à tous les écouteurs concernés.

La méthode `getActionExpression()` renvoie l'expression de méthode (c'est-à-dire l'expression pointant une méthode particulière d'un objet spécifique) permettant d'entreprendre l'action à réaliser lorsque l'internaute active le composant. Suivant la valeur de la propriété `immediate` (voir ci-après), l'exécution de cette action intervient soit durant la phase *Apply Request Values*, soit durant la phase *Invoke Application*).

Attributs caractéristiques éventuels :

Liste des écouteurs d'événements associés à cette source.

Flag indiquant si le traitement des événements doit être traité immédiatement ou non (méthodes `isImmediate()` et `setImmediate(…)`).

`UIData`

Ce composant est conçu pour être lié à une collection de données et en afficher le contenu suivant un rendu particulier.

Le composant `UIData` le plus courant est le tableau HTML, instancié sous la forme d'un objet de type `HtmlDataTable` (classe dérivée de `UIData`).

© ENI Editions - All rigths reserved

Attributs caractéristiques éventuels :

Contenus de l'en-tête et du pied de tableau (méthodes `setHeader(…)` et `setFooter(…)`.

Nombre de lignes (`getRowCount()`), donnée située au rang courant (`getRowData()`).

`UIForm`

Représente un formulaire destiné à être présenté à l'internaute. Ce composant dispose d'un ensemble de composants "enfants", correspondant aux différents champs de saisie utilisés dans le formulaire.

Attributs caractéristiques éventuels :

Famille à laquelle le composant est rattaché (`getFamily()`).

`UIGraphic`

Il s'agit d'un composant chargé de présenter une image à l'utilisateur. Cette image ne peut en aucun cas être manipulée.

Attributs caractéristiques éventuels :

L'URL de l'image à présenter.

`UIInput`

Cette classe est dérivée de la classe `UIOutput` (voir ci-dessous).

Elle représente un composant graphique capable à la fois d'éditer du contenu (cette capacité provient de l'héritage de `UIOutput`) et de recevoir des informations en saisie.

L'information stockée dans un champ `UIInput` est prise en charge lors de la validation du formulaire qui le contient. Durant la phase *Apply request values* du cycle de vie, la valeur du composant (habituellement une chaîne de caractère) doit être stockée au moyen de l'appel de la méthode `setSubmittedValue()`. À ce niveau, la valeur du composant n'a pas encore été prise en charge par un éventuel convertisseur (voir la section Conversions de données ci-dessous). Lorsque le composant `UIInput` ne dispose d'aucune valeur au moment de la soumission du formulaire, il a la capacité de ne rien faire, ou de soumettre la valeur `null`.

Pendant la phase *Process Validators* du cycle de vie, une tentative de conversion de la valeur soumise est réalisée, dans le but de pouvoir sauvegarder cette valeur au niveau de la couche métier de l'application web. En cas de succès, la valeur en question est de surcroît sauvegardée au niveau du composant graphique lui-même, par appel de la méthode `setValue()`.

Lorsque la méthode `validate()` de `UIInput` détecte un changement de valeur, et que cette nouvelle valeur subit avec succès les processus de validation et de conversion, un événement `javax.faces.event.ValueChangeEvent` est placé en file d'attente. Cet événement est par la suite propagé à l'ensemble des écouteurs concernés, par appel de la méthode `broadcast()`. Cette propagation intervient en général durant la phase *Process Validators*, mais elle peut toutefois intervenir pendant la phase *Apply Request Values* si la propriété immediate du composant `UIInput` est positionnée sur la valeur `true`.

Attributs caractéristiques éventuels :

Liste des validateurs, modifiables par appel des méthodes `addValidator(…)` et `removeValidator(…)`.

Flag indiquant si la précision d'une valeur pour ce champ est obligatoire (méthode `setRequired(…)`).

Message renvoyé à l'internaute en cas de non précision d'une valeur, dans le cas où celle-ci est requise (méthodes `getRequiredMessage()` et `setRequiredMessage(…)`).

Message renvoyé à l'internaute en cas d'échec du processus de conversion de la valeur du champ (méthodes `getConverterMessage()` et `setConverterMessage(…)`).

Message renvoyé à l'internaute en cas d'échec du processus de validation de la valeur du champ (méthodes

© ENI Editions - All rigths reserved

`getValidatorMessage()` et `setValidatorMessage(…)`).

`UIMessage`

Ce composant assure l'affichage de messages concernant un autre composant graphique (le client) situé sur l'interface web. Le contenu du message à afficher s'obtient à partir d'un FacesMessage généré, par exemple, par un validateur ou un convertisseur.

Attributs caractéristiques éventuels :

L'identifiant du client, c'est-à-dire du composant concerné par le message (méthode `setFor(…)`).

`UIMessages`

Permet l'affichage d'un ensemble de messages dans l'interface web. Un composant `UIMessages` est en mesure de présenter à l'utilisateur les éventuels messages relatifs à l'ensemble des autres composants disposés sur la page web.

Famille à laquelle le composant est rattaché (`getFamily()`).

`UIOutput`

Ce composant est conçu pour permettre l'affichage de sa valeur dans l'interface utilisateur. Cette valeur, qui peut notamment être obtenue dynamiquement depuis un JavaBean ou un bean managé, n'est pas modifiable par l'internaute.

Pendant la phase *Render response* du cycle de vie, la valeur du composant doit être convertie en chaîne de caractères pour pouvoir être présentée à l'écran. Cette conversion peut être assurée de deux manières :

- Dans le cas où l'objet `UIOutput` dispose d'un convertisseur défini (objet issu d'une classe implémentant l'interface `javax.faces.convert.Converter`), la méthode `getAsString()` de celui-ci est invoquée.

- Lorsqu'aucun convertisseur spécifique n'est associé au `UIOutput`, la conversion est assurée par l'appel de la méthode `toString()` du composant.

Si la conversion échoue, une `ConverterException` est générée.

Attributs caractéristiques éventuels :

Convertisseur éventuellement associé au composant (méthode `getConverter()`).

`UIPanel`

Composant jouant le rôle de conteneur d'autres composants graphiques. Il est chargé de contrôler la disposition visuelle de ces composants.

Attributs caractéristiques éventuels :

Famille à laquelle le composant est rattaché (`getFamily()`).

`UISelectBoolean`

Cette classe dérive de la classe `UIInput`.

Elle correspond à un champ permettant de saisir une valeur booléenne. Le plus souvent, ce champ est représenté par une case à cocher.

Attributs caractéristiques éventuels :

Flag indiquant si le champ est sélectionné (méthodes `setSelected(…)` et `isSelected()`).

`UISelectItem`

Peut être positionné au sein d'un composant graphique conteneur de type `UISelectMany` ou `UISelectOne`. L'utilisation

© ENI Editions - All rigths reserved

de ce composant provoque la création d'une instance de la classe `javax.faces.model.SelectItem`, qui vient s'ajouter à l'ensemble des autres éléments sélectionnables déjà rattachés au composant conteneur.

Attributs caractéristiques éventuels :

La description de l'élément (méthode `getItemDescription()`), son label (`getItemLabel()`) et sa valeur (`getItemValue()`).

Flag indiquant si l'élément de liste est sélectionnable ou non (méthode `isItemDisabled()`).

`UISelectItems`

Ce composant peut être positionné au sein d'un composant graphique conteneur de type `UISelectMany` ou `UISelectOne`. L'utilisation de ce composant provoque la création d'une ou de plusieurs instances issues de la classe `SelectItem`. Ces instances viennent s'ajouter à la liste des autres éléments sélectionnables déjà rattachés au composant conteneur.

La propriété `value` d'un `UISelectItems` peut contenir :

- Une unique instance de la classe `SelectItem` ;
- Un tableau d'instances de la classe `SelectItem`. Les instances en question sont alors placées dans l'ordre numérique croissant ou dans l'ordre alphabétique.
- Une collection d'instances de la classe `SelectItem` ;
- Une carte (instance de la classe `Map`). Chaque élément de la carte est utilisé pour créer une instance de la classe `SelectItem`. La clé de l'élément de carte est identifiée au label du `SelectItem`, alors que sa valeur est associée à la valeur du `SelectItem`.

Attributs caractéristiques éventuels :

Famille à laquelle le composant est rattaché (`getFamily()`).

`UISelectMany`

Cette classe dérive de la classe `UIInput`.

Ce composant permet à l'internaute de réaliser une sélection multiple parmi un ensemble de valeurs prédéfinies. Ce composant est généralement restitué visuellement selon l'une des deux formes suivantes :

- Un groupe de cases à cocher (instance la classe `HtmlSelectManyCheckbox`) ;
- Une zone de liste à choix multiples (instance la classe `HtmlSelectManyListbox`).

Un composant `UISelectMany` peut être configuré pour que certains des éléments qu'il contient soient présélectionnés par défaut.

Attributs caractéristiques éventuels :

Liste des éléments sélectionnés (méthode `getSelectedValues()`).

`UISelectOne`

Cette classe dérive de la classe `UIInput`.

Ce composant est utilisé pour permettre la sélection d'un unique élément parmi un ensemble de valeurs prédéfinies. Ce composant est généralement restitué visuellement sous la forme d'une zone de liste déroulante à choix unique, ou sous la forme d'un groupe de boutons radio.

Un composant `UISelectOne` peut être configuré pour qu'un élément particulier soit présélectionné par défaut.

© ENI Editions - All rigths reserved

La classe `UISelectOne` est dérivée de `UIInput`.

Attributs caractéristiques éventuels :

Famille à laquelle le composant est rattaché (`getFamily()`).

`UIViewRoot`

Représente la racine de l'arbre de composants.

2. Interfaces mises en œuvre par les composants graphiques

En complément du fait que les classes représentant les composants graphiques héritent de la classe `UIComponentBase`, ces classes implémentent une ou plusieurs interfaces chargées de définir le comportement des composants graphiques vis-à-vis de certains événements. Ces interfaces de comportement, toutes issues du package `javax.faces.component`, sont les suivantes :

- **ActionSource** : cette interface peut être implémentée par les composants susceptibles d'être la source d'événements de type `javax.faces.event.ActionEvent`. Les classes standards implémentant cette interface sont `HtmlCommandButton`, `HtmlCommandLink` et `UICommand`.

- **ActionSource2** : cette interface étend l'interface `ActionSource` et fournit donc des fonctionnalités identiques. Elle permet cependant aux composants qui l'exploitent d'utiliser le langage d'expressions pour référencer les méthodes chargées de capturer les événements.

- **EditableValueHolder** : cette interface étend l'interface `ValueHolder` (voir ci-dessous). Elle propose davantage de fonctionnalités comme la prise en charge de l'événement `javax.faces.event.ValueChangeEvent` et le support de la validation des données (voir la section Validation de données de ce chapitre).

- **StateHolder** : cette interface permet au composant qui l'implémente de conserver son état entre les différentes requêtes.

- **ValueHolder** : le composant exploitant cette interface maintient une valeur locale, et supporte la conversion entre le type `String` et le type de la donnée associée, située dans la couche métier de l'application. Pour plus d'informations sur la notion de conversion de données, voir la section Conversions de données de ce chapitre.

La liste suivante récapitule le nom des interfaces de comportement, et précise pour chacune d'elles le(s) composant (s) graphique(s) qui l'implémente(nt). Naturellement, les classes dérivées des classes de composants mentionnées exploitent également les interfaces correspondantes.

Nom de l'interface	Classe(s) de composant(s) implémentant l'interface
ActionSource, ActionSource2	UICommand
EditableValueHolder	UIInput, UISelectBoolean, UISelectMany, UISelectOne
NamingContainer	UIData, UIForm
StateHolder	UIColumn, UICommand, UIData, UIForm, UIGraphic, UIInput, UIMessage, UIMessages, UIOutput, UIPanel, UISelectBoolean, UISelectItem, UISelectItems, UISelectMany, UISelectOne, UIViewRoot
ValueHolder	UIInput, UIOutput, UISelectBoolean, UISelectMany, UISelectOne

Sur un plan purement organisationnel pour la conception de l'application web, seuls les concepteurs de composants graphiques sont amenés à connaître/manipuler les différentes classes de composants et leurs interfaces. En l'occurrence, il n'est pas nécessaire que ces détails techniques soient connus du designer web, puisque celui-ci peut exploiter les différents composants graphiques en utilisant les balises des bibliothèques Core et HTML qui leur correspondent.

© ENI Editions - All rigths reserved

La gestion du rendu : notion de renderer

La plupart des composants graphiques JSF peuvent être restitués graphiquement de différentes manières, dans le but principal d'assurer des affichages corrects quel que soit le type de terminal (ordinateur, téléphone mobile, etc.) accédant à une page donnée de l'application. Pour un même terminal cependant, la gestion du rendu peut être exploitée pour obtenir si nécessaire plusieurs représentations graphiques du même composant. Par exemple, les composants de type `UICommand` peuvent être proposés à l'internaute sous la forme d'un simple bouton ou d'un lien hypertexte.

Ce paragraphe traite du fonctionnement de la gestion du rendu des composants graphiques. Il montre également comment les designers web peuvent mettre en œuvre l'un ou l'autre des modes de restitution par l'utilisation de balises adaptées.

1. Intérêt de la gestion du rendu

L'architecture de composants proposée par la technologie Java Server Faces est constituée de telle sorte que le comportement et l'apparence des composants ne sont pas gérés ensemble. Comme cela a été présenté précédemment, le comportement des composants graphiques est décrit au sein de classes dérivées de la classe `UIComponentBase`. L'apparence des composants, quant à elle, est prise en charge par la gestion du rendu. Cette séparation présente plusieurs avantages, notamment :

- Les développeurs de composants graphiques peuvent définir le comportement des composants qu'ils conçoivent, et proposer pour ceux-ci différentes façons de les représenter. Il est alors possible de disposer de composants dont l'apparence peut être adaptée au client qui les affiche (un navigateur web classique sur ordinateur, un navigateur adapté aux téléphones mobiles, par exemple).

- Les designers web peuvent rapidement changer l'apparence du composant sur une page donnée, en utilisant une balise qui associe de manière appropriée un composant particulier et un rendu spécifique.

2. Kit de rendu - Classes de rendu

Un kit de rendu définit comment les classes de composants doivent être associées aux balises de composants, suivant le client qui les exploite. La technologie JSF inclut en standard un kit de rendu assurant la restitution des composants graphiques pour les clients HTML.

Un kit de rendu comprend un ensemble de classes de rendu, pour chaque type de composant supporté. Chaque classe de rendu précise la manière dont doit être restitué un composant graphique spécifique, compte tenu du type de client mentionné dans le kit de rendu. À titre d'illustration, un composant de type `UISelectOne` peut être restitué de trois façons : sous la forme d'un jeu de boutons radio, sous la forme d'une zone de liste déroulante (combo box) ou sous la forme d'une zone de liste déroulante avec ascenseur.

Chaque balise JSP définie dans le kit de rendu standard HTML comprend à la fois les fonctionnalités du composant concerné (définies dans la classe dérivée de `UIComponentBase` qui lui correspond), et les attributs de rendu, précisés dans la classe de rendu.

3. Bibliothèque de balises pour le rendu des composants

L'implémentation JSF fournit en standard une bibliothèque de balises destinées à la restitution des composants graphiques pour les clients HTML. Il s'agit de la bibliothèque HTML déjà traitée au chapitre Composants graphiques personnalisés, dont les points essentiels sont rappelés ici :

Nom de balise	Rôle	Restituée comme...
`column`	Représente une colonne de données dans un composant de type `UIData`.	Une colonne d'un tableau HTML.
`commandButton`	Sert à soumettre les données d'un formulaire web.	Une balise `<input>` dont l'attribut `type` peut prendre les valeurs suivantes : `submit`, `reset` ou `image`. Présentée sous la forme d'un

© ENI Editions - All rigths reserved

		bouton.
commandLink	Constitue un lien vers une ressource interne ou externe à l'application web. Son utilisation provoque la génération d'un javax.faces.event.ActionEvent.	Une balise <a href>.
dataTable	Permet de représenter un tableau de données.	Une balise <table>. Il s'agit d'un tableau HTML dont le contenu peut évoluer dynamiquement.
form	Représente un formulaire de saisie dont tous les éléments permettent à l'utilisateur de soumettre des informations à l'application web.	Une balise <form>.
graphicImage	Affiche une image dans le navigateur.	Une balise <img>.
inputHidden	Permet d'introduire un champ caché au sein d'un formulaire web.	Une balise <input> de type hidden.
inputSecret	Permet de placer un champ de mot de passe dans un formulaire. Les caractères saisis ne sont pas visualisables.	Une balise <input> de type password.
inputText	Représente une zone de saisie de type texte.	Une balise <input> de type text.
inputTextArea	Permet à l'internaute de saisir une chaîne de caractères multilignes.	Une balise <textarea>.
message	Affiche un message dans le navigateur.	Une balise dont l'exploitation est effective lorsque les styles CSS sont utilisés. Le message en tant que tel est représenté par une chaîne de caractères.
messages	Affiche plusieurs messages textuels dans le navigateur.	Un ensemble de balises dont l'exploitation est effective lorsque les styles CSS sont utilisés. Les messages en tant que tels sont représentés par des chaînes de caractères.
outputFormat	Affiche un message dans le navigateur.	Une simple chaîne de caractères.
outputLabel	Représente un label affecté à un champ de saisie spécifique.	Une balise <label>.
outputLink	Constitue un lien vers une ressource interne ou externe à l'application web. À la différence de la balise commandLink, son utilisation ne provoque pas la génération d'un javax.faces.event.ActionEvent.	Une balise <a href>.
outputText	Affiche une ligne de texte.	Une simple chaîne de caractères.

© ENI Editions - All rigths reserved

panelGrid	Affiche un tableau dans le navigateur.	L'association des balises <table>, <tr> et <td>.
panelGroup	Sert à regrouper au sein d'un conteneur un ensemble d'autres composants.	Une ligne de tableau.
selectBooleanCheckbox	Permet à l'utilisateur de sélectionner une valeur booléenne.	Une balise <input> de type checkbox.
selectItem	Représente un élément particulier d'un ensemble d'éléments d'une zone de liste (instance de la classe UISelectOne).	Une balise <option>.
selectItems	Représente un ensemble d'éléments appartenant à une zone de liste (instance de la classe UISelectOne).	Un ensemble de balises <option>.
selectManyCheckbox	Présente à l'utilisateur un ensemble de cases à cocher.	Un ensemble de balises <input> de type checkbox.
selectManyListBox	Permet à l'utilisateur de faire une sélection multiple parmi un ensemble d'éléments.	Une balise <select> disposant d'un attribut multiple.
selectOneListbox	Permet à l'utilisateur de faire une sélection unique parmi un ensemble d'éléments.	Une balise <select>.
selectOneRadio	Permet à l'utilisateur de faire une sélection unique parmi un ensemble d'éléments.	Une balise <input> de type radio.

Le chapitre Exploitation des composants standards JSF traite de façon plus approfondie la restitution des composants graphiques au sein des pages web. Vous pourrez notamment y trouver toutes les informations nécessaires en vue de l'intégration de composants JSF dans les pages.

© ENI Editions - All rigths reserved

Validation de données

Dans la technologie Java Server Faces, la validation des données saisies dans un formulaire est assurée par des objets spécifiques, dénommés validateurs. Ces objets sont issus de classes implémentant l'interface `javax.faces.validator.Validator`.

1. Principe de fonctionnement de la validation

Pour exercer son rôle, un validateur doit obligatoirement être associé à un composant graphique JSF. Au moment de la soumission du formulaire, plus précisément dans la phase *Process Validators* du cycle de vie, la méthode `validate(…)` du validateur est invoquée, dans le but de valider la valeur mentionnée dans le composant graphique associé.

```
void validate(FacesContext context, UIComponent component,
Object value) throws ValidatorException
```

La signature de cette méthode présente trois paramètres :

- `context`, correspondant au contexte de l'application ;
- `component`, identifiant le composant associé au validateur ;
- `value`, représentant la valeur à contrôler.

Lorsque la validation échoue, la méthode `validate(…)` génère une `ValidatorException`, à laquelle est associé un message explicite, fourni sous la forme d'un objet de type `javax.faces.application.FacesMessage`. La génération de cette exception a pour effet d'interrompre le processus de validation du formulaire : le message rattaché à l'exception peut alors éventuellement être présenté dans la page web, si celle-ci contient une balise `<h:messages>` ou une balise `<h:message>` associé au composant graphique contrôlé.

2. Différents types de validateurs

Il existe plusieurs validateurs standards : ils permettent par exemple de contrôler la longueur d'une chaîne (`LengthValidator`), de vérifier qu'une valeur numérique est bien située dans un intervalle précis (`DoubleRangeValidator` ou `LongRangeValidator`). Les principaux validateurs standards sont présentés dans le chapitre Exploitation des composants standards JSF.

Il est également possible de créer ses propres validateurs, en créant une classe implémentant l'interface `Validator`. Ce point est traité au chapitre Convertisseurs et validateurs.

3. Exemple d'utilisation d'un validateur

Dans le cadre de l'application illustrant ce chapitre, il est nécessaire d'associer un validateur au champ permettant de saisir le nom de la personne à inscrire. Il faut en effet, d'après les contraintes énoncées, que ce nom ait un nombre de caractères compris entre 5 et 100. Le validateur standard apte à réaliser ce type de contrôle est un `LengthValidator`.

Pour cela, le designer de la page web doit positionner une balise spécifique, issue de la bibliothèque Core, correspondant à ce composant JSF. Il s'agit de `<f:validateLength>`, qui accepte deux attributs nommés `minimum` et `maximum`, bien évidemment chargés de représenter les nombres de caractères minimaux et maximaux.

Pour associer un validateur à un composant graphique particulier, il faut placer la balise qui lui correspond entre les balises ouvrantes et fermantes du composant. Ainsi, pour associer un `LengthValidator` au `UIInput` représentant le champ de saisie `Nom`, la balise `<f:validateLength>` doit être insérée entre les balises `<h:inputText>` et `</h:inputText>`.

Les modifications apportées au code source de la page sont représentées ci-dessous :

```
<h:inputText id="itxtNom" required="true" >
     <f:validateLength minimum="5" maximum="100"></f:validateLength>
</h:inputText>
```

© ENI Editions - All rigths reserved

Notez au passage l'ajout d'un attribut `required` à la balise `<h:inputText>`. Positionner celui-ci sur la valeur `true` revient à rendre impérative la saisie d'au moins un caractère dans le champ `UIInput`. La présence de l'attribut en question est ici nécessaire, dans la mesure où le `LengthValidator` ne prend pas en charge l'absence de saisie, contrairement à ce que l'on pourrait croire.

© ENI Editions - All rigths reserved

Conversions de données

1. Pourquoi parler de conversion de données?

La technologie JSF est conçue pour permettre l'association entre un composant graphique présent sur l'interface web, et une propriété d'un objet situé côté serveur (un JavaBean ou un bean managé). La section Liaison de composants à des beans managés du chapitre Exploitation des composants standards JSF présente une mise en application de cet artifice technologique.

Lorsque ce genre d'association est mis en œuvre dans un processus de saisie (par exemple, si un champ `UIInput` est associé à une propriété de bean managé), il est impératif qu'il y ait adéquation entre le type de la valeur saisie dans le composant, et celui de la propriété du bean. De même, lorsqu'une telle association est utilisée dans un processus d'affichage (par exemple, un champ `UIOutput` est lié à une propriété de bean), il faut que la valeur de la propriété soit affichable ; en d'autres termes, il faut que le type de la propriété du bean managé soit compatible ou convertible en `String`.

Pour respecter ces contraintes, la notion de conversion de données est mise en application. On dit qu'une conversion est réalisée entre la vue **Modèle** du composant JSF, caractérisée par le type de la propriété du bean managé, et la vue **Présentation** de ce même composant, visible sous la forme d'une chaîne de caractères dans l'interface web.

Il existe une autre situation dans laquelle les conversions de données peuvent être mises en œuvre. Dans ce cas particulier, il n'y a pas nécessairement de liaison entre un composant graphique et une propriété de JavaBean, et l'usage de la conversion se limite au contrôle du format de la valeur saisie par l'utilisateur dans le champ de saisie concerné. L'exemple proposé dans le cadre de ce chapitre illustre ce type d'application de la conversion de données (le format de la date de naissance doit impérativement être jj/mm/aaaa).

2. Conversions implicites et spécifiques

Dans la plupart des situations, l'implémentation Java Server Faces assure une conversion automatique qui permet de s'assurer de la compatibilité des types de données. C'est le cas par exemple lorsque qu'une case à cocher (composant JSF de type `UISelectBoolean`) est liée à une propriété de bean booléenne : une conversion s'effectue de manière transparente entre le type `String` et le type `boolean`.

Mais il est possible que la conversion ne soit pas réalisable automatiquement par JSF : cela se produit lorsqu'un composant graphique est lié à une propriété dont le type n'est pas standard, ou lorsqu'il est nécessaire de contrôler/modifier le format de la valeur saisie. Comme la mise en œuvre de la conversion est impérative, le concept de convertisseur est introduit par la technologie JSF. Il permet de réaliser une conversion spécifique de la valeur du composant graphique auquel il est associé.

3. Convertisseurs standards et convertisseurs personnalisés

Un convertisseur est un objet issu d'une classe implémentant l'interface `javax.faces.convert.Converter`. Cette interface présente les deux méthodes suivantes :

```
Object getAsObject(FacesContext context,
                   UIComponent component,
               String value)

String getAsString(FacesContext context,
                   UIComponent component,
                   Object value)
```

La méthode `getAsObjet(…)` entre en jeu dans un processus de saisie : elle sert à convertir la valeur de type `String` saisie dans un composant graphique sous la forme d'un `Object`, compatible avec la propriété du bean managé lié au composant. La valeur convertie est stockée dans la propriété en question durant la phase *Apply request values* du cycle de vie.

La méthode `getAsString(…)` effectue l'opération inverse, permettant ainsi l'affichage dans l'interface web de la valeur stockée dans la propriété d'un JavaBean ou d'un bean managé. L'exécution de cette méthode intervient durant la phase *Render Response* du cycle de vie.

Lorsque la conversion échoue, ces méthodes déclenchent une `javax.faces.convert.ConverterException`, provoquant ainsi l'interruption du cycle de vie. Le motif de l'échec de la conversion peut alors être placé dans un `javax.faces.application.FacesMessage` en vue d'être présenté à l'internaute.

© ENI Editions - All rigths reserved

Java Server Faces propose bien sûr de manière standard un ensemble de convertisseurs directement utilisables. Ils sont répartis en deux catégories principales :

- Les convertisseurs numériques, adaptés aux situations faisant notamment intervenir des `Integer`, des `Float`, des `Long`, etc.
- Les convertisseurs de dates, grâce auxquels il est possible de prendre en charge la conversion d'objets de type `String` en objets de type `Date`, selon un format particulier (par exemple jj/mm/aaaa).

Il est par ailleurs possible de créer des convertisseurs personnalisés, en développant des classes implémentant l'interface `Converter`. La personnalisation est alors assurée par la surcharge du comportement par défaut des méthodes `getAsString(…)` et `getAsObject(…)`.

Les convertisseurs standards sont étudiés plus en détail dans le chapitre Exploitation des composants standards JSF, alors que le développement de convertisseurs personnalisés est traité au chapitre Convertisseurs et validateurs.

4. Exemple d'utilisation d'un convertisseur

L'exemple proposé dans le cadre de ce chapitre fait intervenir un convertisseur au niveau du champ de saisie de la date de naissance. Il faut en effet imposer à l'internaute le respect du format jj/mm/aaaa. Comme précisé précédemment, il ne s'agit pas ici d'assurer une conversion entrant dans le cadre d'une liaison entre le champ de saisie "Date de naissance" et un éventuel bean managé : l'usage de la conversion se limite ici au contrôle du format de la chaîne saisie.

Le convertisseur le plus approprié pour réaliser l'opération demandée est un composant de type `DateTimeConverter` pour lequel on spécifie le format de date à respecter grâce à la méthode `setPattern(…)`. Le designer de l'application web, dont le métier ne l'amène pas à manipuler les classes Java, peut facilement mettre en œuvre l'utilisation de ce convertisseur en utilisant la balise `<f:convertDateTime>` de la bibliothèque Core, en précisant une valeur adaptée pour son attribut `pattern`.

Pour associer ce convertisseur au champ de saisie de la date de naissance, la balise `<f:convertDateTime>` doit être placée entre les balises `<h:inputText>` et `</h:inputText>` représentant le composant `UIInput` de la date.

Concrètement, les modifications à apporter dans le code source sont les suivantes :

```
<h:inputText id="itxtDateNaissance"
converterMessage="La date indiquée doit être au format JJ/MM/AAAA" >
     <f:convertDateTime pattern="dd/MM/yyyy" />
</h:inputText>
```

Dans le cas où la conversion ne peut être réalisée (format non respecté par l'utilisateur), une `ConverterException` est générée. Le motif de l'échec de conversion est généralement disponible sous la forme d'un `FacesMessage`, mais il est éventuellement possible de surcharger simplement ce message si celui-ci ne convient pas (notamment lorsque la langue utilisée pour le message n'est pas la même que celle employée au sein de l'interface web). Pour cela, le designer web ajoute un attribut `converterMessage` à la balise `<h:inputText>` dont la valeur correspond au message à afficher en cas d'erreur de conversion.

© ENI Editions - All rigths reserved

Gestion des événements

La gestion événementielle au sein des applications Java Server Faces s'appuie sur l'utilisation d'un ensemble de classes et d'interfaces spécifiques définies dans le package `javax.faces.event`.

1. Principes mis en œuvre dans la gestion événementielle avec JSF

La classe de base à partir de laquelle sont définis les différents types d'événements est `FacesEvent`. Il s'agit d'une classe abstraite qui permet à toutes ses classes dérivées de maintenir une référence sur l'objet à l'origine de l'événement, ainsi que sur la description de l'événement lui-même.

Pour qu'une application puisse réagir à des événements, celle-ci doit disposer de classes implémentant des écouteurs (listeners) et référencer ceux-ci auprès des composants susceptibles de générer les événements.

Lorsqu'un utilisateur active un composant, en cliquant sur un bouton par exemple, un événement particulier est déclenché. À ce moment, l'application JSF invoque la méthode de l'écouteur conçu pour prendre en charge l'événement en question.

2. Deux types d'événements exploitables

Deux types d'événements peuvent actuellement être exploités au sein des applications JSF : les événements de type `action` et les événements de type `value-change`.

Un événement de type `action` survient lorsque l'internaute active un composant graphique basé sur une classe implémentant l'interface `ActionSource`. Parmi ces composants, on trouve les boutons et les hyperliens, tous deux basés sur la classe `UICommand`.

Un événement de type `value-change` survient lorsque l'utilisateur change la valeur d'un composant graphique basé sur la classe `UIInput`. Cela peut correspondre, par exemple, à la sélection d'une case à cocher, à la modification du texte contenu dans une zone de texte. Les composants standards suivants sont en mesure de déclencher ce type d'événement : `UIInput`, `UISelectOne`, `UISelectMany` et `UISelectBoolean`. Notez que les événements `value-change` ne sont générés que si le processus de validation de la valeur du composant concerné s'est réalisé avec succès.

3. Rôle de la propriété "immediate" des composants générateurs d'événements

Tous les composants graphiques qui viennent d'être cités, susceptibles d'être à l'origine de la génération d'un événement, disposent d'une propriété nommée `immediate`. Cette propriété peut être lue ou modifiée respectivement par l'appel des méthodes `isImmediate()` et `setImmediate()`.

Suivant la valeur de cette propriété, les événements de type `action` peuvent être traités durant la phase *Application phase* ou durant la phase *Apply Request Values*. Les événements de type `value-change` sont, quant à eux, traités soit durant la phase *Process Validators*, soit durant la phase *Apply Request Values*.

4. Réaction aux événements

Il existe deux moyens exploitables pour que les composants positionnés sur une page web puissent répondre aux différents types d'événements :

- Un écouteur, qui est un objet issu d'une classe implémentant l'interface `javax.faces.event.FacesListener`, ou l'une de ses interfaces dérivées, est chargé de capturer les événements d'un type précis. Cet écouteur doit être associé aux composants susceptibles de provoquer l'événement, par l'ajout d'une balise `<f:valueChangeListener>` ou d'une balise `<f:actionListener>` entre les balises délimitant la définition des composants eux-mêmes.

- Affecter une méthode particulière d'un bean managé ou d'un JavaBean à la prise en charge des événements. Le nom de la méthode en question est connu des composants susceptibles de déclencher les événements, par l'usage d'attributs spécifiques dans les balises définissant les composants.

De plus amples renseignements concernant la mise en œuvre de la gestion événementielle sont fournis dans le titre Association d'écouteurs aux composants présents dans la page du chapitre Exploitation des

© ENI Editions - All rigths reserved

composants standards JSF.

© ENI Editions - All rigths reserved

Utilisation de messages

1. Génération des messages d'erreur issus des processus de validation ou de conversion

Il arrive qu'au moment la soumission d'un formulaire web, la validation et/ou la conversion des données saisies échouent. Cela se produit par exemple lorsqu'une information ne respecte pas un format requis (échec de la conversion), ou lorsqu'une valeur numérique saisie ne fait pas partie d'une plage de valeurs particulière (échec de la validation). Dans ces situations, les convertisseurs et/ou les validateurs associés aux composants graphiques génèrent une exception ayant pour effet d'interrompre la soumission du formulaire.

Ces exceptions, de type `ConverterException` ou `ValidatorException`, se chargent de mettre à disposition une chaîne de caractère précisant le motif de l'échec de la conversion ou de la validation. Cette chaîne est utilisée pour instancier un objet issu de la classe `javax.faces.application.FacesMessage`.

2. Affichage de messages dans l'interface web

Au moment du réaffichage du formulaire web, le contenu d'un `FacesMessage` a la possibilité d'être présenté à l'utilisateur. Pour cela, au cours de la conception du formulaire, il est nécessaire de placer un composant graphique de type `UIMessage` et de l'associer à un autre composant susceptible de provoquer un échec de validation ou de conversion au cours de la soumission du formulaire. Cet `UIMessage` aura alors pour rôle de présenter le contenu des `FacesMessage` concernant le composant graphique auquel il est rattaché.

Ainsi, les lignes suivantes correspondent à la mise en œuvre des messages dans le code source du formulaire proposé en illustration de ce chapitre :

```
<h:message id="msgNom" for="itxtNom"
     style="color: #FF0000; background-color: #FFFF00">
</h:message>
...
<h:message id="msgPrenom" for="itxtPrenom" style="color: #FF0000;
background-color: #FFFF00"></h:message>
...
<h:message id="msgDateNaissance" for="itxtDateNaissance"
style="color: #FF0000; background-color: #FFFF00"></h:message>
```

Chaque composant de type `UIMessage` est représenté par une balise `<h:message>`, issue de la bibliothèque HTML. Cette balise dispose d'un attribut `for` dont la valeur correspond à la valeur de l'attribut `id` du composant graphique associé. Par exemple, le message *msgNom* est associé au champ de type `UIInput` dont l'identifiant est *itxtNom*. À chaque fois qu'un problème de conversion ou de validation survient à propos de ce champ, le motif du problème pourra être présenté à l'utilisateur par l'intermédiaire du message *msgNom*.

3. Surcharge des informations contenues dans les FacesMessage

Les composants graphiques de type `UIMessage` placés au sein des pages web servent donc à présenter à l'utilisateur le contenu du message stocké dans les éventuels `FacesMessage` générés au cours des processus de validation et de conversion. Il arrive cependant que le contenu brut du message en question ne convienne pas en vue d'un affichage. C'est notamment le cas lorsque le message comporte des détails techniques n'intéressant pas l'utilisateur final de l'application web, ou lorsque la langue utilisée dans le message n'est pas la même que celle employée dans le reste du site.

Dans ces situations, il est possible de surcharger le message contenu dans le `FacesMessage` par un message plus adapté. Ce nouveau message peut être précisé de manière simple par le designer de l'interface web, en ajoutant aux balises correspondant à des composants `UIInput` l'un des attributs suivants :

- `converterMessage` : utilisé pour surcharger les erreurs de conversion.
- `validatorMessage` : utilisé pour surcharger les erreurs de validation.
- `requiredMessage` : utilisé pour préciser le message à présenter lorsqu'aucune information n'a été fournie dans un champ obligatoire. Attention, pour que cet attribut soit pris en compte, il faut que la balise représentant le

© ENI Editions - All rigths reserved

composant de saisie dispose également d'un attribut required ayant une valeur égale à true.

Les lignes ci-dessous présentent les modifications à apporter au code source du formulaire proposé en illustration de ce chapitre, pour mettre en œuvre la surcharge de messages.

```
<h:inputText id="itxtNom" requiredMessage="N'oubliez pas de fournir
une valeur dans ce champ!" required="true">
...
</h:inputText>
<h:inputText id="itxtDateNaissance"
          validatorMessage="La valeur fournie est invalide."
          converterMessage="La date indiquée doit être au format
          JJ/MM/AAAA"
          requiredMessage="Veuillez fournir une valeur" required="true">
</h:inputText>
```

4. Les groupes de messages

Il est également possible d'utiliser un unique composant graphique JSF chargé d'afficher sous forme de liste l'ensemble des messages d'erreurs générés lors de la soumission d'un formulaire. Ce composant, de type javax.faces.component.UIMessages, n'est alors pas spécifique d'un champ de saisie particulier, au contraire des champs de type UIMessage. Il prend en charge la totalité des messages d'erreurs éventuellement issus des processus de validation et de conversion des données saisies dans le formulaire.

L'usage des UIMessages peut notamment être intéressant lorsque l'affichage des messages d'erreurs en regard des champs de saisie ne présente pas d'avantages, ou lorsque sur un plan purement ergonomique, il est souhaitable que les messages apparaissent toujours à la même position sur la page web (par exemple, toujours en haut du formulaire de saisie). Il faut toutefois prendre garde à ce que les messages à afficher soient suffisamment explicites pour que l'utilisateur soit capable de déterminer à quels éléments de formulaire ils se rapportent. En l'occurrence, il est souhaitable que le label des champs en question soit mentionné dans les messages.

Dans l'exemple illustrant ce chapitre, le designer web de l'application peut insérer un composant de type UIMessages en positionnant sur la page une balise <h:messages>. Différents attributs peuvent être ajoutés à cette balise, pour permettre notamment de préciser le style à utiliser pour l'affichage des messages. Dans le code source ci-dessous, le groupe de messages est présenté à l'internaute en italique, en majuscules, avec une taille de caractères de 10pt. Le texte est écrit en bleu sur fond jaune.

```
<f:view>
<h:form id="formulaireInscription">
<table align="center">
...
<tr>
<td colspan="3">
<h:messages id="groupeDeMessages" style="color: #0000FF; text-transform:
uppercase; font-style: italic; font-size: 10px; background-color:
#FFFF00; font-weight: bold" layout="table">
</h:messages>
</td>
</tr>
</table>
</h:form>
</f:view>
```

5. Regroupement des contenus de messages dans des fichiers de ressources

Pour garantir une bonne cohérence des messages d'erreurs délivrés sur l'ensemble des pages d'une application web, il est fortement recommandé de définir des messages d'erreur types au sein d'un fichier texte particulier. Il est par exemple souhaitable que le message présenté à l'utilisateur lorsqu'il oublie de fournir une valeur dans un champ obligatoire soit toujours le même, quel que soit le formulaire de l'application web affiché à l'écran.

Dans la terminologie Java Server Faces, le fichier texte contenant les messages types est un fichier de ressources, dont le nom possède obligatoirement l'extension **properties** et dont l'existence est précisée à l'application web au travers du fichier de configuration **faces-config.xml**.

Ainsi, dans l'exemple illustrant le chapitre, il est possible de définir un fichier de ressources pour regrouper les messages types à afficher lorsque l'utilisateur omet de renseigner un champ requis, ou lorsqu'il saisit une valeur

© ENI Editions - All rigths reserved

incorrecte. Pour cela, les lignes suivantes peuvent être placées au sein d'un fichier dénommé **inscriptionsMessages.properties** :

```
valeurRequise=Veuillez fournir une valeur pour ce champ!
valeurErronee=La valeur mentionnée dans ce champ est incorrecte!
```

La déclaration du fichier de ressources auprès de l'application web se fait dans le fichier **faces-config.xml** par l'ajout des lignes suivantes :

```
<faces-config
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd"
    version="1.2">
<application>
<resource-bundle>
<base-name>
premierProjetJSF.inscriptionsMessages
</base-name>
<var>msgs</var>
</resource-bundle>
...
</application>
...
</faces-config>
```

Le contenu de ce fichier de configuration indique qu'un fichier de ressources nommé **inscriptionsMessages.properties** est disponible dans le sous-répertoire **src/premierProjetJSF** de l'application web. Il précise également que le fichier de ressources en question peut être référencé au sein des pages web par l'usage d'une variable nommée *msgs*.

De fait, si l'on veut exploiter le message type proposé dans le fichier de ressources pour indiquer à l'utilisateur qu'il a oublié de saisir une valeur dans le champ `UIInput` correspondant à la date de naissance, les modifications suivantes doivent être apportées par le designer web au code source du formulaire :

```
<h:inputText id="itxtDateNaissance"
     validatorMessage="La valeur fournie est invalide."
     converterMessage="La date indiquée doit être au format
JJ/MM/AAAA"
     requiredMessage="#{msgs.valeurRequise}" required="true">
</h:inputText>
```

L'utilisation de fichiers de ressources pour les messages est également requise dans le cadre de l'internationalisation des applications web. Pour plus d'informations à ce sujet, reportez-vous au chapitre Composants graphiques personnalisés de cet ouvrage.

© ENI Editions - All rigths reserved

Déclaration et configuration de bean managés

Les applications Java Server Faces sont prévues pour permettre la gestion de beans s'exécutant côté serveur. Ces beans peuvent être exploités en association avec les composants graphiques présentés dans les formulaires web. Dans ce cas, ils sont qualifiés de beans managés.

L'utilité principale de ces beans managés se résume aux actions suivantes :

- Validation de la donnée saisie dans un composant graphique.
- Prise en charge des événements générés par un composant.
- Réalisation du traitement permettant de définir vers quelle page la navigation doit se poursuivre après la validation du formulaire.

Ces actions sont mises en œuvre au travers de méthodes définies dans le bean.

Ce paragraphe illustre la déclaration, la configuration et l'utilisation d'un bean managé, dans le cadre de l'exemple d'application web proposé en début de chapitre.

1. Création d'une classe de bean s'exécutant côté serveur

Comme tous les JavaBeans, un bean s'exécutant côté serveur doit disposer d'un constructeur sans argument. Celui-ci doit nécessairement être déclaré, même si son contenu est vide, de manière à ce que le bean puisse être instancié par l'application web : son absence provoque une erreur d'exécution.

Un bean managé doit également définir autant de propriétés que de champs de saisie auquel il peut potentiellement être lié. Ces propriétés doivent être de portée privée, et disposer d'accesseurs en lecture et en écriture pour rendre leur accès possible depuis les pages web de l'application, ou depuis d'autres beans ou classes. De surcroît, un bean managé peut proposer des méthodes chargées de réaliser des contrôles ou actions spécifiques, comme cela vient d'être précisé plus haut. Naturellement, toute classe déjà existante respectant ces conventions, applicables à l'ensemble des JavaBeans, peut être utilisée en tant que bean managé d'une application web JSF.

Le code source ci-dessous présente le contenu d'une classe potentiellement utilisable en association avec le formulaire web proposé en début de chapitre. Cette classe présente des propriétés privées en mesure de stocker les valeurs saisies dans les différents champs du formulaire (nom, prénom, date de naissance, etc.), et expose des accesseurs publics utilisables pour lire ou modifier les propriétés en question. Enfin, cette classe propose une méthode publique dénommée `inscriptionRealisee()` dont le rôle peut être d'assurer une navigation dynamique dans l'application web (voir le titre Navigation entre les pages de ce chapitre). La classe ci-dessous est bien conforme aux conventions applicables aux JavaBeans ; elle peut être donc être exploitée en tant que bean managé dans l'application web.

```
package premierProjetJSF;

import java.util.Date;

public class BeanInscription {
     private String nom;
     private String prenom;
     private Date dateNaissance;
     private String sexe;
     private int nombreEnfants;

     public BeanInscription()
     {
     //Constructeur par défaut, obligatoire,
     //même si son contenu est vide.
     }

     public String getNom() {
          return nom;
     }
     public void setNom(String nom) {
          this.nom = nom;
     }
```

© ENI Editions - All rigths reserved

```
     public String getPrenom() {
          return prenom;
     }
     public void setPrenom(String prenom) {
          this.prenom = prenom;
     }
     public Date getDateNaissance() {
          return dateNaissance;
     }
     public void setDateNaissance(Date dateNaissance) {
          this.dateNaissance = dateNaissance;
     }
     public String getSexe() {
          return sexe;
     }
     public void setSexe(String sexe) {
          this.sexe = sexe;
     }
     public int getNombreEnfants() {
          return nombreEnfants;
     }
     public void setNombreEnfants(int nombreEnfants) {
          this.nombreEnfants = nombreEnfants;
     }

     public String inscriptionRealisee()
     {
     //Retourne une chaîne de caractères dont la valeur
     //peut être utilisée pour réaliser de la navigation
     //dynamique via JSF.
     return "echecInscription";
     }
}
```

Lorsqu'une propriété de bean managé est liée à un composant graphique de formulaire web, son type doit être primitif, à moins que l'application puisse disposer d'un convertisseur adapté au type personnalisé de la propriété du bean.

La liaison de composants graphiques à des propriétés de beans managés est étudiée au chapitre Exploitation des composants standards JSF ; un exemple plus approfondi de création de bean personnalisé est présenté au chapitre Convertisseurs et validateurs de cet ouvrage.

2. Configuration d'un bean managé

Une fois que la classe définissant le bean managé est constituée, elle doit être rendue disponible à l'application web pour que celle-ci puisse l'exploiter. Cette action se fait au travers du fichier de configuration **faces-config.xml**, dans lequel il est possible de préciser :

- Le nom de la classe correspondant au bean managé.
- Le nom du bean tel qu'il pourra être utilisé dans les pages de l'application web.
- La portée du bean (page, session ou application).
- Les valeurs par défaut potentiellement prises par les propriétés du bean au moment de son initialisation.

La classe `BeanInscription` présentée ci-dessus peut ainsi être mise à la disposition de l'application web au moyen des lignes suivantes, à insérer dans le fichier **faces-config.xml** :

```
<faces-config
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd"
```


© ENI Editions - All rigths reserved

```
    version="1.2">
...
<managed-bean>
          <managed-bean-name>
          beanInscription</managed-bean-name>
          <managed-bean-class>
          premierProjetJSF.BeanInscription</managed-bean-class>
          <managed-bean-scope>
          session</managed-bean-scope>
          <managed-property>
               <property-name>
               nom</property-name>
               <property-class>
               java.lang.String</property-class>
               <value>
               DUPOND</value>
          </managed-property>
          <managed-property>
               <property-name>
               prenom</property-name>
               <property-class>
               java.lang.String</property-class>
               <value>
               Jacques</value>
          </managed-property>
          <managed-property>
               <property-name>
               nombreEnfants</property-name>
               <property-class>
               int</property-class>
               <value>
               0</value>
          </managed-property>
          <managed-property>
               <property-name>
               sexe</property-name>
               <property-class>
               java.lang.String</property-class>
               <value>
               masculin</value>
          </managed-property>
     </managed-bean>
...
</faces-config>
```

3. Utilisation du langage d'expressions pour exploiter les beans managés

Dès lors que le fichier de configuration de l'application web comporte des éléments indiquant l'utilisant de beans managés, ceux-ci sont instanciés au moment du lancement de l'application web elle-même. À partir du moment où l'un d'eux est utilisé par une page web quelconque de l'application, les éventuelles valeurs initiales des propriétés du bean sont appliquées et le bean est positionné dans la portée prévue.

Dans l'exemple proposé ci-dessus, le bean managé nommé `beanInscription` est instancié à partir de la classe `BeanInscription` au moment du lancement de l'application web. Lorsqu'un premier internaute accède à une page utilisant ce bean managé, ce dernier est placé dans la portée session, et ses propriétés `nom`, `prenom`, `nombreEnfants` et `sexe` sont initialisées avec les valeurs prévues dans **faces-config.xml**.

Lorsque des composants graphiques JSF sont liés aux propriétés de beans managés, ils peuvent facilement être utilisés pour lire ou mettre à jour ces propriétés. Le designer web met en œuvre cette possibilité en utilisant le langage d'expression comme valeur de certains attributs représentant les composants graphiques. Par exemple, pour qu'un champ `UIOutput` placé dans une page web puisse afficher le contenu de la propriété `prenom` du bean managé `beanInscription`, la balise suivante doit être utilisée :

```
<h:outputText value="#{beanInscription.prenom}" />
```

La valeur de l'attribut `value` de cette balise exploite bien le langage d'expression, grâce auquel sont précisés entre les séquences de caractères `#{` et `}` le nom du bean managé et celui de la propriété recherchée.

© ENI Editions - All rigths reserved

Plus généralement, le langage d'expression est utilisé avec certains attributs des balises correspondant aux composants graphiques, à la fois pour lire et modifier les valeurs des propriétés auxquelles ils sont liés. Ce langage peut également être exploité pour invoquer les méthodes de beans managés dans le but :

- De gérer les événements survenant au niveau des composants graphiques.
- De valider les données mentionnées dans les champs de saisie.
- De convertir ces mêmes données.

Dans le code source présenté ci-dessous, le langage d'expression est utilisé au sein de l'attribut `action` de la balise `<h:commandButton>`. Il permet l'invocation de la méthode `inscriptionRealisee()` du bean managé `beanInscription`, pour déterminer dynamiquement le cas de navigation lors de la soumission du formulaire.

```
<h:commandButton
    id="cmdValider"
    value="Valider votre inscription"
    action="#{beanInscription.inscriptionRealisee}"
></h:commandButton>
```


© ENI Editions - All rigths reserved

Navigation entre les pages

Le modèle de navigation proposé par la technologie Java Server Faces est conçu pour permettre aux architectes d'applications web de définir rapidement et facilement l'ordre de succession des pages visitées. La souplesse de sa constitution garantit par ailleurs la possibilité de modifier simplement la navigation entre les pages, en cas d'évolution de la logique métier de l'application ou de la structure du site elle-même.

1. Définition de règles de navigation

Dans la terminologie JSF, la navigation est définie comme un ensemble de règles utilisé pour déterminer la page qui doit être présentée à l'internaute une fois que celui-ci a cliqué sur un bouton de formulaire ou sur un lien hypertexte. Ces règles sont précisées au sein du fichier de configuration des ressources de l'application, **faces-config.xml**, à partir d'un ensemble d'éléments XML.

La démarche à adopter pour mettre en œuvre le processus de navigation JSF consiste donc tout d'abord à déclarer dans le fichier **faces-config.xml** les différentes règles, pour lesquelles sont notamment précisés le cas de navigation qui leur correspond, ainsi que les pages sources et cibles. Ces choix, réalisés par l'architecte de l'application web, doivent ensuite être pris en compte par le designer web au moment de la conception des pages de l'application. Celui-ci doit en effet exploiter convenablement les cas de navigation dans les attributs `action` des boutons et liens hypertexte qu'il met en place.

Deux types de navigation JSF sont couramment distingués : la navigation simple, dans laquelle les cas de navigation sont mentionnés de manière statique au sein des pages web de l'application, et la navigation dynamique, qui fait appel aux beans managés, chargés de générer dynamiquement les cas de navigation compte tenu du contexte. Quel que soit le type de navigation, la définition des règles de navigation reste identique.

Le code source ci-dessous est un extrait du fichier **faces-config.xml** correspondant à l'application web illustrant ce chapitre.

```
<faces-config
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd"
    version="1.2">
...
<navigation-rule>
          <display-name>
          inscription</display-name>
          <from-view-id>
          /inscription.jsp</from-view-id>
          <navigation-case><from-outcome>succesInscription</from-outcome>
               <to-view-id>
               /confirmationInscription.jsp</to-view-id>
          </navigation-case>
     </navigation-rule>
<navigation-rule>
          <display-name>
          inscription</display-name>
          <from-view-id>
          /inscription.jsp</from-view-id>
          <navigation-case><from-outcome>
               echecInscription</from-outcome>
               <to-view-id>
               /inscriptionImpossible.jsp</to-view-id>
          </navigation-case>
     </navigation-rule>
...
</faces-config>
```

Ce code source présente deux règles de navigation, définies grâce aux balises `<navigation-rule>`. Pour chacune de ces règles sont précisées les informations suivantes :

- Le nom de la page web à partir de laquelle l'internaute provoquera la navigation : cette page source est mentionnée par l'intermédiaire de la balise `<from-view-id>`.

© ENI Editions - All rigths reserved

- La description du cas de navigation, caractérisé par une balise `<navigation-case>` et deux balises enfants, `<from-outcome>` et `<to-view-id>`. La première de ces deux balises enfants encadre une chaîne de caractères identifiant le nom du cas de navigation en tant que tel. Ce peut être cette chaîne qui est recherchée au moment où l'internaute active un bouton ou un lien hypertexte, pour identifier la règle de navigation à exploiter. La deuxième balise enfant, `<to-view-id>`, précise le nom de la page web vers laquelle la navigation est orientée en cas d'utilisation du cas de navigation.

Le choix de la chaîne de caractères identifiant un cas de navigation spécifique (balise `<from-outcome>`) peut être fait librement. Il existe cependant quelques valeurs communément utilisées dans les applications web :

- **success** : employé lorsque toutes les vérifications sont réalisées avec succès.

- **failure** : employé lorsqu'une erreur au moins est identifiée par le processus métier chargé de réaliser les contrôles (voir le paragraphe Utilisation de la navigation dynamique). La navigation est alors renvoyée vers une page d'erreur.

- **Logon** : signifie que l'internaute doit passer par une étape d'authentification avant de poursuivre la navigation : il est réorienté vers une page comportant un formulaire d'authentification.

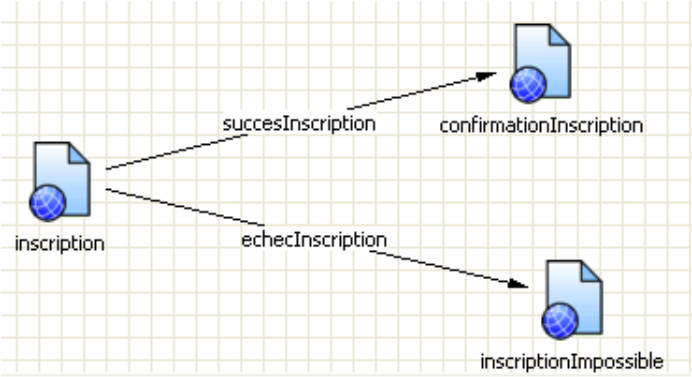
- **no results** : utilisé dans les cas où un processus de recherche (dans une base de données, par exemple) aboutit à une absence de résultats. La navigation est redirigée vers le formulaire de recherche.

Compte tenu de ces explications, le code source présenté plus haut montre que les deux règles de navigation sont applicables à partir d'une unique page web source nommée **inscription.jsp**. La navigation est rendue possible depuis cette page, soit par navigation simple, soit par navigation dynamique, vers la page **confirmationInscription.jsp** lorsque le cas de navigation **succesInscription** est utilisé, ou vers la page **inscriptionImpossible.jsp** si le cas de navigation **echecInscription** est employé.

Pour ce cas particulier, deux règles de navigation distinctes ont été définies, alors que celles-ci concernent la même page web source. Dans une telle situation, il est tout à fait possible de ne définir qu'une seule règle, regroupant les deux cas de navigation évoqués, de sorte que le code source suivant soit équivalent à celui montré précédemment :

```
<navigation-rule>
          <display-name>
          inscription</display-name>
          <from-view-id>
          /inscription.jsp</from-view-id>
          <navigation-case><from-outcome>succesInscription</from-outcome>
               <to-view-id>
               /confirmationInscription.jsp</to-view-id>
          </navigation-case>
          <navigation-case><from-outcome>
               echecInscription</from-outcome>
               <to-view-id>
               /inscriptionImpossible.jsp</to-view-id>
          </navigation-case>
</navigation-rule>
```

Sous l'environnement de développement Eclipse, ces deux cas de navigation peuvent être visualisés graphiquement, comme le montre la figure ci-dessous.



© ENI Editions - All rigths reserved

2. Utilisation de la navigation simple

Comme cela a été précisé plus haut, la navigation simple concerne les situations où les cas de navigation sont spécifiés de manière statique au sein des pages web de l'application. Pour cela, le designer web mentionne simplement le nom du cas de navigation à exploiter grâce à l'attribut `action` des composants de type `UICommand` qu'il dispose sur l'interface (boutons, liens hypertexte).

Ainsi, dans l'exemple proposé pour ce chapitre, la page web **inscription.jsp** peut être conçue de telle façon que la navigation s'oriente systématiquement vers la page **confirmationInscription.jsp**, en cas de soumission du formulaire d'inscription. Pour cela, il est nécessaire de faire appel au cas de navigation **succesInscription**, défini dans le fichier **faces-config.xml** pour permettre la navigation de la première page vers la seconde. L'attribut `action` de la balise `<h:commandButton>` représentant le bouton de soumission du formulaire est donc positionné en conséquence :

```
<h:commandButton id="cmdValider"
                 value="Valider votre inscription"
                 action="succesInscription">
</h:commandButton>
```

3. Utilisation de la navigation dynamique

Mais il est courant que la navigation simple ne convienne pas, en particulier dans le cadre d'applications web à vocation professionnelle ou commerciale. Bien souvent en effet, la navigation est rendue dépendante d'un ensemble de facteurs liés au contexte dans lequel s'inscrit le système d'information mis en place. La navigation peut par exemple varier suivant le profil de l'internaute, en autorisant la visualisation de certaines pages, tout en interdisant l'accès à d'autres ressources. Pour permettre la conception de telles applications, Il est nécessaire de mettre en œuvre le concept de navigation dynamique.

Avec la technologie Java Server Faces, ce type de navigation est rendu possible par l'exploitation de JavaBeans, en partie chargés de générer dynamiquement la chaîne identifiant le cas de navigation adapté au contexte. Concrètement, c'est une méthode spécifique de ces JavaBeans qui est identifiée pour réaliser les traitements nécessaires et renvoyer le cas de navigation attendu. Les traitements en question peuvent consister par exemple à vérifier les paramètres d'authentification fournis par l'internaute en interrogeant une base de données. Compte tenu de la réussite ou de l'échec de l'opération, deux chaînes de caractères distinctes peuvent être renvoyées par la méthode : ces chaînes doivent bien entendu correspondre à des cas de navigations prévus par l'architecte de l'application web, de manière à pouvoir être exploitées efficacement dans le cadre d'une navigation dynamique.

L'exemple proposé dans ce chapitre peut tout à fait être mis à profit pour illustrer ce type de navigation. Plutôt que d'orienter systématiquement la navigation depuis la page **inscription.jsp** vers la page **succesInscription.jsp**, comme cela a été vu dans le paragraphe précédent, la navigation peut être variable suivant les informations saisies par l'internaute, ou suivant le contexte d'application lui-même. Il est par exemple possible d'envisager le succès systématique d'une inscription (et donc la navigation vers **succesInscription.jsp**) dans tous les cas de figure, sauf lorsque l'une des situations suivantes se présente :

- Le nom mentionné ne correspond pas à l'une des valeurs autorisées, spécifiées dans une table de base de données.
- La base de données permettant de contrôler le nom n'est pas disponible.

Dans ces cas particuliers, la navigation doit être orientée vers la page **inscriptionImpossible.jsp**.

Pour rendre possible une telle navigation, le bean managé dénommé `BeanInscription` (présenté au paragraphe Création d'une classe de bean s'exécutant côté serveur de la section Déclaration et configuration de beans managés de ce chapitre) peut être utilisé. En déléguant l'une de ses méthodes à la réalisation des contrôles requis au moment d'une demande d'inscription, il est possible de générer dynamiquement la chaîne de caractères identifiant le cas de navigation à utiliser, compte tenu du résultat des contrôles. La méthode `inscriptionRealisee()`, dont une partie du code source est présentée ci-dessous, est un exemple de méthode assignée aux contrôles à entreprendre.

```
public String inscriptionRealisee()
     {
     //Retourne une chaîne de caractères dont la valeur
     //peut être utilisée pour réaliser de la navigation dynamique
     //via JSF.
     String casNavigation="succesInscription";

     if (!baseDeDonneesDisponible) casNavigation="echecInscription";
     if (nomInvalide) casNavigation="echecInscription";

     return casNavigation;
```

© ENI Editions - All rigths reserved

```
    }
```

Pour que cette méthode puisse être invoquée au moment de la soumission du formulaire d'inscription, le designer web doit mentionner une expression de méthode adaptée, en tant que valeur de l'attribut `action` de la balise `<h:commandButton>`. Cette expression de méthode doit faire référence au bean managé de type `BeanInscription`, tel qu'il est déclaré au sein du fichier **faces-config.xml**, ainsi qu'au nom de la méthode chargée de réaliser les contrôles.

```
<h:commandButton
    id="cmdValider"
    value="Valider votre inscription"
    action="#{beanInscription.inscriptionRealisee}"
></h:commandButton>
```

4. Rôle des instances par défaut de ActionListener et de NavigationHandler dans la navigation dynamique

Il peut être intéressant de s'attarder plus précisément sur la façon dont est géré le traitement de la navigation par le modèle de navigation JSF.

Au moment où l'internaute active un bouton ou un lien hypertexte au niveau de l'interface, le composant correspondant génère un événement de type `action`. Celui-ci est alors pris en charge par l'écouteur prévu par défaut par l'implémentation JSF : il s'agit d'un écouteur implémentant l'interface `javax.faces.event.ActionListener`, dont le rôle est d'appeler la méthode référencée par le composant à l'origine de l'événement, puis de récupérer la réponse fournie par cette méthode, c'est-à-dire une chaîne de caractère correspondant normalement à un cas de navigation prévu.

L'instance par défaut de `ActionListener` transmet ensuite cette chaîne de caractère à une instance de la classe `javax.faces.application.NavigationHandler` (il en existe une fournie par défaut par l'implémentation JSF). Ce `NavigationHandler` sélectionne alors la page suivante à afficher, en recherchant une correspondance entre le cas de navigation qu'il a reçu de l'`ActionListener` et l'ensemble des règles de navigation définies dans le fichier de configuration **faces-config.xml**.

Lorsque le `NavigationHandler` trouve une correspondance valide, la phase de restitution de la vue commence, et la nouvelle page à afficher est présentée à l'internaute.

© ENI Editions - All rigths reserved

Arbre de composants et cycle de vie d'une page JSF

Ce paragraphe est destiné à apporter des informations complémentaires à celles déjà fournies à la section Le cycle de vie à phases multiples d'une page JSF du chapitre Langage d'évaluation d'expressions. Il présente de plus amples détails concernant les différentes phases constituant le cycle de vie d'une application Java Server Faces.

Pour rappel, une page JSF diffère d'une page JSP par le fait qu'elle est représentée sous la forme d'un arbre de composants basés sur la classe `javax.faces.component.UIComponentBase`. Dans la terminologie JSF, l'arbre en question est appelé *vue*. Au cours du cycle de vie d'une page web ; l'implémentation JSF construit la vue, tout en prenant en compte les états précédents des différents composants (en cas de réaffichage de la page). Lorsque l'internaute soumet un formulaire, ou s'il engage une action impliquant une évolution de la navigation, deux tâches principales sont assurées : la validation des données saisies dans les différents composants graphiques, et la conversion des types de ces mêmes données en types compatibles avec les propriétés des Java Beans associés aux champs de saisie. Ces actions sont prises en charge de manière séquentielle au cours du cycle de vie : leur traitement est réparti en différentes phases dont le détail vous est présenté dans la suite de ce chapitre.

1. Les différents types de requêtes et leur mode de prise en charge

Les requêtes assurant la navigation au sein de l'application peuvent être vues de deux façons différentes par l'implémentation JSF : il s'agit soit de requêtes initiales, soit de requêtes de retour, dites postback. Les premières correspondent aux situations où les internautes consultent une page pour la première fois au cours d'une même session. Les secondes se rencontrent principalement lorsque les internautes valident des formulaires contenus dans des pages web, elles-mêmes résultant de la prise en charge d'une requête initiale.

a. Les objets FacesContext et FacesContextFactory

Un `FacesContext` est un type d'objet issu de la classe `javax.faces.context.FacesContext`. Il est conçu pour contenir toutes les informations relatives à une requête JSF spécifique. Il est associé à la requête dès le début du processus de traitement par un appel de la méthode `getFacesContext()` sur l'instance de la classe `javax.faces.context.FacesContextFactory`, obligatoirement disponible pour l'application web. Le `FacesContext` est ensuite transmis successivement dans chacune des phases du cycle de vie, pour y être potentiellement manipulé. Il est notamment utilisé, comme cela a été vu précédemment dans le processus de validation des données saisies.

b. Traitement d'une requête initiale

Seules les phases *Restore View* et *Render Response* sont mises en œuvre lors du traitement d'une requête initiale. En effet, aucune action de l'utilisateur n'est à prendre en compte dans cette situation, au contraire de ce qui est fait pour le traitement des requêtes postback, pour lesquelles la totalité des phases du cycle de vie est exploité.

Lorsqu'un internaute clique sur bouton ou un lien hypertexte pointant vers une page JSF, l'application web doit fournir une réponse correspondant à la page JSF demandée. Pour cela, elle crée une nouvelle vue (un nouvel arbre de composants) et la place dans une instance de la classe `FacesContext`.

Une fois la vue placée dans le `FacesContext`, l'application web récupère la référence des différents composants graphiques constituant la page. Dans le cas d'une requête initiale, l'application poursuit son action en appelant la méthode `renderResponse()` de l'objet `FacesContext`. Le traitement de la requête est alors directement transmis à la phase *Render Response*, outrepassant les autres phases intermédiaires.

c. Traitement des requêtes visant des ressources externes à l'application

Il arrive parfois que l'application doive prendre en charge une requête qui ne lui est pas destinée. Cela se produit notamment lorsque l'utilisateur tente d'atteindre un autre site web, ou lorsqu'une tentative d'accès à un service web est réalisée. Dans ce cas, la méthode `responseComplete()` du `FacesContext` est invoquée et la phase *Render Response* n'est pas traitée.

2. Les différentes phases du cycle de vie

La situation la plus courante est celle au cours de laquelle une action réalisée sur un composant graphique JSF provoque l'appel d'une autre page JSF de l'application. La requête correspondante est alors capturée par l'application. Elle est ensuite associée au `FacesContext`, puis transmise successivement aux différentes phases du cycle de vie pour que les éventuelles validations et conversions soient réalisées.

© ENI Editions - All rigths reserved

a. La phase Restore View

Dès qu'une requête à destination d'une page JSF est émise, l'implémentation Java Server Faces active le cycle de vie de la page en question par la phase *Restore View*.

Au cours de cette phase, l'arbre de composants de la page est constitué. Chaque composant graphique se voit attribuer ses éventuels gestionnaires d'événements et validateurs, et l'arbre de composant est enfin sauvegardé dans le `FacesContext`. Au sein de la page JSF, l'ensemble des composants, gestionnaires d'événements, convertisseurs et validateurs peuvent avoir accès au `FacesContext`.

Si la requête ayant enclenché le cycle de vie est une requête initiale, une nouvelle vue est constituée et le cycle de vie avance directement jusqu'à la phase *Render Response*.

Sinon, la requête est une requête postback et une vue correspondant à la page demandée existe déjà. L'implémentation JSF restaure alors cette vue (d'où le nom de la phase) à partir des informations d'état sauvegardées côté client et/ou côté serveur.

Concernant l'exemple proposé pour ce chapitre, la vue correspondant à la page **inscriptions.jsp**, dans laquelle se trouve le formulaire d'inscription, est constituée d'un composant `javax.faces.component.UIViewRoot`, faisant office de racine pour l'arbre de composants. Le premier composant enfant de cette racine correspond au formulaire d'inscription, représenté par un composant de type `javax.faces.component.UIForm`. Ce dernier composant possède à son tour plusieurs composants enfants, un pour chacun des éléments présentés dans le formulaire.

b. La phase Apply Request Values

À l'issue de la phase *Restore View*, l'arbre de composants correspondant à la page demandée est constitué ou restauré. Chacun des composants de cette vue récupère depuis la chaîne de requête la nouvelle valeur susceptible de lui être attribué. Cette opération se réalise au travers d'un appel à la méthode `decode()`, disponible dans la classe `UIComponentBase`, et donc accessible à tous les composants graphiques JSF. Les nouvelles valeurs sont alors stockées localement dans les composants eux-mêmes, à moins que le processus de conversion engagé sur ces valeurs provoque une erreur. Dans ce dernier cas, des messages d'erreurs sont alors constitués, puis placés dans le `FacesContext`. Chaque message, qui contient la description de l'erreur et la référence du composant en cause, pourra par la suite être restitué à l'internaute durant la phase *Render Response*, en même temps que les autres éventuels messages d'erreurs issus du processus de validation des données.

Au cours de cette phase, il est possible que l'exécution de la méthode de l'un des composants provoque l'appel de la méthode `renderResponse()` du `FacesContext`. Le cycle de vie évolue alors directement vers la phase *Render Response*.

Si des événements sont survenus durant la phase *Apply Request Values*, ceux-ci sont transmis aux écouteurs concernés. Par ailleurs, si certains des composants de l'arbre disposent d'un attribut `immediate` positionné sur la valeur `true`, les processus de conversion, de validation, ainsi que les événements associés sont traités immédiatement.

Par ailleurs, si l'application JSF est amenée à engager une navigation vers une ressource externe (autre site web, service web…), ou si elle doit produire une page web ne comportant aucun composant JSF, alors la méthode `responseComplete()` du `FacesContext` peut être invoquée.

À l'issue de cette phase, tous les composants contenus dans la vue possèdent leur nouvelle valeur. Les messages d'erreurs issus du processus de conversion, ainsi que les événements éventuels, sont placés en attente de traitement.

c. La phase Process Validations

Au cours de cette phase, tous les validateurs éventuellement affectés aux composants de l'arbre sont pris en considération. L'implémentation JSF examine les attributs de composants décrivant les règles à respecter pour la validation. Ces attributs sont alors comparés aux valeurs stockées localement dans le composant. La comparaison est assurée par appel de la méthode `validate()` de la classe `UIInput`.

Lorsque cette comparaison, pour un composant donné, se solde par la mise en évidence du non-respect de la règle énoncée, le processus de validation échoue. Un message d'erreur est alors stocké dans le `FacesContext` et le cycle de vie évolue directement vers la phase *Render Response* : la page est alors restituée à l'internaute, accompagnée du (des) message(s) d'erreur.

Tout comme pour la phase *Apply Request Values*, si l'application JSF est amenée à engager une navigation vers une ressource externe (autre site web, service web…) ou si elle doit produire une page web ne comportant aucun composant JSF, alors la méthode `responseComplete()` du `FacesContext` peut être invoquée.

Dans le cas particulier de l'exemple choisi pour illustrer ce chapitre, un validateur standard est associé au champ de saisie du nom. Celui-ci est utilisé pour vérifier que la longueur du nom est toujours comprise entre 5 et 100 caractères. Lorsque cette contrainte n'est pas respectée, une erreur de validation survient durant la phase *Process Validation*. La page contenant le formulaire est alors restituée à l'internaute, et le message d'erreur est présenté par l'intermédiaire des composants `UIMessage` ou `UIMessages` disposés sur l'interface web.

© ENI Editions - All rigths reserved

d. La phase Update Model Values

Une fois que l'ensemble des données saisies dans les composants graphiques a été validé, l'implémentation JSF parcourt à nouveau tous les composants et tente d'affecter leurs valeurs aux propriétés de JavaBeans auxquels ils sont liés. Si les valeurs en question ne peuvent pas être converties dans un type compatible avec celui des propriétés qui leur correspondent, alors le cycle de vie de la page évolue directement vers la phase *Render Response*. La page est alors de nouveau présentée à l'internaute. Les messages d'erreurs sont également affichés si des composants `UIMessage` ou `UIMessages` sont présents.

Ce processus de mise à jour des propriétés des JavaBeans est concrètement assuré par appel de la méthode `updateModel()` de chacun des composants graphiques issus de la classe `UIInput`. Il n'est donc appliqué qu'aux éléments de formulaire permettant de réaliser une saisie.

Là encore, si l'application JSF est amenée à engager une navigation vers une ressource externe (autre site web, service web…), ou si elle doit produire une page web ne comportant aucun composant JSF, alors la méthode `responseComplete()` du `FacesContext` peut être invoquée.

Si des événements surviennent durant cette phase, ceux-ci sont transmis aux écouteurs concernés.

e. La phase Invoke Application

Cette phase assure la prise en charge des événements de niveau application, tels que les clics sur les boutons ou les liens hypertextes.

Par exemple, dans l'illustration proposée pour ce chapitre, le formulaire est en mesure de provoquer un événement de niveau application. Celui-ci survient lorsque l'internaute clique sur le bouton de soumission du formulaire. Le traitement de cet événement est assuré par l'`ActionListener` associé à l'application web, qui se charge de récupérer le cas de navigation mentionné dans l'attribut `action` de la balise correspondant au bouton de soumission. L'`ActionListener` transmet ensuite ce cas d'utilisation au `NavigationHandler`, afin de déterminer la règle de navigation associée au cas de navigation spécifié. Le contenu de la règle de navigation ainsi identifiée précise le nom de la page web à afficher par la suite. L'implémentation JSF prend alors en compte la vue correspondant à cette page cible, et le cycle de vie se poursuit par la phase *Render Response*.

f. La phase Render Response

Il s'agit de la phase finale du cycle de vie, au cours de laquelle la responsabilité de la restitution de la page dans le navigateur est transmise au conteneur JSP. Tous les composants contenus dans la vue sont restitués par ce conteneur dans l'ordre des balises rencontrées dans la page.

Dans le cas d'une requête postback, des erreurs ont pu être détectées durant les phases *Apply Request Values*, *Process Validation*, et *Update Model Values*. La page présentée initialement est restituée telle quelle à l'utilisateur ; les messages d'erreurs sont affichés si des balises `<h:message>` ou `<h:messages>` sont présentes dans le code source de la page web.

© ENI Editions - All rigths reserved

Code source de l’application illustrant le chapitre

Pour clore ce chapitre, voici l’intégralité du code source correspondant à la page JSF comportant le formulaire de saisie. À la différence de la version présentée en début de chapitre, cette version comporte tous les éléments nécessaires au bon fonctionnement de l’application, notamment les validateurs et les convertisseurs. Elle est donc fournie ici dans l’unique but de faciliter l’analyse de ce code source, après les multiples ajouts qui y ont été faits au cours de ce chapitre.

```
<%@taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
<%@taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Formulaire d’inscription</title>
</head>
<body>

<f:view>
  <h:form id="formulaireInscription">
      <table align="center">
           <tr>
               <td colspan="3"><h:outputText
                    value="Saisissez les paramètres suivants pour valider
votre inscription:"
                       style="color: #0000FF; text-decoration:
underline"></h:outputText><br>
               <br>

               </td>


           </tr>
           <tr>
               <td colspan="3"><h:messages id="groupeDeMessages"
style="color: #0000FF; text-transform: uppercase; font-style: italic;
font-size: 10px; background-color: #FFFF00; font-weight: bold"
layout="table"></h:messages></td>


           </tr>
           <tr>
               <td><h:outputText id="otxtNom"
value="Nom*">
</h:outputText></td>
               <td><h:inputText id="itxtNom"
                    requiredMessage="N’oubliez pas de fournir une valeur
dans ce champ!">


                    <f:validateLength minimum="5" maximum="100">
</f:validateLength>
               </h:inputText></td>
               <td><h:message id="msgNom" for="itxtNom"
                    style="color: #FF0000; background-color: #FFFF00">
                   </h:message></td>
           </tr>
           <tr>
               <td><h:outputText id="otxtPrenom" value="Prénom">
</h:outputText></td>
               <td><h:inputText id="itxtPrenom" value="Prénom
par défaut"></h:inputText></td>
               <td><h:message id="msgPrenom" for="itxtPrenom"
                    style="color: #FF0000; background-color: #FFFF00">
</h:message></td>
```

© ENI Editions - All rigths reserved

```
           </tr>
           <tr>
               <td><h:outputText id="otxtDateNaissance"
                    value="Date de naissance*">
</h:outputText></td>
               <td><h:inputText id="itxtDateNaissance"
                    validatorMessage="La valeur fournie est invalide."
                    converterMessage="La date indiquée doit être
au format JJ/MM/AAAA"
                    requiredMessage="#{msgs.valeurRequise}"
required="true">
                    <f:convertDateTime pattern="dd/MM/yyyy"
/>
               </h:inputText></td>
               <td><h:message id="msgDateNaissance"
for="itxtDateNaissance"
                    style="color: #FF0000; background-color: #FFFF00">
</h:message></td>
           </tr>
           <tr>
               <td><h:outputText id="otxtSexe" value="Sexe*">
</h:outputText></td>
               <td><h:selectOneRadio id="radioSx">
                    <f:selectItem id="sexeChoix1" itemLabel="Masculin"
itemValue="Masc" />
                    <f:selectItem id="sexeChoix2" itemLabel="Féminin"
itemValue="Fem" />
               </h:selectOneRadio></td>
               <td><h:message id="msgSexe" for="radioSx"
                    style="color: #FF0000; background-color: #FFFF00">
</h:message></td>
           </tr>
           <tr>
               <td><h:outputText id="otxtNombreEnfants"
                    value="Nombre d’enfants">
</h:outputText></td>
               <td><h:selectOneListbox id="listeAge" size="1">
                    <f:selectItem id="itemZero" itemLabel="0"
itemValue="0" />
                    <f:selectItem id="itemUn" itemLabel="1"
itemValue="1" />
                    <f:selectItem id="itemDeux" itemLabel="2"
itemValue="2" />
                    <f:selectItem id="itemTrois" itemLabel="3"
itemValue="3" />
                    <f:selectItem id="itemQuatre" itemLabel="4"
itemValue="4" />
                    <f:selectItem id="itemCinqPlus" itemLabel="5
et plus" itemValue="5" />
               </h:selectOneListbox></td>
               <td></td>
           </tr>
           <tr>
               <td colspan="3" align="center"><h:commandButton
id="cmdValider"
                    value="Valider votre inscription" action=
"#{beanInscription.inscriptionRealisee}"></h:commandButton></td>


           </tr>
      </table>


  </h:form>
</f:view>
</body>
</html>
```


© ENI Editions - All rigths reserved

Description de l'environnement Eclipse pour JSF

La prise en charge de la technologie Java Server Faces par l'environnement Eclipse est facilitée par l'utilisation de différentes vues. Certaines sont bien évidemment communes aux autres types de projets s'appuyant sur Eclipse, mais d'autres sont spécifiques à JSF. Les principales vues concernent l'affichage de la zone d'édition, de l'explorateur de projets, de la sortie, des serveurs exploités par l'application, des propriétés, et de l'explorateur de bases de données. Voici un descriptif de chacune d'elles :

1. La zone d'édition

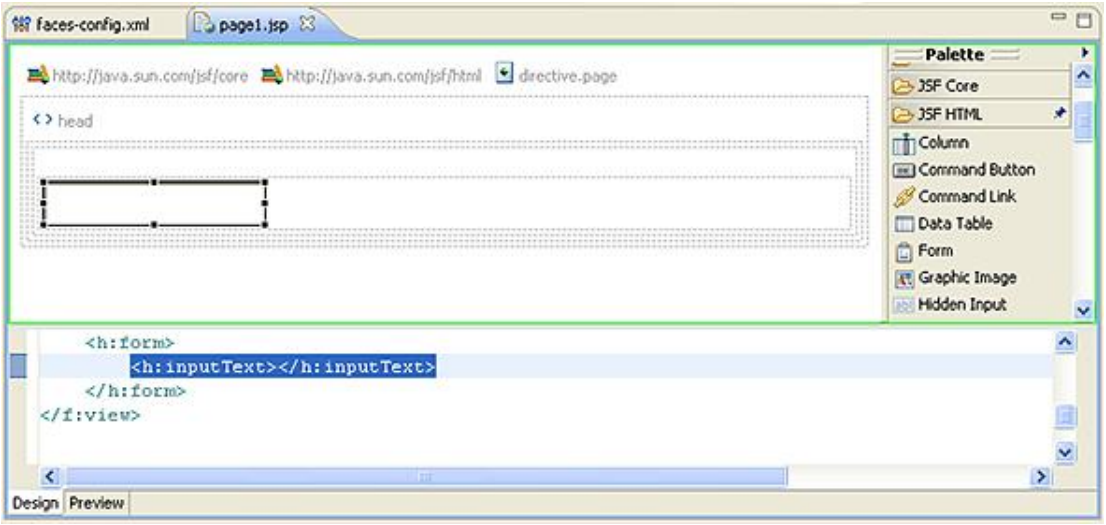
Il s'agit de la partie principale de l'environnement de développement. Son rôle principal consiste à présenter à l'utilisateur le contenu des divers fichiers listés dans l'explorateur de projets, ainsi qu'à lui faciliter la mise à jour de ces fichiers.

En l'occurrence, cette zone est spécifiquement adaptée à la création de pages JSP : elle permet en effet la présentation de la page en **mode code source** ou en **mode design**, et propose une **palette d'outils** autorisant le glissé-déposé d'objets divers, tels que des éléments de formulaires HTML, des composants JSP (balises taglig, balises param, etc.), et bien sûr des composants JSF (zones de saisies, convertisseurs, validateurs, etc.). La **prévisualisation des pages JSP** créées est également rendue possible par l'utilisation d'un onglet spécifique.

Enfin, la zone d'édition a la particularité de faciliter l'affichage et la mise à jour du fichier de configuration des ressources des applications JSF, dénommé par défaut **faces-config.xml**, par le biais de formulaires de saisie particuliers.

a. Édition d'une page JSP

La figure suivante donne un aperçu de la zone d'édition, au moment de l'affichage d'une page JSP.



Il est possible que cet affichage ne soit pas activé par défaut au moment de l'affichage des pages JSP. Dans ce cas, seul le code source de la page est visualisé à l'écran. Pour obtenir l'affichage présenté ci-dessus, il est alors nécessaire d'indiquer à Eclipse que l'éditeur à utiliser est le Web Page Editor : cela se fait simplement en cliquant droit sur le nom du fichier JSP depuis l'explorateur de projet, puis en sélectionnant le sous-menu **Open With - Web Page Editor**.

Dans cette situation, la zone d'édition est constituée de deux lignes. La première présente la zone d'affichage à proprement parler, ainsi que la palette d'outils à partir de laquelle il est possible de sélectionner les objets à positionner sur la page. La seconde ligne expose le code source de la page JSP affichée.

Deux onglets **Design** et **Preview** sont également visibles en bas de cette zone d'édition : leur usage permet de basculer entre les modes développement et prévisualisation de la page.

En mode d'édition de pages JSP, la **palette d'objets** présente un contenu spécifique, regroupant divers objets répartis en une vingtaine de catégories distinctes. Parmi ces catégories, il en existe quatre qui intéresseront particulièrement les concepteurs d'applications web basées sur les technologies JSP/JSF : il s'agit des catégories *HTML*, *JSF HTML*, *JSFCore* et *JSP*.

© ENI Editions - All rigths reserved

Catégorie HTML

Elle regroupe tous les objets correspondant aux éléments de formulaires HTML (zones de textes, zones de listes déroulantes, cases à cocher, etc.). L'utilisation de ces objets par glissé-déposé permet de concevoir rapidement des sections de pages web purement statiques, qui n'impliquent la mise en œuvre d'aucun dynamisme particulier.

Catégorie JSF HTML

Elle contient des objets facilitant l'intégration des composants graphiques JSF dans les pages web. On y retrouve donc la plupart des composants s'appuyant sur des classes dérivées de `UIComponentBase`, par exemple des objets de type `UICommande`, `UIForm` ou `UIInput`. Comme cela sera présenté plus en détail dans la suite de ce chapitre, le glissé-déposé d'un de ces objets sur la page web à créer entraîne la génération automatique de la balise JSF correspondante, dans le code source de la page. Il s'agit obligatoirement d'une balise issue de la bibliothèque HTML.

Catégorie JSF Core

Elle regroupe principalement les objets JSF qui peuvent être associés aux composants graphiques listés dans la catégorie JSF HTML. Il s'agit notamment des convertisseurs (objets implémentant l'interface `Converter`), des validateurs (objets implémentant l'interface Validator), et des écouteurs (objets implémentant l'interface `FacesListener`). Le fait de glisser-déposer l'un des objets de la catégorie *JSF Core* dans la page web à créer entraîne la génération automatique de la balise JSF correspondante, dans le code source de la page. Il s'agit obligatoirement d'une balise issue de la bibliothèque Core.

Catégorie JSP

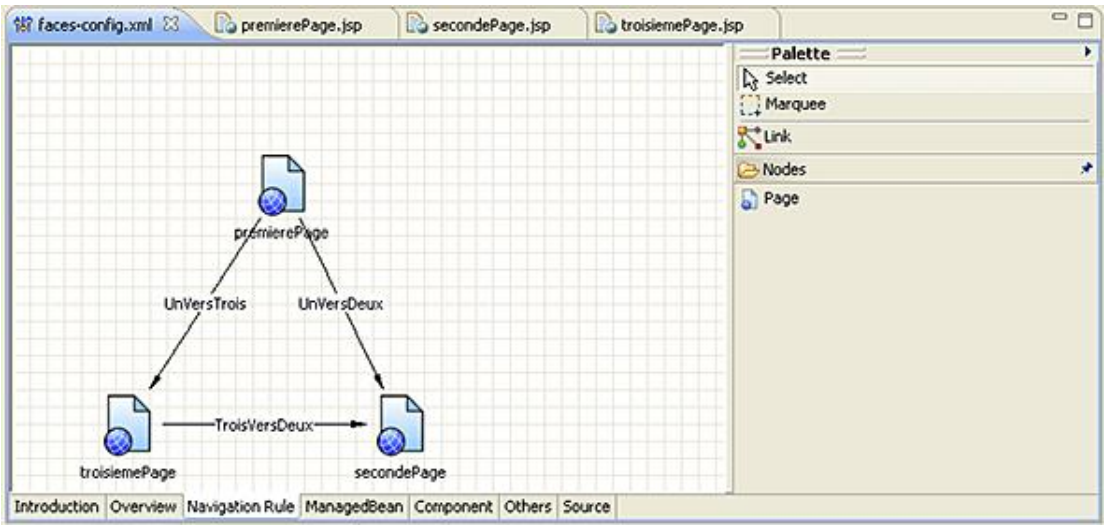
Elle regroupe des objets dont l'exploitation permet une intégration plus aisée des balises spécifiques à la technologie Java Server Pages, comme les balises correspondant à des directives, à des redirections, à l'usage de JavaBeans ou à l'intégration de scriptlets.

b. Édition du fichier de configuration des ressources des applications web

Outre la conception des pages JSP, la zone d'édition est également conçue pour faciliter la mise à jour du fichier de configuration des ressources des applications web. En double cliquant sur le nom de ce fichier (en général **faces-config.xml**) depuis l'explorateur de projets, différents formulaires de saisie sont proposés pour permettre la définition des règles de navigation, la déclaration des beans managés et de leurs propriétés, ainsi que la déclaration de divers autres composants, tels que les convertisseurs, les validateurs ou les écouteurs. Bien sûr, il est également possible de visualiser d'un seul coup d'œil l'ensemble du code source de ce fichier de configuration des ressources.

Définition des règles de navigation

Celle-ci se fait directement après la sélection de l'onglet **Navigation Rule** et se réalise exclusivement de manière graphique.



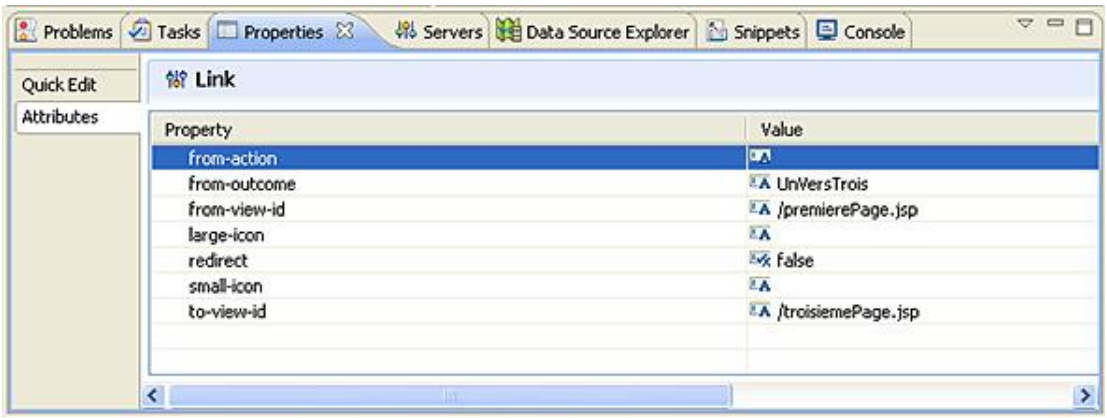
L'illustration ci-dessus présente les règles de navigation valables pour une application composée d'au moins trois pages JSP dénommées **premierePage.jsp**, **secondePage.jsp**, et **troisiemePage.jsp**.

Pour mettre ces règles en place, il faut dans un premier temps sélectionner l'outil **Page** situé dans la palette, puis intégrer successivement dans la zone d'édition chacune des pages concernées, par l'intermédiaire d'un formulaire

© ENI Editions - All rigths reserved

proposé par l'environnement de développement.

Il faut ensuite, dans un second temps, préciser les différents cas de navigation en utilisant l'outil **Link** disponible dans la palette. Un cas de navigation particulier se définit alors simplement en cliquant successivement sur la page source, puis sur la page cible. La mention d'une valeur pour les attributs `from-outcome` et `from-action` se fait par l'intermédiaire de la vue **Properties**.



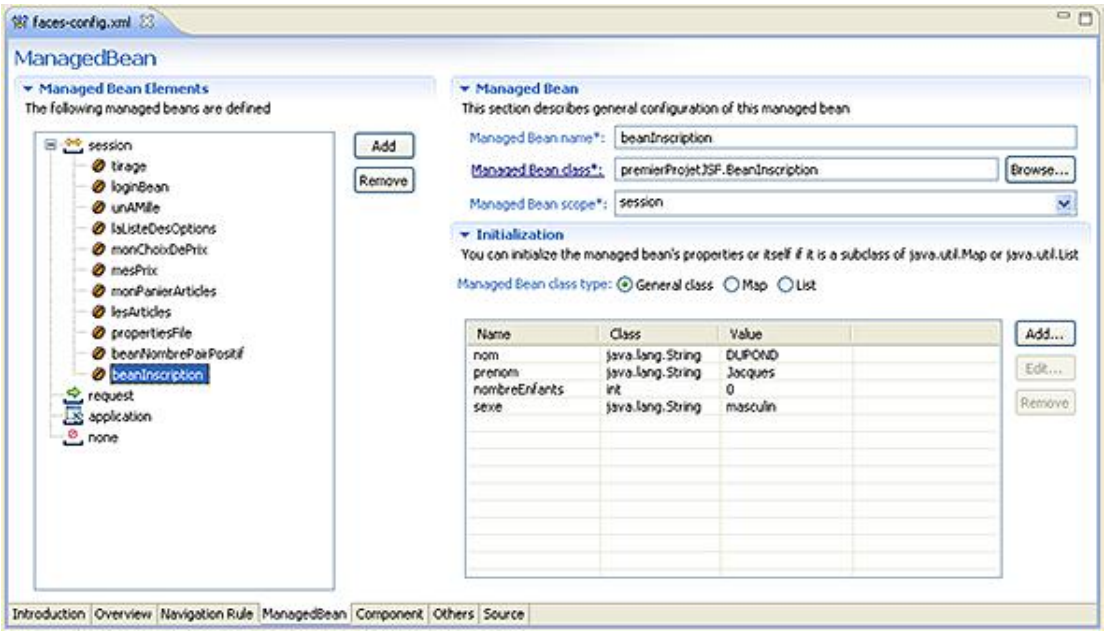
Un clic sur l'onglet **Source** de la zone d'édition permet de visualiser le code source du fichier de configuration. Les cas de navigation présentés ci-dessus correspondent au code suivant :

```
<faces-config
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd"
    version="1.2">
     <navigation-rule>
          <display-name>
          premierePage</display-name>
          <from-view-id>
          /premierePage.jsp</from-view-id>
          <navigation-case>
               <from-outcome>
               UnVersDeux</from-outcome>
               <to-view-id>
               /secondePage.jsp</to-view-id>
          </navigation-case>
     </navigation-rule>
     <navigation-rule>
          <display-name>
          premierePage</display-name>
          <from-view-id>
          /premierePage.jsp</from-view-id>
          <navigation-case>
               <from-outcome>
               UnVersTrois</from-outcome>
               <to-view-id>
               /troisiemePage.jsp</to-view-id>
          </navigation-case>
     </navigation-rule>
     <navigation-rule>
          <display-name>
          troisiemePage</display-name>
          <from-view-id>
          /troisiemePage.jsp</from-view-id>
          <navigation-case>
               <from-outcome>
               TroisVersDeux</from-outcome>
               <to-view-id>
               /secondePage.jsp</to-view-id>
          </navigation-case>
     </navigation-rule>

</faces-config>
```

© ENI Editions - All rigths reserved

Déclaration des beans managés et de leurs propriétés

Chaque bean managé intervenant dans le cadre d'une application web doit être déclaré dans le fichier de configuration des ressources. La sélection de l'onglet **ManagedBean** de la zone d'édition permet de faciliter cette démarche, par la présentation de formulaires de saisie spécifiques.



L'arborescence située sur la gauche de la zone d'édition propose une répartition des beans managés suivant leur portée au sein de l'application (session, request, application ou none).

Lorsque des beans managés sont déjà disponibles dans cette arborescence, la sélection de l'un d'eux permet de visualiser l'ensemble de ses caractéristiques sur la partie droite de la zone d'édition : il est alors possible de modifier ou préciser le nom du bean lui-même, sa portée, la classe qui lui correspond, ainsi que les valeurs initialement attribuées à ses propriétés.

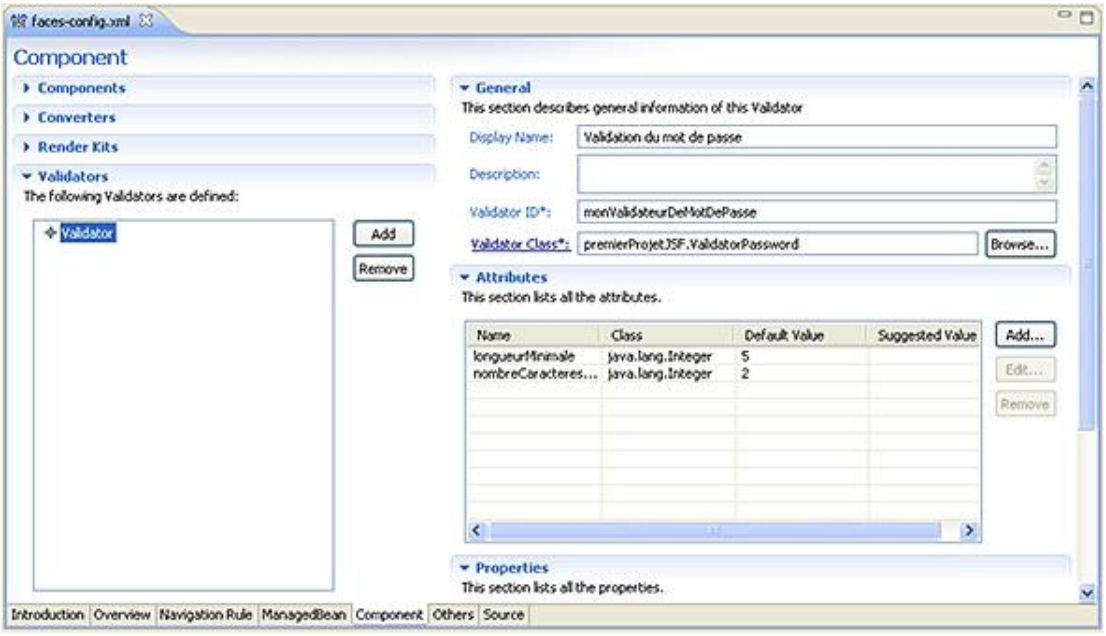
La déclaration d'un nouveau bean managé se fait également très simplement par le biais de cette interface, en précisant le nom du bean, sa classe et sa portée. Il est en général préférable d'avoir préalablement créé la classe correspondant au nouveau bean managé, mais cela n'est cependant pas indispensable : la création de cette classe peut en effet être faite à la volée, au moment même de la déclaration du bean managé.

Naturellement, toute modification apportée dans la déclaration des beans managés est immédiatement répercutée dans le code source de l'application, visible par la sélection de l'onglet **Source** de la zone d'édition.

Déclaration de composants personnalisés

La démarche à suivre est ici identique à celle présentée pour la déclaration des beans managés. La sélection de l'onglet **Component** de la zone d'édition permet d'afficher différents formulaires chargés de faciliter la déclaration des objets personnalisés suivants : composants graphiques, convertisseurs, validateurs et kits de rendus.

© ENI Editions - All rigths reserved



Après avoir sélectionné le type de composant à créer/modifier depuis l'arborescence située à gauche de la zone d'édition, il est possible de choisir l'un des composants déjà déclarés pour en modifier les caractéristiques, ou d'en déclarer un nouveau. Il est important de préciser ici que les classes correspondant aux nouveaux composants doivent impérativement exister préalablement pour que la déclaration puisse être réalisée.

Parmi les caractéristiques qui peuvent ou doivent être mentionnées, on trouve le nom du composant personnalisé, sa description, son identifiant, la classe qui le caractérise, et enfin les attributs et propriétés qui lui sont propres.

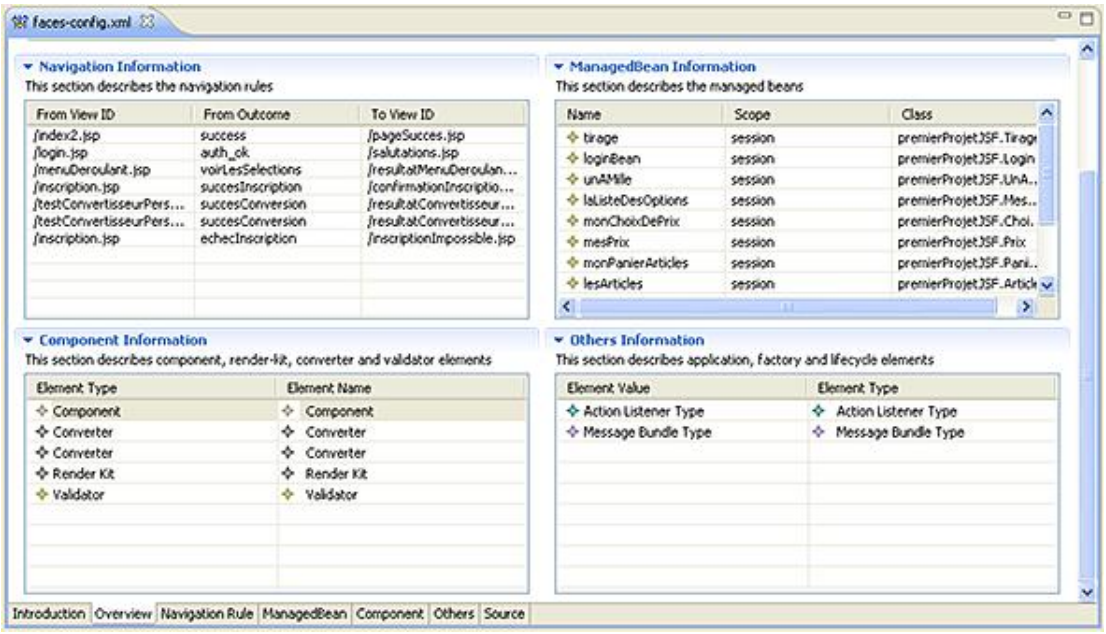
Déclaration des autres types de composants

Cette action peut être entreprise par la sélection de l'onglet **Others** de la zone d'édition. Elle permet d'assurer la déclaration des ressources qui n'ont pas encore été évoquées, à savoir, notamment : les écouteurs et les fichiers de messages personnalisés.

Le mode de déclaration de ces ressources se fait sur le même principe que ce qui a été vu précédemment : sélection du type de ressource à déclarer, puis précision/modification des propriétés qui le caractérisent (par exemple : nom de classe pour les écouteurs ; nom de fichier pour les messages personnalisés).

Aperçu général des ressources déclarées

La répartition de l'affichage du contenu du fichier de configuration des ressources dans différents onglets peut potentiellement rendre sa lecture difficile. Il peut en effet être intéressant de visualiser de manière globale l'ensemble des ressources, en vue d'une impression par exemple. La zone d'édition propose de répondre à ce besoin par la sélection de l'onglet **Overview**, qui offre une vue récapitulative des ressources de l'application.

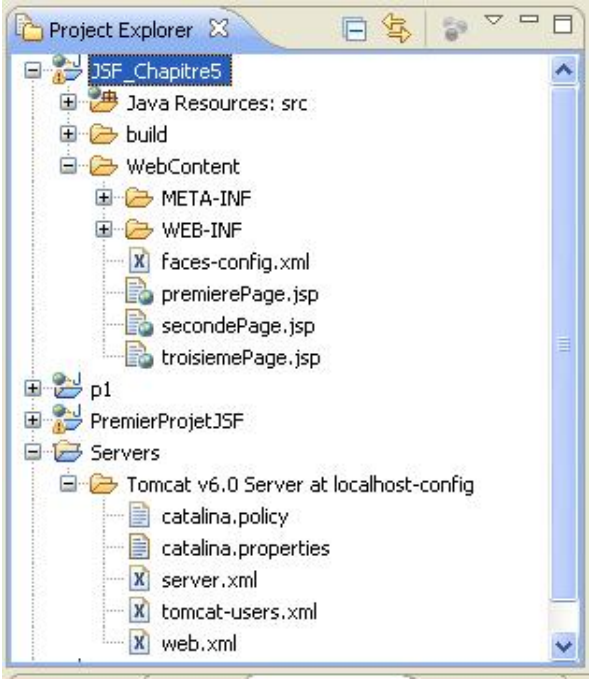


© ENI Editions - All rigths reserved

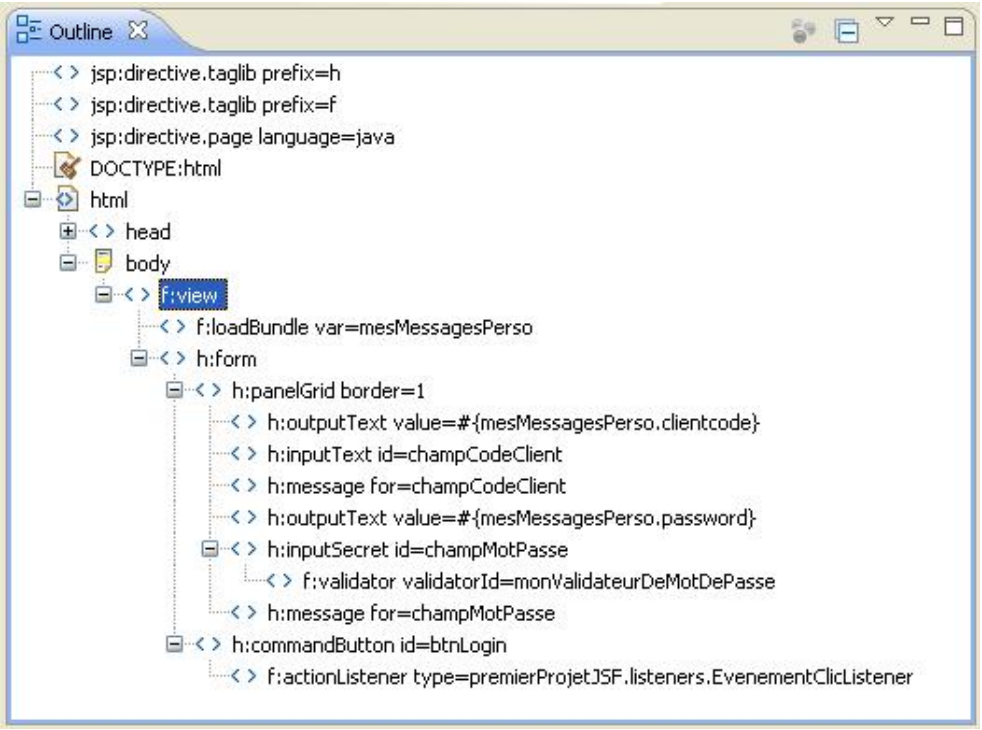
2. La vue d'exploration de projets

Il s'agit d'une vue habituellement rencontrée dans les différents types de projets supportés par l'environnement de développement Eclipse. Elle permet de naviguer au sein des différents projets et d'accéder ainsi facilement aux multiples pages JSP et classes Java utilisées par les applications web en cours de création.

À noter la présence d'un nœud d'arborescence correspondant aux serveurs déclarés et utilisables par l'environnement Eclipse. Son utilisation, qui permet d'accéder rapidement aux différents fichiers de configuration de ces serveurs, se fait conjointement à l'affichage de la fenêtre de propriétés, ainsi qu'avec celui de la vue **Servers** elle-même.



3. La vue Outline



© ENI Editions - All rigths reserved

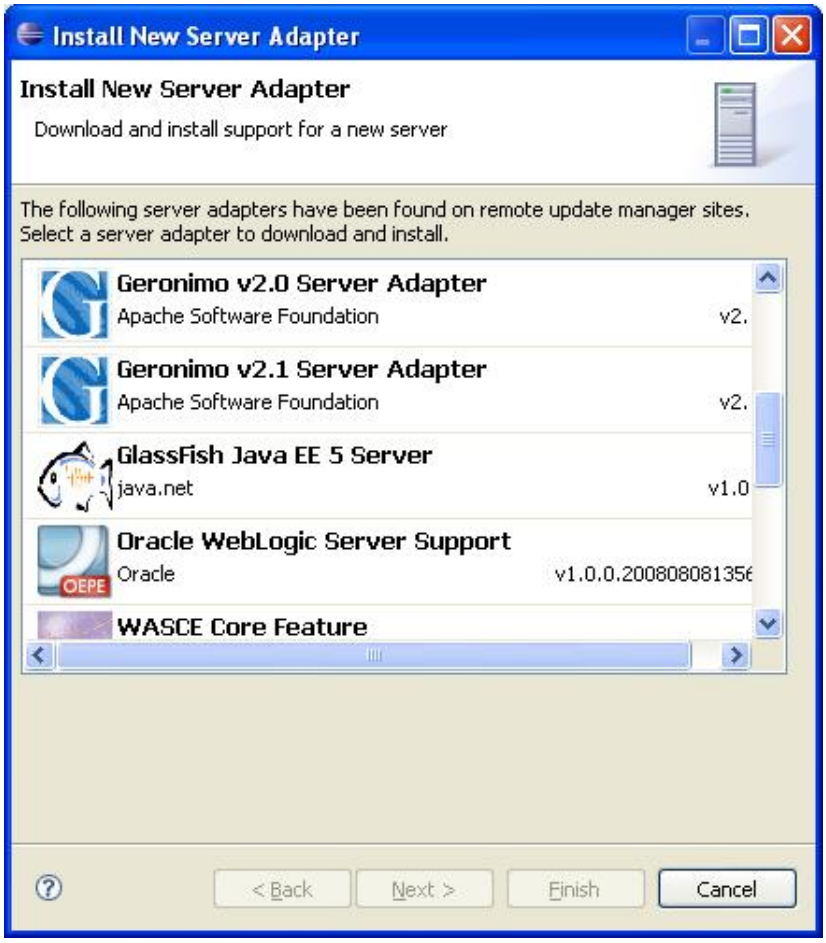
Cette vue offre un aperçu arborescent de la page JSP en cours d'édition. L'ensemble des composants disposés sur la page y sont représentés, qu'il s'agisse de composants HTML purs, de composants graphiques JSF ou de composants qui peuvent leur être associés. Un clic sur l'un des éléments de cette arborescence provoque la sélection du composant correspondant dans la zone d'édition, et l'affichage des propriétés qui le caractérisent dans la fenêtre de propriétés.

4. La vue Servers

L'utilisation de cette vue permet de déclarer différents serveurs et moteurs de servlets, dans le but de les rendre directement exploitables par Eclipse sans qu'il soit nécessaire de passer par des utilitaires annexes.

a. Adaptateurs de serveurs

Outre le classique serveur HTTP de Apache, Eclipse propose la gestion de serveurs spécifiques à J2EE disponibles gratuitement, tels que Tomcat, GlassFish et Sun Application Server. L'exploitation de produits payants est également proposée, notamment celle de IBM WebSphere et Oracle OC4J Server. Il est important de noter que la liste de serveurs gérables par Eclipse n'est pas fermée : elle peut très facilement être complétée par l'installation d'adaptateurs spécifiques recherchés sur le web. L'illustration suivante montre l'utilitaire d'Eclipse chargé de trouver les adaptateurs disponibles. Cet utilitaire s'active depuis la vue **Servers**.



Naturellement, l'installation d'un adaptateur spécifique ne dispense pas de l'installation du serveur lui-même dans le système d'exploitation. Cette étape obligatoire doit être entreprise indépendamment de l'IDE Eclipse.

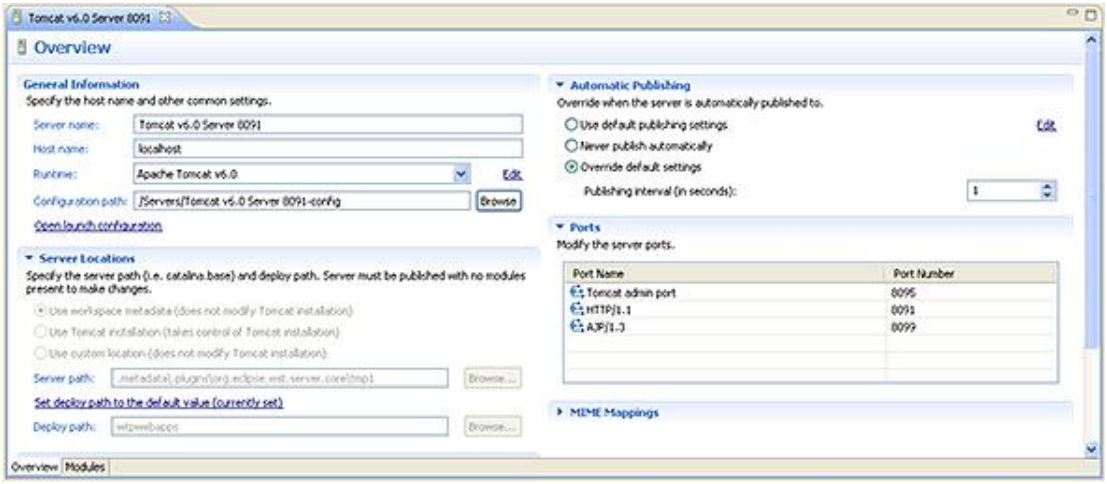
b. Configuration des serveurs

La vue **Servers** est également conçue pour faciliter la configuration des différents serveurs : une fois l'un des serveurs sélectionné, la zone d'édition est mise à jour pour en faire apparaître les multiples paramètres de configuration. Il est notamment possible de mentionner/modifier les ports TCP/IP utilisés par le service de publication et de référencer les différentes applications web prises en charge par le serveur sélectionné.

À titre d'illustration, l'image ci-dessous présente l'apparence de la zone d'édition pour la configuration d'un serveur

© ENI Editions - All rigths reserved

s'appuyant sur le moteur de servlets Apache Tomcat. La présence d'un onglet **Modules** permet de basculer vers l'affichage et la gestion des applications web prises en charge.



Eclipse permet également la gestion de plusieurs serveurs s'appuyant sur le même produit (par exemple, Apache Tomcat). Ces serveurs peuvent alors fonctionner indépendamment les uns des autres, à condition qu'il n'y ait pas de conflits dans les numéros de ports TCP/IP attribués. Ils peuvent exploiter des listes d'applications web distinctes, tout comme disposer d'un certain nombre d'applications communes.

5. La vue Properties

Cette vue est classiquement chargée de présenter la ou les propriétés de l'un des éléments sélectionné dans l'environnement de développement. Par exemple, elle affiche les diverses informations concernant les fichiers ou dossiers exposés dans la vue d'exploration de projets (chemin d'accès complet, date de la dernière modification, taille, etc.).

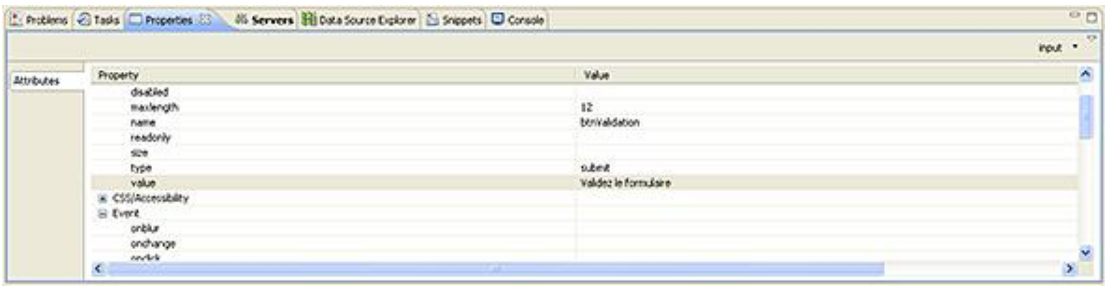
Dans le cadre de la création d'applications web exploitant des composants JSF, cette vue est d'une importance certaine, dans la mesure où elle facilite grandement l'affichage et la mise à jour des propriétés des divers composants positionnés sur les pages.

a. Affichage des propriétés de composants de type HTML ou JSF

Pour ce type de composants, la vue **Properties** présente une unique grille de saisie, dénommée **Attributes**, à partir de laquelle le concepteur de l'application web a la possibilité de préciser rapidement les valeurs qui doivent être attribuées aux différentes propriétés du composant sélectionné.

Par exemple, pour un simple bouton représenté par une balise HTML `<input type='button'>`, la vue **Properties** expose l'intégralité des attributs de balise définis par la norme : le développeur peut donc facilement préciser les valeurs de l'attribut `name`, mentionner le code JavaScript à exécuter lors d'un clic sur le bouton, ou encore définir une image à associer au même bouton. L'image ci-dessous montre le contenu de la vue **Properties** pour ce cas de figure spécifique.

Bien sûr, l'utilisation de cette grille de saisie dispense le concepteur de modifier directement le code source de la page web pour préciser la valeur des attributs.



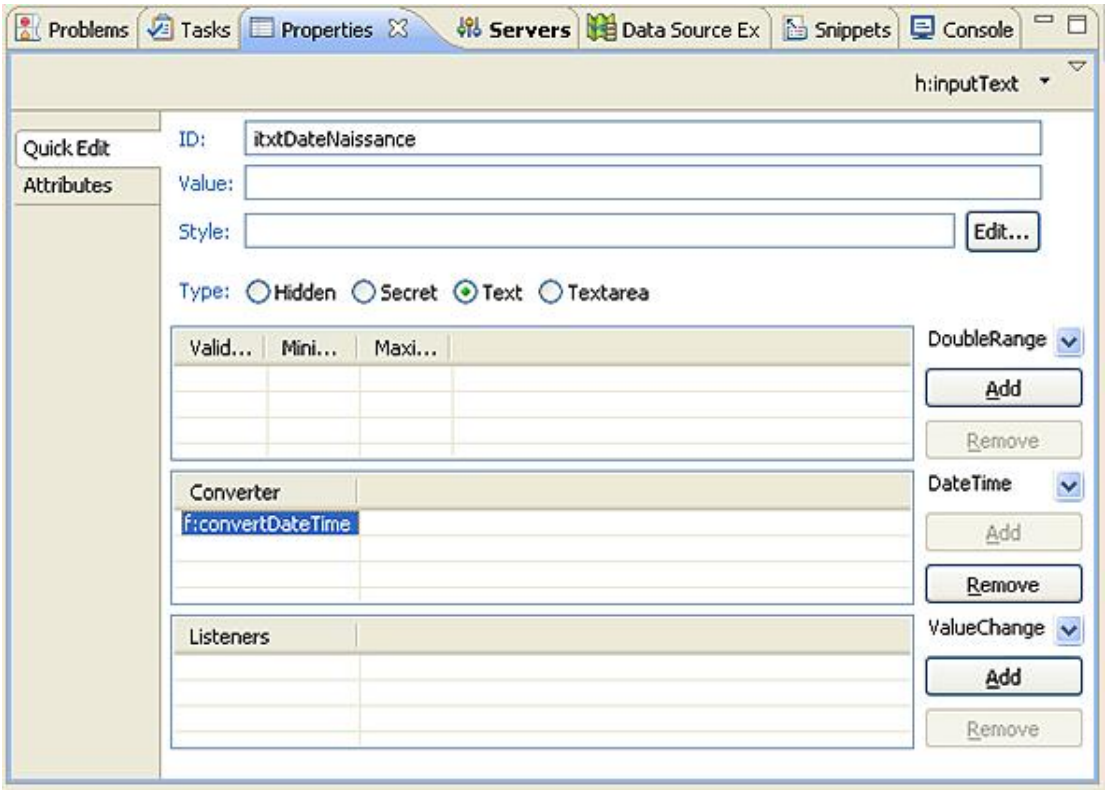
b. Affichage des propriétés de composants issus des catégories JSF Core et JSF HTML

Concernant le traitement des composants JSF issus des catégories JSF Core et JSF HTML de la palette d'édition, la

© ENI Editions - All rigths reserved

vue **Properties** se fait beaucoup plus précise. Outre la présentation de la grille de saisie **Attributes** qui vient d'être évoquée, cette vue propose systématiquement un formulaire de saisie spécifique au type de composant JSF sélectionné dans la zone d'édition. Ce formulaire est accessible par le biais d'un onglet **Quick Edit** situé sur la gauche de la vue **Properties**.

Les informations qui peuvent être fournies grâce à ce formulaire dépendent bien entendu du type de composant JSF : elles peuvent notamment concerner la précision de validateurs, de convertisseurs ou d'écouteurs à associer au composant choisi. L'illustration suivante montre le contenu des informations affichées dans l'onglet **Quick Edit** pour un composant JSF de type `InputText`, auquel est associé un convertisseur de type `ConvertDateTime`.



6. La vue DataSource Explorer

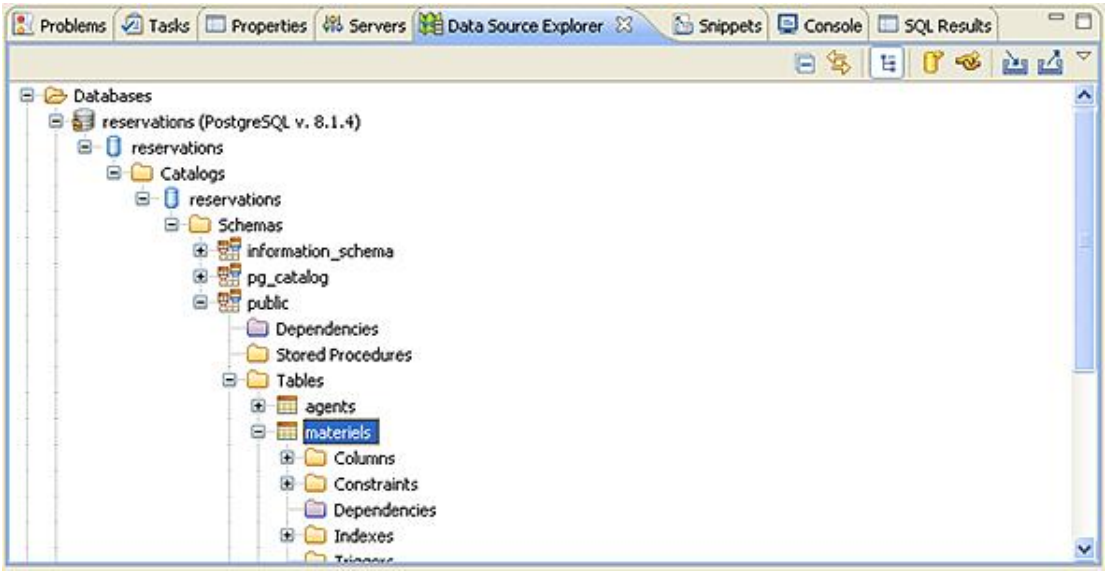
La dernière vue qui peut potentiellement intéresser le concepteur d'applications web basées sur JSF est la vue **Data Source Explorer**. Comme certaines des autres vues qui ont été évoquées précédemment, celle-ci n'est pas spécifique à ce type de projets : elle peut tout à fait être exploitée dans des contextes complètement différents. Néanmoins, compte tenu de l'importance de l'utilisation des bases de données dans les projets web actuels, il paraît nécessaire d'évoquer l'existence de cette vue dans le cadre de cet ouvrage.

a. Accès aux bases de données

Le rôle de la vue **Data Source Explorer** est de fournir au développeur un moyen d'accéder rapidement aux différentes bases de données utilisables par l'application web. Cet outil dispense de passer par des outils annexes, tels que les logiciels clients édités par les fournisseurs de systèmes de gestion de bases de données, et permet l'exploitation d'Eclipse dans toute sa dimension d'environnement de développement intégré.

Les principales actions qui peuvent être entreprises à partir de cette vue sont des opérations de gestion, tels que l'ajout ou la suppression de tables, l'import ou l'export de données. Bien sûr, les données elles-mêmes peuvent être visualisées en sélectionnant la base, puis la table dont on veut connaître le contenu : les données sont alors présentées dans la zone d'édition. Il est également possible de mettre à jour les informations, et même d'ajouter de nouveaux enregistrements.

© ENI Editions - All rigths reserved



b. Accès à d'autres types de sources de données

Outre l'exploitation des bases de données, la vue **Data Source Explorer** permet également de faire appel à d'autres sources de données. En l'occurrence, les fichiers plats (au format CSV, par exemple) peuvent être pris en charge, de même que les fichiers XML et les sources de données accessibles par Web Services.

© ENI Editions - All rigths reserved

Ajout de composants graphiques à une page

Comme cela a été mentionné dans le paragraphe précédent, l'ajout de composants graphiques JSF dans les pages web se fait très simplement, par utilisation du glissé-déposé, depuis la palette d'objets vers la zone d'édition.

Au moment de l'intégration d'un composant dans une page, le code source de celle-ci est mis à jour pour faire apparaître une balise spécifique au type de composant choisi : c'est ce que l'on appelle une balise de restitution. Il s'agit, selon le cas, d'une balise appartenant à la bibliothèque HTML ou d'une balise référencée dans la bibliothèque Core. À titre d'illustration, l'intégration d'un composant de type `UICommand` peut provoquer l'ajout de la balise suivante dans le code source de la page web :

```
<h:commandButton style="color: #FF0000" action="suite"
value="Cliquez ici pour valider le formulaire" id="idValidation">
</h:commandButton>
```

Dans ce cas particulier, le composant `UICommand` a été restitué par une balise `<h:commandButton>` de la bibliothèque HTML, dont l'apparence graphique correspond à celle d'un classique bouton de soumission HTML. Cette balise présente plusieurs attributs qui, en général, servent à préciser l'apparence graphique du composant (couleurs, styles...) ou à indiquer les actions qui peuvent être engagées lors de l'utilisation de ce composant (par exemple, appel d'une fonction JavaScript au moment du survol par la souris ; utilisation d'un cas de navigation particulier au moment du clic).

D'une manière plus générale, il est important de noter que la plupart des balises correspondant aux composants graphiques JSF partagent un jeu d'attributs communs. C'est le cas notamment de l'attribut `id`, qui est disponible pour la totalité des composants JSF. Sa présence est d'ailleurs mentionnée dans l'exemple ci-dessus, concernant le bouton de soumission. Mais ces balises proposent également un jeu d'attributs spécifiques, qui diffère suivant le type de composant graphique représenté. Par exemple, la balise correspondant à un bouton de soumission expose un attribut `action` qui lui est propre : cet attribut ne peut pas être utilisé pour d'autres composants (champ de saisie, par exemple).

Ce paragraphe a pour objectif de présenter les différentes balises de restitution de composants JSF, en précisant notamment le rôle de leurs principaux attributs communs, et de montrer comment se réalise l'intégration de ces balises dans les pages web.

Seuls sont traités ici les composants JSF restitués dans la page par des balises issues de la bibliothèque HTML. Il s'agit donc uniquement des objets se trouvant dans la catégorie JSF HTML de la palette. Des informations concernant les autres types de composants (convertisseurs, écouteurs et validateurs) sont fournies dans les paragraphes suivants de ce chapitre.

1. Classes et balises de restitution

Ce point fait écho à certaines informations déjà données dans le chapitre Composants graphiques personnalisés de cet ouvrage. Il rappelle la liste des classes correspondant à des composants graphiques JSF, indique pour chacun d'eux la balise de restitution à utiliser en vue d'une présentation dans une page web, et rappelle l'apparence visuelle du composant.

Le tableau ci-dessous présente l'ensemble de ces informations, et mentionne, pour chaque composant, le nom de l'objet correspondant tel qu'il est présenté dans la palette Eclipse. Les composants sont classés ici en catégories, compte tenu de leur rôle respectif.

Classe	Balise de restitution	Apparence visuelle	Nom de l'objet correspondant dans la palette (catégorie JSF HTML)
Catégorie : Composants de type "Texte"			
HtmlInputText	<h:inputText>	Zone de texte sur une seule ligne	Text Input
HtmlInputSecret	<h:inputSecret>	Zone de mot de passe	Secret Input
HtmlInputHidden	<h:inputHidden>	Champ caché	Hidden Input
HtmlInputTextArea	<h:inputTextarea>	Zone de texte multilignes	Textarea Input

© ENI Editions - All rigths reserved

HtmlOutputLabel	<h:outputLabel>	Étiquette associée à un autre composant	Output Label
HtmlOutputFormat	<h:outputFormat>	Affichage d'un ensemble de messages selon un format spécifique	Output Format
HtmlOutputLink	<h:outputLink>	Hyperlien dont l'utilisation ne provoque pas d'événement de type ActionEvent (contrairement au composant de type HtmlCommandLink)	Non représenté
HtmlOutputText	<h:outputText>	Texte simple. Peut être associé à d'autres composants, tels que HtmlCommandLink.	Output Text
HtmlSelectBooleanCheckbox	<h:selectBooleanCheckbox>	Case à cocher	Select Boolean Checkbox
HtmlSelectManyCheckbox	<h:selectManyCheckbox>	Ensemble de cases à cocher	Select Many Checkbox
HtmlSelectOneMenu	<h:selectOneMenu>	Zone de liste déroulante à choix unique	Select One Menu
HtmlSelectManyMenu	<h:selectManyMenu>	Zone de liste à choix multiple (un seul élément visible)	Select Many Menu
HtmlSelectOneListBox	<h:selectOneListbox>	Zone de liste à choix unique. Plusieurs éléments visibles.	Select One Listbox
HtmlSelectManyListBox	<h:selectManyListbox>	Zone de liste à choix multiples. Plusieurs éléments visibles.	Select Many Listbox
HtmlSelectOneRadio	<h:selectOneRadio>	Bouton radio	Select One Radio
Catégorie : composants de type "Commande"			
HtmlCommandButton	<h:commandButton>	Bouton	Command Button
HtmlCommandLink	<h:commandLink>	Hyperlien dont l'utilisation provoque la génération d'un événement de type ActionEvent.	Command Link
Catégorie : composants de type "Tableau de données"			
HtmlDataTable	<h:dataTable>	Tableau de données de N lignes et M colonnes	Data Table
HtmlColumn	<h:column>	Colonne de tableau de données.	Column
Catégorie : composants de type "Présentation"			

© ENI Editions - All rigths reserved

HtmlPanelGrid	<h:panelGrid>	Grille de présentation de N lignes et M colonnes	Panel Grid
HtmlPanelGroup	<h:panelGroup>	Conteneur de composants	Panel Group
Catégorie : composants de type "Messages"			
HtmlMessage	<h:message>	Texte affichant les messages d'erreur générés par l'utilisation de l'un des composants de la page.	Message
HtmlMessages	<h:messages>	Texte affichant tous les messages.	Messages
Catégorie : composant de type "Images"			
HtmlGraphicImage	<h:graphicImage>	Image	Graphic Image
Catégorie : composant de type "Formulaire"			
HtmlForm	<h:form>	Formulaire de saisie.	Form

2. Principaux attributs de balises communs

La connaissance du rôle de chaque attribut est indispensable pour que le concepteur d'une l'application web puisse utiliser efficacement les composants JSF dont il dispose.

Il est possible de distinguer deux groupes d'attributs communs : le premier concerne l'ensemble des attributs utilisables par la totalité des balises de restitution. Le second se limite aux attributs exclusivement applicables aux balises correspondant à des composants destinés à la saisie d'information (zones de texte, cases à cocher, etc.).

a. Attributs communs à l'ensemble des balises de restitution

Les attributs communs les plus couramment utilisés sont `binding`, `id`, `immediate`, `rendered`, `style` et `styleClass`.

L'attribut binding

Il permet de lier l'instance du composant représenté par la balise à une propriété de bean managé. Cela permet notamment de modifier par programmation les attributs du composant en question.

Le code source ci-dessous correspond à celui d'un JavaBean extrêmement simple : il ne possède qu'une seule propriété de type `UIInput`, dénommée `champEntree`, qui lui permet de stocker la référence d'un objet de ce type.

```
package premierProjetJSF;
import javax.faces.component.UIInput;
public class Liaison {
     UIInput champEntree;

     public UIInput getChampEntree() {
          return champEntree;
     }

     public void setChampEntree(UIInput champEntree) {
          this.champEntree = champEntree;
     }
}
```

Un tel JavaBean peut être exploité en tant que bean managé dans une application web JSF, à condition de le déclarer dans le fichier de configuration des ressources de l'application. Les lignes suivantes correspondent à une

© ENI Editions - All rigths reserved

telle déclaration : le bean est dénommé `liaison` ; il est de type `premierProjetJSF.Liaison` et dispose de la portée session dans l'application web :

```
<managed-bean>
<managed-bean-name>liaison</managed-bean-name>
<managed-bean-class>premierProjetJSF.Liaison</managed-bean-class>
<managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

Toutes les conditions sont donc réunies pour réaliser une liaison entre une instance d'un composant graphique de l'application et la propriété `champEntree` du bean `liaison`. Naturellement, le type du composant à lier ne peut pas être choisi librement : la propriété `champEntree` étant de type `UIInput`, seuls les composants de ce type (restitués par une balise `<h:inputText>`) sont éligibles. Voici un exemple de contenu de page JSP dans lequel un composant `UIInput` est lié à la propriété `champEntree` du bean `liaison`.

```
<f:view><h:form>
<h:inputText binding="#{liaison.champEntree}"></h:inputText>
</h:form></f:view>
```

Par ce mécanisme, le JavaBean dispose de la référence au champ de saisie : il peut donc en modifier les attributs si nécessaire.

L'attribut id

Cet attribut permet de référencer le composant qui l'utilise au sein de l'application web. Il ne s'agit pas d'un caractère obligatoire, mais son absence empêche la manipulation du composant concerné par les autres composants de la page, ou par d'autres classes.

L'exemple suivant correspond à la présence d'un composant de type `UIMessage` sur une page JSP. Ce composant est restitué sous la forme d'une balise `<h:message>`, dont l'attribut `id` prend ici la valeur `messageMotPasse` :

```
<h:message for="champMotPasse" id="messageMotPasse">
</h:message>
```

L'attribut rendered

Cet attribut de balise permet de préciser si le composant doit être présenté à l'internaute ou s'il doit être caché. La valeur de l'attribut `rendered` est bien entendu un booléen, dont la valeur peut être mentionnée directement ou par l'intermédiaire d'une expression booléenne. Il est également possible de préciser cette valeur grâce à l'emploi du langage d'expression spécifique à JSF, dont l'évaluation renvoie un booléen.

```
<h:outputText rendered="#{beanInscription.nom!='test'}"
value="texte affiché">
</h:outputText>
```

Dans l'exemple ci-dessus, l'affichage d'un champ de sortie de type `UIOutput` est contrôlé grâce à l'attribut `rendered`, dont la valeur correspond à l'évaluation d'une expression JSF. Le test assuré par cette expression consiste à vérifier si la propriété `nom` d'un bean managé dénommé `beanInscription` possède une valeur différente de la chaîne test.

Les attributs style et styleClass

Ces deux attributs de balise permettent de spécifier le style à appliquer au composant graphique. L'attribut `style` accepte une valeur correspondant à la définition complète du style à utiliser (précision d'une couleur, d'un fond, d'une police, etc.).

```
<h:outputText
     binding="#{liaison.champLabel}"
     id="txtLabel1"
     value="Saisissez la quantité souhaitée (entre 1 et 10):"
     rendered="true"
     style='color: #0000FF; font-style: italic; font-size: 12px;
font-family: "Comic Sans MS", Sans-Serif; font-weight: bold'
     >
</h:outputText>
```

Dans cet exemple, le champ de type `UIOutput` dénommé `txtLabel1` permet d'afficher un texte italique de couleur bleue, de taille égale à douze pixels. La police utilisée pour ce texte est Comic Sans MS.

© ENI Editions - All rigths reserved

b. Attributs communs aux seules balises de restitution correspondant à des champs de saisie

Dans la spécification Java Server Faces, les composants graphiques destinés à la saisie sont instanciés à partir de la classe `UIInput`, ou de ses classes dérivées. Il existe actuellement onze classes dérivées de `UIInput`, auxquelles sont associées des balises de restitution HTML. Outre les attributs communs à tous les types de balises de restitution, les balises restituant des composants de saisie dans les pages web partagent spécifiquement un ensemble d'attributs communs dont voici une description :

Attribut converter

Cet attribut permet de spécifier le convertisseur associé au champ de saisie. Pour rappel, un convertisseur est un objet chargé de s'assurer que la donnée saisie dans le composant graphique est compatible avec la valeur attendue pour la propriété du bean à laquelle est lié le composant.

Supposons par exemple que le fichier de configuration des ressources d'une application web dispose de la référence à un convertisseur personnalisé, par l'intermédiaire des lignes de code suivantes :

```
<converter>
          <display-name>
          convertisseurPerso</display-name>
          <converter-id>
          convertisseurPersoID</converter-id>
          <converter-class>
          premierProjetJSF.MonConvertisseur</converter-class>
     </converter>
```

Le convertisseur en question se nomme convertisseurPerso. Il est issu d'une classe `premierProjetJSF.MonConvertisseur` implémentant l'interface `Converter` et peut être utilisé par l'ensemble des pages web de l'application JSF grâce à son identifiant `convertisseurPersoID`.

Ainsi, ce convertisseur peut, par exemple, être associé à un champ de saisie de type `HtmlInputText`. Pour cela, l'identifiant du convertisseur est utilisé comme valeur de l'attribut `converter` de la balise `<h:inputText>` représentant le composant.

```
<h:inputText id="txtPrenom" converter="convertisseurPersoID"
rendered="true" required="true">
</h:inputText>
```

Attribut converterMessage

Il indique le contenu du message à présenter à l'internaute lorsque la conversion de la donnée saisie échoue.

Dans l'exemple suivant, un champ de type `HtmlInputText` dispose d'un convertisseur standard issu de la classe `DateTimeConverter`. Ce convertisseur est utilisé pour vérifier que la date mentionnée dans le champ de saisie respecte bien un format particulier (par exemple : *dd/MM/yyyy*). Ce format est indiqué par l'attribut `pattern` de la balise `<f:convertDateTime>` représentant le convertisseur. En cas de non-respect du format de date attendu, le message mentionné par l'attribut `converterMessage` de la balise correspondant au champ de saisie est généré : celui-ci peut éventuellement être présenté à l'internaute, si la page prévoit une association entre le champ de saisie et un objet de type `UIMessage`. Ce qui est le cas ici.

```
<h:inputText id="itxtDateNaissance"
     converterMessage="La date indiquée doit être au format
JJ/MM/AAAA"
     <f:convertDateTime pattern="dd/MM/yyyy" />
</h:inputText>
<h:message id="msgDateNaissance" for="itxtDateNaissance"
     style="color: #FF0000; background-color: #FFFF00">
</h:message>
```

Attribut dir

Cet attribut sert à préciser le sens dans lequel doit être affiché le composant de saisie. Il peut s'agir d'un défilement de la droite vers la gauche (la valeur de l'attribut `dir` est alors égale à `rtl`, pour right-to-left), ou d'un défilement de la gauche vers la droite (la valeur de l'attribut sera alors égale à `ltr`, pour left-to-right).

Dans l'exemple suivant, la saisie réalisée dans le champ `txtCommentaire` se fait de droite à gauche.

```
<h:inputText id="txtCommentaire" dir="rtl">
</h:inputText>
```

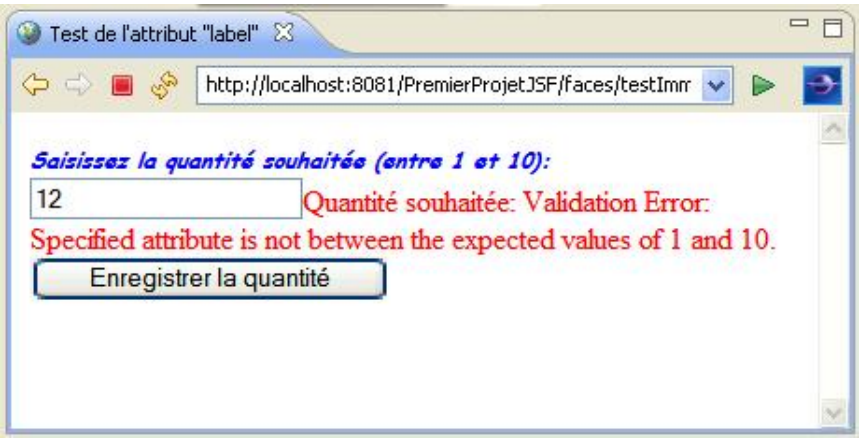
© ENI Editions - All rigths reserved

Attribut label

Cet attribut permet d'attribuer une étiquette au champ de saisie. Cette étiquette n'est pas visible en tant que telle dans le formulaire, mais elle est utilisée lorsque des messages d'erreur concernant le champ de saisie sont générés. Cela permet, en particulier lorsqu'une page web présente un formulaire comportant de nombreux champs de saisie, d'identifier rapidement et précisément les champs à l'origine d'une erreur.

Dans le code source suivant, une zone de texte `txtQuantite` permet à l'utilisateur de saisir une valeur numérique. Cette zone est associée à un validateur, chargé de vérifier systématiquement que la valeur mentionnée est bien comprise entre 1 et 10. En cas d'erreur, un message adapté est présenté à l'utilisateur par l'intermédiaire de l'objet `UIMessage` associé à la zone de saisie. Ce message fait spécifiquement mention de l'étiquette correspondant au champ à l'origine du problème, car la balise `<h:inputText>` qui le représente comporte un attribut `label` renseigné.

```
<f:view><h:form>
<h:outputText binding="#{liaison.champLabel}" id="txtLabel1"
value="Saisissez la quantité souhaitée (entre 1 et 10):"
rendered="true" ></h:outputText>
<br>
<h:inputText label="Quantité souhaitée" id="txtQuantite"
   immediate="false">
     <f:validateDoubleRange minimum="1" maximum="10">
     </f:validateDoubleRange>
</h:inputText>
<h:message for="txtQuantite" style="color: #FF0000">
</h:message>
<br>
<h:commandButton id="btnValidation" value="Enregistrer la
quantité" immediate="true">
</h:commandButton>
</h:form></f:view>
```



Attribut required

Cet attribut permet de spécifier que la saisie est obligatoire dans un champ de saisie particulier. La valeur de cet attribut est bien entendu un booléen.

L'exemple suivant correspond à l'utilisation d'un champ de saisie `txtCommentaire`, dans lequel l'utilisateur est forcé de fournir une information : l'attribut `required` de la balise `<h:inputText>` est en effet positionné sur la valeur `true`. En cas de non-respect de cette contrainte, un message adapté est renvoyé à l'internaute par le biais de l'objet `UIMessage` associé au champ de saisie.

```
<f:view>
     <h:form>
          <h:outputText id="txtIntitule" value="Saisissez
impérativement un commentaire:"></h:outputText>
          <br>
          <h:inputText id="txtCommentaire"
label="Commentaire impératif"
required="true">
          </h:inputText>
          <h:message id="messageCommentaire"
                    for="txtCommentaire"
```


© ENI Editions - All rigths reserved

```
style="color: #FF0000; font-weight: bold">
          </h:message>
          <br>
          <h:commandButton id="btnValidation"
value="Validation du formulaire">
          </h:commandButton>
     </h:form>
</f:view>
</html>
```

Attribut requiredMessage

Lorsque la saisie dans un champ est rendue obligatoire par la présence de l'attribut de balise `required`, un message par défaut est renvoyé à l'utilisateur en cas de non respect de la contrainte. Il est probable que ce message par défaut ne convienne pas dans certaines situations, pour une raison quelconque (par exemple : l'information délivrée est trop précise (ou au contraire, pas assez) ; la langue utilisée pour le message n'est pas la même que celle utilisée dans le reste de la page web).

Il est alors possible d'empêcher l'affichage du message proposé par défaut, grâce à l'utilisation de l'attribut de balise `requiredMessage`. La valeur fournie pour cet attribut sert alors de message d'erreur affiché lorsque l'internaute oublie de fournir une information dans un champ requis.

L'exemple proposé précédemment pour l'attribut `required` peut être adapté en rajoutant un attribut `requiredMessage` à la balise `<h:inputText>` correspondant au champ de saisie :

```
<h:inputText
     id="txtCommentaire"
     label="Commentaire impératif"
     required="true"
     requiredMessage="Vous devez obligatoirement saisir une
information dans ce champ!">
```

Attribut validator

Cet attribut sert à identifier une méthode spécifique d'un JavaBean, chargée de faire toutes les vérifications requises par rapport à la valeur saisie dans un champ particulier.

La méthode en question doit impérativement être de type void et accepter les trois arguments suivants :

- Un argument de type `FacesContext` permettant au JavaBean de détenir une référence au contexte de l'application web.
- Un argument de type `UIComponent`, correspondant au composant JSF à l'origine de la demande de validation.
- Un argument de type `Object`, représentant la valeur du composant JSF à l'origine de la demande de validation.

À titre d'exemple, considérons une page web dans laquelle se trouve un champ de saisie de type `HtmlInputText`. On peut imaginer la mise en place d'une validation triviale, destinée à s'assurer que le texte saisi dans le champ contienne exactement deux occurrences de la lettre *A*. Pour répondre à cette attente, il est possible d'entreprendre successivement les étapes suivantes :

- Création d'un JavaBean contenant une méthode spécifiquement chargée de la validation.
- Déclaration du JavaBean dans le fichier de configuration des ressources.
- Association entre le composant graphique dont on veut tester la valeur, et la méthode de validation.

Conformément aux contraintes évoquées plus haut concernant la signature que doit respecter la méthode de validation, voici le code source d'un JavaBean capable de déterminer le nombre d'occurrences de la lettre *A* dans la chaîne représentant la valeur d'un composant graphique (quel que soit son type).

```
package premierProjetJSF;

import javax.faces.application.FacesMessage;
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
```

© ENI Editions - All rigths reserved

```
import javax.faces.validator.ValidatorException;

public class BeanAttributValidator {

     private int nombreOccurences(char carac,String chaine)
     {
          //Retourne le nombre d'occurences du caractère
          //"carac" dans la chaîne passée en second argument.
          int nbOccurences=0;
          for (int i=0;i<chaine.length();i++)
          {
               if (chaine.charAt(i)==carac) nbOccurences++;
          }

          return nbOccurences;
     }


     public void checkName(FacesContext arg0, UIComponent arg1,
Object arg2)
     throws ValidatorException {

System.out.println("Composant demandant la validation:
"+arg1.getId());
System.out.println("Valeur saisie dans le composant:
"+arg2.toString());

String valeur=arg2.toString().toUpperCase();
int nombreDeA_requis=2;
int nombreDeA=nombreOccurences('A', valeur);

if (nombreDeA<nombreDeA_requis) throw new ValidatorException(new
FacesMessage("Il manque "+(nombreDeA_requis-nombreDeA)+" caractère(s)
'A' dans la chaîne saisie!"));
if (nombreDeA>nombreDeA_requis)
throw new ValidatorException(new FacesMessage("Il y a
"+(nombreDeA-nombreDeA_requis)+" caractère(s) 'A' en trop dans
la chaîne saisie!"));
}
}
```

Le code de ce JavaBean montre que la méthode pouvant faire office de méthode de validation se nomme `checkName` : elle présente effectivement les trois arguments requis pour assurer une validation. Le contenu de cette méthode indique que si le nombre d'occurrences de la lettre *A* est incorrect, alors une exception de type `ValidatorException` est générée. La création de cette exception se fait obligatoirement en fournissant un message décrivant la cause de l'erreur : ce message est un objet de type `FacesMessage`.

Pour que ce JavaBean puisse être exploité dans les pages de l'application web, son existence doit être déclarée au sein du fichier de configuration des ressources de l'application. Les lignes suivantes peuvent correspondre à une telle déclaration :

```
<managed-bean>
          <managed-bean-name>
          beanAttributValidator
</managed-bean-name>
          <managed-bean-class>
          premierProjetJSF.BeanAttributValidator
</managed-bean-class>
          <managed-bean-scope>
          session
</managed-bean-scope>
</managed-bean>
```

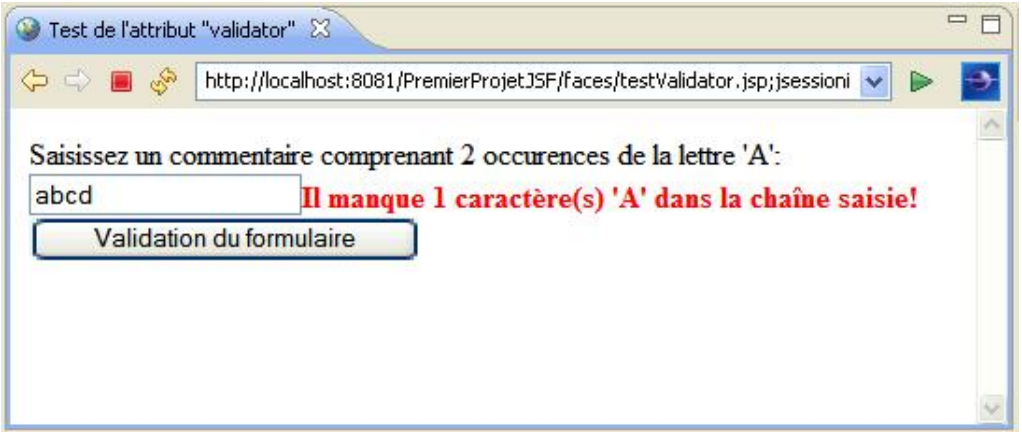
Par ces modifications dans le fichier **faces-config.xml**, toutes les pages web l'application peuvent exploiter le bean de session nommé `beanAttributValidator`, instance de la classe `BeanAttributValidator`, et exploiter sa méthode `checkName` pour réaliser une validation.

Voici pour finir le contenu d'une page JSP contenant un champ de saisie de type `HtmlInputText`. Ce champ, représenté par la balise `<h:inputText>`, est associé à la méthode `checkName` de `beanAttributValidator`, grâce à l'utilisation de l'attribut `validator`.

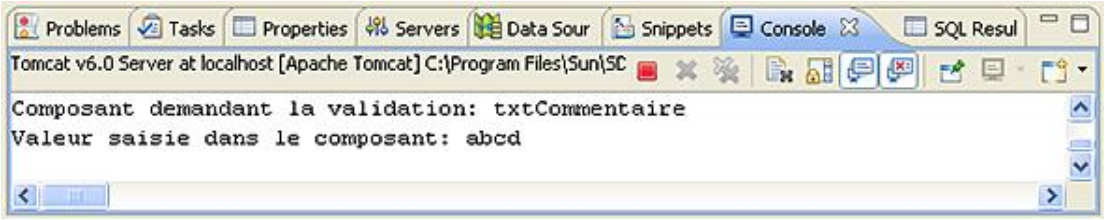
© ENI Editions - All rigths reserved

```
<f:view>
     <h:form>
          <h:outputText id="txtIntitule" value="Saisissez un
commentaire comprenant 2 occurences de la lettre 'A':">
</h:outputText>
          <br>
<h:inputText id="txtCommentaire"
     validator="#{beanAttributValidator.checkName}"
     label="Commentaire"
     required="true"
     requiredMessage="Vous devez obligatoirement saisir une
information dans ce champ!">

          </h:inputText>
          <h:message id="messageCommentaire"
     for="txtCommentaire"
style="color: #FF0000; font-weight: bold">
     </h:message>
     <br>
          <h:commandButton id="btnValidation"
value="Validation du formulaire">
</h:commandButton>
     </h:form>
</f:view>
</html>
```

L'exécution de cette page JSP montre bien la prise en charge correcte de la validation attendue :



De plus, conformément au contenu de la méthode `checkName` de la classe `BeanAttributValidator`, l'identifiant et la valeur du composant pour lequel la validation est demandée sont présentés dans la vue **Console** d'Eclipse :



Attribut validatorMessage

Cet attribut joue un rôle similaire à celui de l'attribut `requiredMessage`, mais il concerne le processus de validation. Il permet au concepteur de pages web de préciser le message qui doit être renvoyé à l'internaute lors de l'échec d'une validation. Ce message vient en remplacement de celui qui est généré par le composant ou la méthode chargé d'assurer la validation.

Dans l'exemple qui vient d'être évoqué à propos de l'attribut `validator`, le message renvoyé par la méthode `checkName` peut être ignoré par l'affectation d'un attribut `validatorMessage` à la balise représentant le champ de saisie dans la page JSP :

```
<h:inputText
```

© ENI Editions - All rigths reserved

```
     id="txtCommentaire"
     validatorMessage="Le commentaire ne respecte pas la contrainte énoncée!"
     validator="#{beanAttributValidator.checkName}"
     label="Commentaire"
     required="true"
     requiredMessage="Vous devez obligatoirement saisir une
information dans ce champ!">
```

Attribut valueChangeListener

Cet attribut permet de mentionner le nom d'une méthode de JavaBean qui doit être exécutée dès qu'un changement intervient au niveau de la valeur d'un composant.

De façon similaire à ce qui a été vu concernant l'attribut `validator`, une méthode capable de prendre en charge ce type d'événement doit respecter une signature particulière. En l'occurrence, la méthode en question doit :

- Être de type `void`.
- Présenter un argument de type `ValueChangeEvent`.

Le concepteur de l'application web doit développer le JavaBean contenant la méthode évoquée, déclarer ce JavaBean dans le fichier de configuration des ressources de l'application et faire appel à la méthode depuis un composant graphique positionné dans une page JSP. Cet appel de méthode se fait par le biais de l'attribut `valueChangeListener` de la balise qui représente le composant.

À titre d'illustration, voici le code source d'un JavaBean comportant une méthode `modificationContenu` pouvant faire office d'écouteur pour les événements de type `ValueChangeEvent`. Lorsqu'un tel événement survient, la méthode en question récupère la référence du composant qui en est la source, puis incrémente un compteur permettant de connaître le nombre de fois que l'événement est survenu. La valeur de ce compteur est enfin affichée dans la console.

```
package premierProjetJSF;

import javax.faces.component.UIComponent;
import javax.faces.event.ValueChangeEvent;

public class BeanAttributValueChangeListener {
     private int compteur;

     public BeanAttributValueChangeListener()
     {
          compteur=0;
     }
     public void modificationContenu(ValueChangeEvent evenement)
     {
          UIComponent composant=evenement.getComponent();
          compteur++;
          System.out.println("Déjà "+compteur+" modification(s)
de la valeur du composant "+composant.getId());
     }
}
```

Ce JavaBean doit ensuite être déclaré dans le fichier de configuration des ressources pour pouvoir être exploité par le reste de l'application web. Les lignes suivantes montrent la déclaration d'un bean de session dénommé `beanAttributValueChangeListener` basé sur la classe `BeanAttributValueChangeListener`.

```
<managed-bean>
          <managed-bean-name>
          beanAttributValueChangeListener</managed-bean-name>
          <managed-bean-class>
          premierProjetJSF.BeanAttributValueChangeListener
</managed-bean-class>
          <managed-bean-scope>
          session</managed-bean-scope>
     </managed-bean>
```

La capture des événements de type `ValueChangeEvent` par la méthode `modificationContenu` peut désormais être assurée. Un composant graphique situé sur une page JSF peut donc faire l'objet de cette capture : la balise qui le

© ENI Editions - All rigths reserved

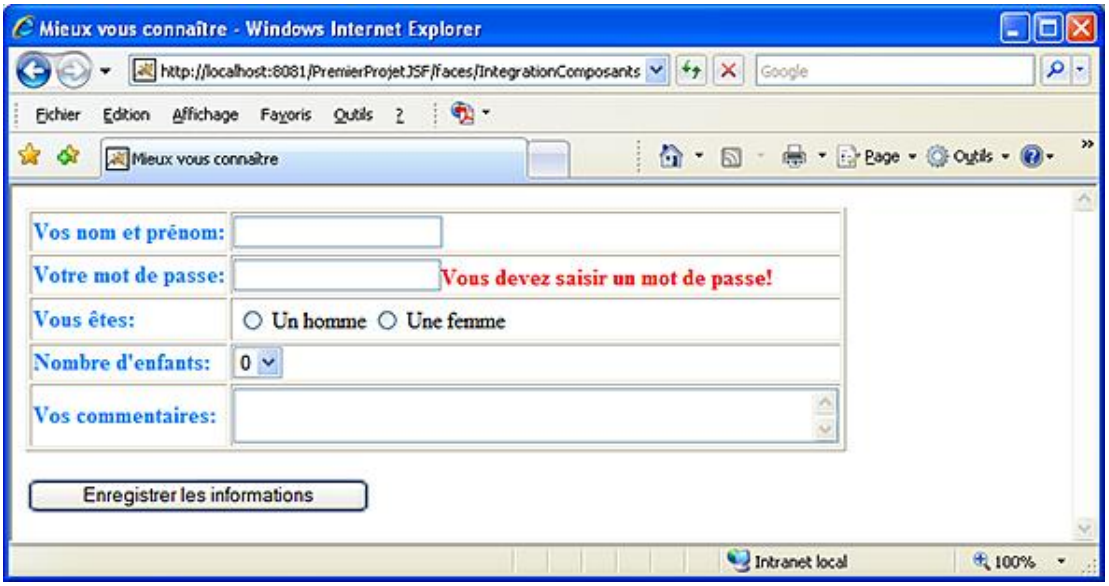
représente doit contenir un attribut valueChangeListener dont la valeur vaut #{beanAttributValueChangeListener.modificationContenu}. Le code suivant en est une illustration possible :

```
<h:inputText id="txtCommentaire"
     valueChangeListener = "#{beanAttributValueChangeListener.
modificationContenu}"
     label="Commentaire"
     required="true"
     requiredMessage="Vous devez obligatoirement saisir une information
dans ce champ!" >
```

3. Intégration des composants graphiques dans les pages JSP

Après avoir abordé les différents attributs partagés par l'ensemble des balises de restitution, et ceux partagés uniquement entre les balises de restitution des composants de saisie, voyons maintenant comment intégrer concrètement les principaux composants graphiques dans les pages JSP.

Pour cela, nous pouvons nous appuyer sur la mise en place d'un formulaire web comprenant les composants graphiques les plus couramment utilisés, et détailler pas à pas les étapes de réalisation. La figure suivante illustre le résultat à obtenir :



a. Intégration du composant correspondant au formulaire de saisie

Après avoir créé la page JSP dans laquelle les différents composants seront positionnés, il faut immédiatement placer la balise de restitution correspondant au formulaire de saisie. Les balises des autres composants graphiques seront par la suite obligatoirement encadrées par la balise du formulaire.

Par un simple glissé-déposé depuis la palette, l'objet Form est positionné dans la zone d'édition. On constate que l'environnement a automatiquement inséré le formulaire au sein d'un couple de balises <f:view> </f:view>. Au besoin, depuis la vue **Properties**, il est possible de préciser certains attributs du formulaire.

```
<%@taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<%@taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
<%@ page language="java"
contentType="text/html; charset=ISO-8859-1"
              pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Mieux vous connaître</title>
</head>
<body>
```

© ENI Editions - All rigths reserved

```
<f:view>
     <h:form></h:form>
</f:view></body>
</html>
```

b. Intégration d'une grille de présentation

Pour que la présentation du formulaire de saisie soit assez esthétique, il est nécessaire de répartir les différents composants graphiques dans un conteneur tabulaire. En l'occurrence, compte tenu de l'apparence qui doit être donnée à ce formulaire, ce conteneur pourra être constitué par l'utilisation de l'objet `Panel Grid` de la palette. Au moment où celui-ci est déposé sur la zone d'édition, il se présente sous la forme d'un tableau de deux lignes et deux colonnes. Si d'autres colonnes doivent être rajoutées, il faut modifier la valeur de la propriété `columns` de la balise `<h:panelGrid>`.

Par défaut, chaque cellule du tableau contient un objet de type `HtmlOutputText` : ceux-ci peuvent être bien entendu supprimés.

```
<h:panelGrid border="1" columns="2">
     <h:outputText value="item1"></h:outputText>
     <h:outputText value="item2"></h:outputText>
     <h:outputText value="item3"></h:outputText>
     <h:outputText value="item4"></h:outputText>
</h:panelGrid>
```

Pour constituer le reste du formulaire de saisie, chacun des composants graphiques devra être déposé dans la cellule adaptée de la grille de présentation.

Les cellules de l'objet `HtmlPanelGrid` ne peuvent contenir qu'un seul composant graphique : il n'est donc pas possible, de prime abord, de disposer dans la même cellule une zone de texte et l'étiquette qui lui est associée (par exemple). Pour réaliser une telle opération, il est indispensable de faire usage d'un conteneur de type `HtmlPanelGroup` dans lequel les composants en question seront placés.

c. Intégration d'un texte simple et d'une zone de texte à ligne unique

On sélectionne successivement les objets `Output Text` et `Text Input` depuis la palette, pour les positionner correctement dans la zone d'édition. L'utilisation de la vue **Properties** permet de préciser certains attributs, tels que `id`, `value` et `style`.

```
<h:outputText id="otxtNom" value="Vos nom et prénom:"
style="color: #0080FF; font-weight: bold">
</h:outputText>
<h:inputText id="txtNom"></h:inputText>
```

d. Intégration d'une zone de mot de passe et d'un message

Il faut ici positionner les deux composants et associer le message à la zone de mot de passe. Cela permettra par la suite d'entreprendre un test de validation concernant le mot de passe, et d'afficher un message d'erreur adapté en cas d'échec de la validation (pour cause de mot de passe trop court, par exemple).

La réalisation d'un tel test de validation requiert également l'association de la zone de mot de passe avec un validateur standard, ou avec un validateur personnalisé. Ce point n'est pas traité ici, dans la mesure où l'on se limite à l'ajout des composants à la page web. Pour obtenir des détails sur l'association entre un composant et un validateur, voyez le titre Exploitation des validateurs standards de ce chapitre.

L'insertion des composants se fait par l'utilisation des objets `Secret Input` et `Message` de la palette. Cependant,

© ENI Editions - All rigths reserved

pour les insérer ensemble dans la même cellule de la grille de présentation, il faut préalablement positionner un conteneur **Panel Group** dans la cellule en question. Les deux composants doivent ensuite être successivement glissés/déposés dans ce conteneur.

Il est impératif de donner une valeur à l'attribut `id` de la balise correspondant au champ de mot de passe, de façon à permettre son association avec le message. La balise de restitution du message doit, quant à elle, comporter un attribut `for` dont la valeur est identique à celle de l'attribut `id` de la zone de mot de passe.

Pour imposer à l'internaute de saisir un mot de passe, il faut donner la valeur `true` à l'attribut `required` de la balise `<h:inputSecret>`. Si l'on souhaite qu'un message d'erreur adapté soit présenté lorsque cette situation survient, il est possible de renseigner l'attribut `requiredMessage` de la même balise.

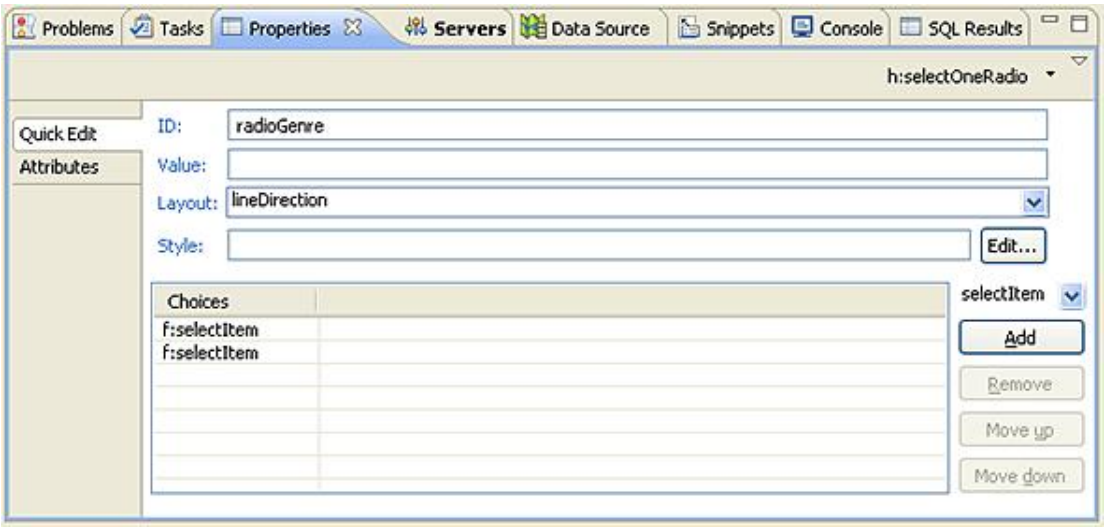
Un message d'erreur à afficher en cas d'échec de validation peut également être spécifié grâce à l'attribut `validatorMessage` du champ de mot de passe. Attention, dans ce cas, le message d'erreur éventuellement généré dynamiquement par un autre biais ne pourra jamais être affiché. Voir le titre Ajout de composants graphiques à une page - Principaux attributs de balises communs ci-dessus, concernant l'attribut `validatorMessage`, pour plus d'explications.

```
 <h:outputText style="color: #0080FF; font-weight: bold"
               id="otxtMotDePasse"
               value="Votre mot de passe:">
</h:outputText>
<h:panelGroup id="idPanel1">
<h:inputSecret id="txtMotDePasse"validatorMessage="Le mot de passe ne comporte pas
           assez de caractères!"
           required="true"
           requiredMessage="Vous devez saisir un mot de passe!">
</h:inputSecret>
<h:message id="messageMotDePasse" for="txtMotDePasse"
           style="color: #FF0000; font-weight: bold">
</h:message>
</h:panelGroup>
```

e. Intégration de boutons radios

L'insertion de boutons radios dans une page web se fait par le biais de l'objet `Select On Radio` de la palette **Eclipse**.

Un groupe de boutons radios permet à l'utilisateur de manifester un choix unique parmi un ensemble de propositions. Pour constituer un tel groupe, il suffit de réaliser un unique glissé-déposé de l'objet `Select One Radio` sur la page web. La définition des différentes propositions se fait très simplement depuis la vue **Properties**, qui présente une rubrique **Choices** permettant de rajouter les items. Ces items sont restitués sous la forme de balises `<f:selectItem>`.



Une fois que chacun des items a été déclaré, il faut impérativement en afficher les propriétés de manière à préciser le label (attribut `itemLabel`) et la valeur (attribut `itemValue`) qui leur correspondent. Il est également fortement recommandé de fixer une valeur pour leur propre attribut `id`.

La disposition des différentes propositions, en ligne ou en colonne, peut être précisée au moyen de l'attribut `layout` de la balise de restitution de l'objet de type `HtmlSelectOneRadio`. Les différentes valeurs autorisées sont

© ENI Editions - All rigths reserved

mentionnées dans la vue **Properties**.

```
<h:outputText id="otxtGenre" value="Vous êtes:"
      style="color: #0080FF; font-weight: bold">
</h:outputText>
<h:selectOneRadio id="radioGenre" layout="lineDirection">
      <f:selectItem id="idGenre1" itemLabel="Un homme"
                      itemValue="masculin"/>
      <f:selectItem id="idGenre2" itemLabel="Une femme"
                      itemValue="feminin"/>
</h:selectOneRadio>
```

f. Intégration d'une zone de liste déroulante à choix unique

La démarche à suivre ici est identique à celle qui vient d'être vue pour l'insertion de boutons radios. On dépose l'objet `Select One Menu` sur la zone d'édition, après l'avoir sélectionné dans la palette. La vue **Properties** spécifique à cet objet présente une rubrique **Choices** qu'il convient remplir avec les différents choix proposables à l'utilisateur. À chaque choix correspond une balise `<f:selectItem>`, pour laquelle il faut préciser les attributs `id`, `itemLabel` et `itemValue`.

```
<h:selectOneMenu id="selectEnfants">
<f:selectItem id="itemEnfant0" itemLabel="0" itemValue="0"/>
<f:selectItem id="itemEnfant1" itemLabel="1" itemValue="1"/>
<f:selectItem id="itemEnfant2" itemLabel="2" itemValue="2" />
<f:selectItem id="itemEnfant3" itemLabel="3" itemValue="3"/>
</h:selectOneMenu>
```

Malgré les facilités qu'offre la vue **Properties** pour saisir les différents items, son utilisation ne peut être envisagée que si le nombre d'éléments à insérer dans la zone de liste déroulante est réduit. Lorsque ce n'est pas le cas (par exemple pour constituer une liste contenant tous les nombres compris entre 1 et 100000), il convient de générer les items dynamiquement : cela peut être fait par l'exploitation d'un bean managé de type ArrayList. Ce cas de figure est décrit dans le paragraphe Liaison de composants à des sources de données externes de ce chapitre.

g. Intégration d'une zone de texte multilignes

Cette étape ne pose pas de difficulté particulière. Il peut être intéressant de fixer un nombre de lignes et de colonnes pour la zone de texte : il faut pour cela utiliser respectivement les attributs `rows` et `cols` de la balise de restitution `<h:inputTextArea>`.

```
<h:outputText id="otxtCommentaires" value="Vos commentaires:"
          style="color: #0080FF; font-weight: bold">
</h:outputText>
<br>
<h:inputTextarea id="txtCommentaires" cols="50" rows="10">
</h:inputTextarea>
```

h. Intégration d'un bouton de soumission

Le traitement des données saisies se fait après la validation du formulaire : celle-ci est initiée par l'utilisation d'un bouton de soumission, que l'on positionne sur la page web grâce à un objet `Command Button`. En général, la balise `<h:commandButton>` représentant ce bouton doit disposer d'un attribut `action`, dont la valeur est utilisée par l'implémentation JSF pour identifier la ressource (autre page JSP, JavaBean…) chargée de traiter les informations saisies par l'internaute. En d'autres termes, la valeur de l'attribut `action` doit correspondre à l'un des cas de navigation prévus dans le fichier de configuration des ressources de l'application.

Si l'attribut `action` n'est pas renseigné, les données sont traitées par la page elle-même.

Lorsque la valeur de l'attribut `action` est mentionnée, mais qu'elle ne correspond à aucun des cas de navigation prévu, l'environnement Eclipse le signale par un avertissement qui précise que les données du formulaire ne pourront pas être prises en charge.

© ENI Editions - All rigths reserved

```
<h:commandButton id="btnEnregistrerInformations"
value="Enregistrer les informations"
 action="sauvegarde">
</h:commandButton>
```

© ENI Editions - All rigths reserved

Exploitation des validateurs standards

1. Les différents types de validateurs et leur mode d'utilisation

a. Description des validateurs

La spécification JSF propose en standard quatre validateurs distincts : ils correspondent à des classes implémentant l'interface `javax.faces.validator.Validator`.

Cependant, l'implémentation JSF choisie dans le cadre de cet ouvrage (Sun JSF 1.2 RI) ne permet l'exploitation que de trois d'entre eux. Ils sont visibles dans la catégorie JSF Core de la palette, et leur utilisation directe au cours du développement d'une application web est rendu possible par l'exploitation de balises de restitution issues de la bibliothèque Core.

Le tableau suivant récapitule les informations concernant chaque validateur standard utilisable pour la création d'applications web : il précise la classe dont le validateur est issu, la balise `Core` qui le représente, et l'objet de la palette qui lui correspond. Le rôle du validateur est également mentionné.

Nom de classe	Balise Core	Objet de la palette (catégorie JSF Core)	Rôle
LengthValidator	<f:validateLength>	validateLength	Contrôle de la longueur d'une chaîne de caractères.
LongRangeValidator	<f:validateLongRange>	validateLongRange	Permet de contrôler une valeur numérique entière.
DoubleRangeValidator	<f:validateDoubleRange>	validateDoubleRange	Permet de contrôler une valeur numérique décimale.

Le quatrième validateur, non disponible dans l'interface Eclipse pour la conception de pages web, est issu de la classe `javax.faces.validator.MethodExpressionValidator`. Il est chargé de vérifier la conformité d'une méthode d'expression (c'est-à-dire qu'il s'assure qu'une expression correspond effectivement à l'appel d'une méthode sur un objet).

b. Composants supportant les validateurs

Dans le cadre du développement d'applications web, ces validateurs ne présentent d'intérêt que lorsqu'ils sont associés à un champ de saisie particulier. Un `LengthValidator` sera utilisé pour vérifier qu'un nom ou un mot de passe a bien été saisi, alors qu'un `LongRangeValidator` pourra vérifier que la quantité demandée pour un article de boutique en ligne est bien comprise entre deux valeurs données.

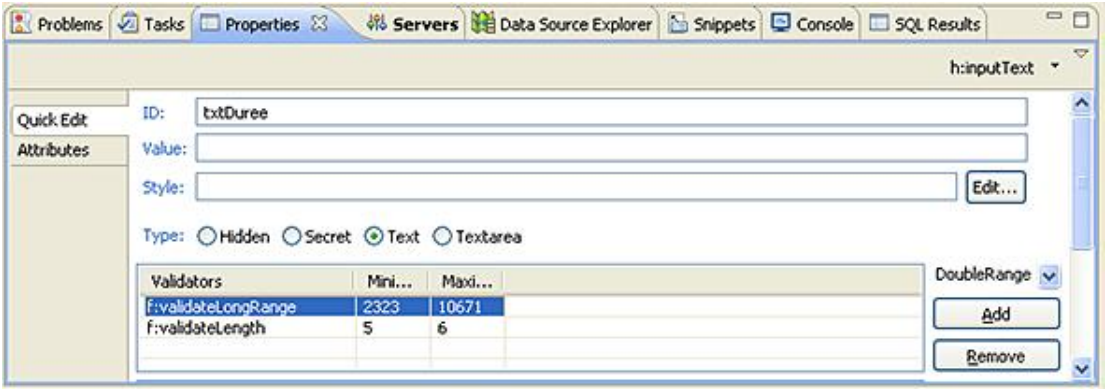
Les composants JSF capables de supporter les validateurs sont ceux qui sont directement issus de la classe `UIInput` : celle-ci présente en effet une méthode `addValidator(…)` permettant d'associer un validateur au composant, ainsi qu'une méthode `getValidators()` utilisable pour obtenir la liste de tous les validateurs associés au composant.

c. Mode d'utilisation des validateurs

Avec l'environnement Eclipse, l'association entre un composant graphique et un validateur particulier se définit à partir de la vue **Properties**, lorsque le composant de saisie en question est sélectionné. L'onglet **Quick Edit** de cette vue comporte une rubrique **Validators**, à partir de laquelle il est possible d'associer un ou plusieurs validateurs au composant, et de préciser leurs attributs.

Dans l'illustration ci-dessous, deux validateurs distincts sont associés à un champ de type `HtmlInputText`. Le premier d'entre eux est un `LongRangeValidator` vérifiant que la chaîne saisie dans le champ est un entier compris entre 2323 et 10671. Le second est un `LengthValidator` s'assurant que la longueur de la chaîne contient au minimum 5 caractères et au maximum 6. Naturellement, les deux contraintes imposées par les validateurs devront être vérifiées simultanément au moment de la saisie dans le champ.

© ENI Editions - All rigths reserved



2. Contrôler la longueur d'une chaîne de caractères

Il est courant, dans les formulaires de saisie, de devoir s'assurer que la chaîne saisie dans un champ a une certaine taille. Ce type de situation se rencontre par exemple pour vérifier qu'un mot de passe contient un nombre suffisant de caractères. La mise en œuvre d'un tel contrôle avec JSF se fait facilement par l'utilisation d'un validateur standard de type `LengthValidator`.

Ce validateur dispose de deux propriétés de type `int`, nommées `minimum` et `maximum`, qui permettent de réaliser le traitement attendu.

- Lorsqu'une valeur est attribuée à `minimum`, le validateur s'assure que la chaîne saisie dans le champ de formulaire contient au moins `minimum` caractères. Si ne n'est pas le cas, une exception de type `ValidatorException` est générée : la prise en charge de celle-ci par l'application web peut permettre l'affichage d'un message d'erreur adapté par l'intermédiaire d'un objet de type `HtmlMessage`.

- Lorsqu'une valeur est attribuée à `maximum`, le validateur s'assure que la chaîne saisie dans le champ de formulaire contient au plus `maximum` caractères. Si ne n'est pas le cas, une exception de type `ValidatorException` est générée : elle peut être prise en charge par l'intermédiaire d'un objet de type `HtmlMessage`.

- Si les deux propriétés sont définies, la chaîne saisie doit répondre simultanément aux deux contraintes précédemment citées.

Le code source suivant illustre l'exploitation d'un `LengthValidator` chargé de s'assurer que la chaîne saisie dans un champ `HtmlInputText` dispose d'une longueur comprise entre 5 et 10 caractères. En cas du non-respect de cette contrainte, un message d'erreur spécifique est présenté à l'internaute grâce à un objet `HtmlMessage` associé au champ de saisie.

```
<f:view>
     <h:form>
          <h:outputText id="txtIntitule"
               value="Saisissez impérativement un commentaire
               (entre 5 et 10 caractères):">
     </h:outputText>
     <br>
     <h:inputText id="txtCommentaire"
               label="Commentaire impératif"
               required="true">
     <f:validateLengthminimum="5"
           maximum="10">
     </f:validateLength>
     </h:inputText>
     <h:message id="messageCommentaire"
                for="txtCommentaire"
                style="color: #FF0000; font-weight: bold">
     </h:message>
     <br>
     <h:commandButton id="btnValidation"
                value="Validation du formulaire">
           </h:commandButton>
```


© ENI Editions - All rigths reserved

```
    </h:form>
</f:view>
```

Notez que la balise `<h:inputText>`, représentant le champ de saisie, dispose d'un attribut `required` positionné à `true`. En effet, bien que le `LengthValidator` soit configuré pour vérifier que la chaîne saisie contient entre 5 et 10 caractères, celui-ci n'est pas en mesure de faire un quelconque traitement lors de l'absence de saisie. En l'occurrence, aucune `ValidatorException` ne sera générée dans ce cas de figure et la soumission du formulaire se fera avec succès, malgré la contrainte exprimée par le validateur. Il est donc indispensable d'imposer une saisie dans ce champ texte, par l'usage de l'attribut de balise `required`.

La génération d'une `ValidatorException` par un validateur, quel que soit son type, lors du non-respect d'une contrainte, provoque systématiquement dans les applications web l'échec de validation du formulaire de saisie. Cela oblige l'internaute à fournir des informations cohérentes et limite les risques de mise en échec de l'application elle-même pour cause de fourniture de données inadaptées, par exemple.

3. S'assurer qu'une valeur numérique est comprise entre deux bornes

Deux validateurs permettent de répondre à ce besoin : `LongRangeValidator` et `DoubleRangeValidator`. Bien qu'ils soient tous deux en mesure de contrôler des valeurs numériques, leur différence réside dans le fait que le premier ne peut prendre en charge que des valeurs entières, alors que le second traite à la fois les valeurs entières et les valeurs décimales. Cette différence est clairement visible au regard des constructeurs respectifs des deux classes :

```
public LongRangeValidator(long maximum, long minimum)
public DoubleRangeValidator(double maximum, double minimum)
```

Les modes de fonctionnement de ces deux validateurs sont identiques : ils effectuent une comparaison entre la valeur saisie par l'internaute dans un champ de formulaire et deux valeurs, `minimum` et `maximum`.

- Si aucune valeur n'est saisie, aucun traitement n'est réalisé. Le validateur ne bloque pas la soumission du formulaire.
- Si les propriétés `minimum` et `maximum` sont précisées, le validateur s'assure que la valeur saisie se situe bien entre les deux. Lorsque ce n'est pas le cas, une exception de type `ValidatorException` est générée. Ce qui a pour effet d'empêcher la soumission du formulaire.
- Si seule la propriété `minimum` est indiquée, alors le validateur vérifie que l'utilisateur a mentionné une valeur supérieure ou égale à celle-ci. Dans le cas contraire, une `ValidatorException` est générée.
- Si seule la propriété `maximum` est indiquée, alors le validateur vérifie que l'utilisateur a mentionné une valeur inférieure ou égale à celle-ci. Dans le cas contraire, une `ValidatorException` est générée.

a. Exemple d'utilisation d'un LongRangeValidator

Après avoir déposé un champ de saisie dans la zone d'édition d'Eclipse, il est possible de lui associer un `LongRangeValidator` depuis la vue **Properties**. Les propriétés `minimum` et `maximum` peuvent également être précisées depuis cette vue. L'aperçu du code source de la page web montre que la balise `<f:validateLongRange>` correspondant au validateur est une balise enfant de celle représentant le champ de saisie (par exemple `<h:inputText>`).

```
<f:view>
     <h:form>
          <h:inputText id="txtDuree">
          <f:validateLongRangeminimum="2323"
                maximum="10671">
          </f:validateLongRange>
          </h:inputText>
          <h:message for="txtDuree">
          </h:message>
          <br>
          <h:commandButton value="Tester le validateLongRange">
          </h:commandButton>
     </h:form>
```

© ENI Editions - All rigths reserved

```
</f:view>
```

L'association d'un objet `HtmlMessage` avec le champ de saisie est utile pour présenter à l'utilisateur les raisons d'un éventuel échec de la validation. Dans le code ci-dessus, cette association est assurée par la déclaration de l'attribut `for` de la balise `<h:message>`, dont la valeur correspond à celle de l'attribut `id` du champ à contrôler.

Outre les cas précédemment cités pour lesquels le `LongRangeValidator` déclenche une `ValidatorException`, un échec de la validation peut également survenir lorsque la chaîne de caractères saisie par l'utilisateur n'est pas convertible en type long. Cela se produit dès qu'un caractère non numérique est trouvé dans la chaîne, et a fortiori le caractère point ("."). C'est la raison pour laquelle la validation d'un nombre décimal ne peut pas être assurée par un `LongRangeValidator` : il faut pour cela exploiter un `DoubleRangeValidator`.

b. Exemple d'utilisation d'un DoubleRangeValidator

La mise en œuvre de ce validateur ne diffère en aucun point de ce qui vient d'être dit concernant le `LongRangeValidator` : l'association du validateur à un champ de saisie peut se faire depuis la vue **Properties** d'Eclipse, en même temps que la précision des valeurs des deux propriétés `minimum` et `maximum`.

À l'exécution, on constate qu'effectivement la saisie d'un nombre décimal ne pose aucun problème, contrairement à ce qui est observé avec l'usage d'un `LongRangeValidator`.

```
<f:view>
     <h:form>
          <h:inputText id="txtDuree">
               <f:validateDoubleRange minimum="10" maximum="100">
               </f:validateDoubleRange>
          </h:inputText>
          <h:message for="txtDuree"></h:message>
          <br>
          <h:commandButton value="Tester le validateDoubleRange">
          </h:commandButton>
     </h:form>
</f:view>
```


© ENI Editions - All rigths reserved

Exploitation des convertisseurs standards

1. Les différents types de convertisseurs et leur mode d'utilisation

a. Description des convertisseurs

Un convertisseur est un objet issu d'une classe implémentant l'interface `Converter`, dont le rôle consiste à tenter la conversion d'une valeur donnée pour la rendre compatible avec une propriété particulière d'un JavaBean.

La spécification JSF définit en standard treize classes de convertisseurs, permettant d'assurer des conversions vers divers types numériques, ainsi que vers le type `Date`. Dans le cadre du développement d'applications web basées sur JSF, seuls deux convertisseurs sont proposés en standard :

- Le premier traite de la conversion des nombres, en facilitant notamment la prise en charge de plusieurs formats de données (pourcentages, monnaies, etc.) : il est disponible dans la catégorie **JSF Core** de la palette d'Eclipse sous le nom *convertNumber*. La balise de restitution qui lui correspond est `<f:convertNumber>`.

- Le second prend en charge les conversions de chaîne vers un format spécifique de date (et réciproquement). Il se situe dans la même catégorie de la palette et se nomme *dateTimeConverter*. Sa balise de restitution est `<f:dateTimeConverter>`.

b. Composants supportant les convertisseurs

L'analyse du diagramme des classes montre que seuls les composants graphiques JSF issus de classes basées sur `UIOutput` peuvent se voir associer un convertisseur.

Il est donc possible d'attribuer un convertisseur à un champ de saisie particulier, tel qu'une zone de texte ou une zone de liste déroulante à choix unique. Dans ce cas, le convertisseur est utilisé pour convertir la donnée saisie, de manière à la rendre compatible avec la suite du traitement (il s'agit le plus souvent du stockage de la valeur saisie dans une propriété d'un JavaBean).

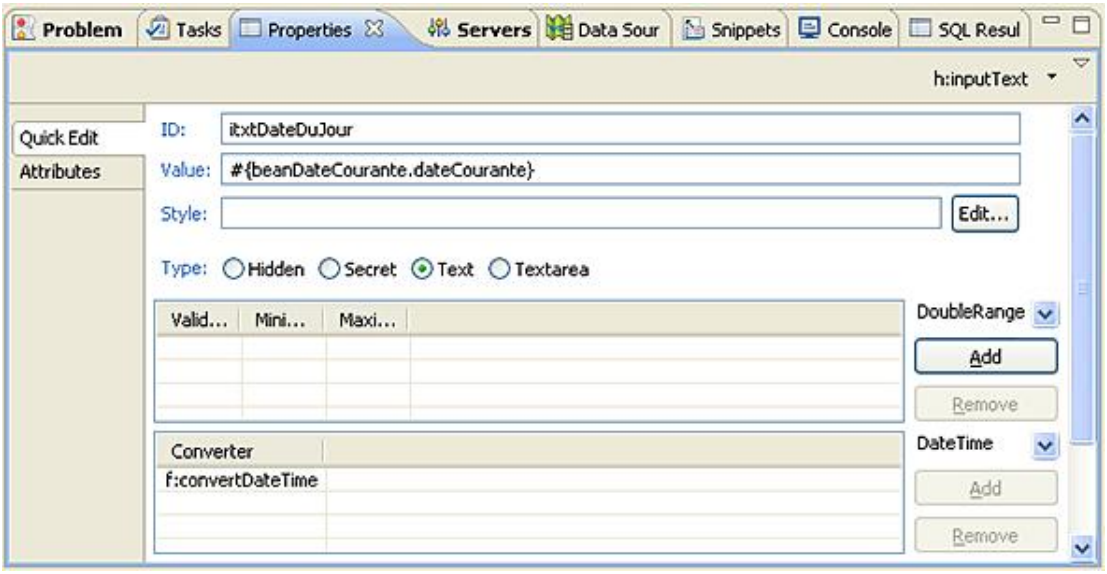
Mais il est également possible d'utiliser un convertisseur pour l'associer à un label ou un texte présenté en lecture seule. Dans cette situation, le convertisseur effectue un traitement réciproque de celui qui vient d'être décrit : il se charge de convertir, si cela est possible, la valeur d'une propriété de JavaBean en vue d'un affichage au niveau de l'interface web. Il s'agit donc d'une tentative de conversion d'un type quelconque vers le type `String`.

Il n'est pas possible d'associer plusieurs convertisseurs à un composant graphique particulier.

c. Mode d'utilisation des convertisseurs

De la même manière que ce qui a été vu concernant les validateurs, l'association entre un composant graphique et un convertisseur se fait facilement depuis la vue **Properties** d'Eclipse, lorsque le composant en question est sélectionné. Une fois le convertisseur déclaré, ses différents attributs peuvent rapidement être précisés depuis la même vue.

© ENI Editions - All rigths reserved



L'illustration proposée montre les propriétés d'un objet de type `HtmlOutputText`. La rubrique **Converter** contient la déclaration d'un convertisseur de type `DateTimeConverter`, alors que l'attribut `value` indique que ce champ texte est associé à la propriété `dateCourante` du bean managé `beanDateCourante`.

2. Conversion des valeurs numériques

a. Propriétés utilisées dans ce type de conversion

Ce type de conversion se fait à l'aide d'un `NumberConverter`. Lors de la déclaration de ce type de convertisseur, la vue **Properties** montre l'existence de quatre propriétés :

- **`type`** : sert à préciser si le convertisseur est utilisé pour convertir des devises monétaires, des pourcentages ou des nombres quelconques. Elle permet aussi d'indiquer que le convertisseur ne correspond à aucun de ces types prédéfinis : le type est alors personnalisé.

- **`Currency Code`** : cette propriété est utilisée pour des conversions de devises. Elle permet de préciser une chaîne de caractères correspondant à la devise en question (par exemple, *EUR* pour les euros). Les différentes valeurs possibles sont proposées dans une zone de liste déroulante et sont issues de la norme ISO 4217.

- **`Currency symbol`** : utilisée également pour la conversion de devises, elle sert à mentionner le symbole correspondant à la monnaie (par exemple, € pour les euros).

- **`Pattern`** : définit le format que doit respecter la chaîne ou la valeur de la propriété du JavaBean.

Suivant les choix réalisés au moment de la définition des propriétés du convertisseur, toutes les propriétés ne peuvent pas être précisées : en l'occurrence, l'usage d'un convertisseur de devises exclut la mention d'une valeur pour la propriété `Pattern`, alors que la déclaration d'un convertisseur de pourcentages rend inutile la précision d'une valeur pour les propriétés `Currency Code` et `Currency Symbol`.

b. Mode de fonctionnement de la conversion

Le fonctionnement de ce type de convertisseur diffère selon que la conversion se réalise en vue du stockage d'une valeur saisie dans une propriété de JavaBean, ou que, à l'inverse, elle est utilisée dans le but d'afficher le contenu d'une propriété de JavaBean sur l'interface web. La première situation correspond à la conversion d'une chaîne en Object ; la seconde est la conversion d'un Object en chaîne de caractères.

Lors de la conversion d'une chaîne en Object

La chaîne source est analysée dans le but de la convertir en objet de type Long ou Double.

© ENI Editions - All rigths reserved

- Si la longueur de la chaîne est nulle, aucune conversion n'est assurée.

- Si la propriété `Pattern` est renseignée, sa valeur doit respecter un certain format, selon les règles précisées dans la classe `java.text.DecimalFormat`. La précision d'une valeur pour `Pattern` provoque l'absence de prise en compte de la valeur de la propriété `type`.

- Si la propriété `Pattern` n'est pas renseignée, l'analyse de la chaîne source est réalisée compte tenu de la valeur de la propriété `type`. Selon le cas (conversion de devises, de pourcentages ou de nombres quelconques), le format à respecter correspond à celui mentionné dans les méthodes `getCurrencyInstance()`, `getNumberInstance()` et `getPercentInstance()` de la classe `java.text.NumberFormat`.

Lors de la conversion d'un Object en chaîne de caractères

La constitution de la chaîne de caractères s'effectue selon la démarche suivante :

- Si l'Object source correspond à la valeur `null`, une chaîne de longueur nulle est restituée.

- Si l'Object source est une chaîne de caractères, celle-ci est restituée telle quelle.

- Si la propriété `Pattern` possède une valeur, celle-ci doit respecter le formatage prévu dans la classe `java.text.DecimalFormat`. Elle rend inutile la précision d'une valeur pour la propriété type du convertisseur.

- Si la propriété `Pattern` n'est pas mentionnée, l'Object source est converti sous la forme d'une chaîne dont le format correspond, selon la valeur de l'attribut `type`, à une devise, à un pourcentage ou à un nombre quelconque. Le format en question correspond à celui mentionné dans les méthodes `getCurrencyInstance()`, `getNumberInstance()` et `getPercentInstance()` de la classe `java.text.NumberFormat`.

Échec d'une conversion

Si la conversion demandée, dans un sens ou dans l'autre, ne peut pas être assurée, une exception de type `ConverterException` est générée : cela empêche la soumission du formulaire dans lequel se trouve le composant JSF auquel est associé le convertisseur. Le cas échéant, le motif de l'échec de la conversion peut être précisé à l'internaute par le biais d'un objet de type `HtmlMessage` ou `HtmlMessages`.

c. Exemple d'utilisation d'un NumberConverter

Le code source ci-dessous illustre l'emploi d'un `NumberConverter` associé à un champ de type `HtmlInputText`. Ce convertisseur est représenté par une balise `<f:convertNumber>` possédant un attribut `pattern` renseigné : la validité de la conversion est donc déterminée en comparant le format de la chaîne saisie par l'internaute dans le champ `<h:inputText>` avec la valeur du pattern.

La présence d'un objet `HtmlMessage` associé au champ de saisie permet de présenter à l'utilisateur le motif d'un éventuel échec de conversion. En l'occurrence, la teneur du message en question correspondra systématiquement à la valeur de l'attribut `converterMessage` de la balise `<h:inputText>`.

```
<f:view>
     <h:form id="formulaireInscription">
          <br>
          <h:panelGrid border="1" columns="2">
               <h:outputText id="otxtValeurNumerique"
                     value="Saisissez une valeur numérique">
               </h:outputText>
               <h:panelGroup>
               <h:inputText id="itxtValeurNumerique"converterMessage="Format non respecté (0,00%)"
               requiredMessage="Vous devez renseigner ce champ!"
               required="true">
               <f:convertNumber pattern="0.00%"/>
               </h:inputText>
               <h:message id="msgValeurNumerique"
                          for="itxtValeurNumerique"
                          style="color: #FF0000; background-color:
                     #FFFF00">
               </h:message>
               </h:panelGroup>
```

© ENI Editions - All rigths reserved

```
          </h:panelGrid><br>
          <h:commandButton id="btnTester"
          value="Tester la conversion de l'information saisie">
     </h:commandButton>
     </h:form>
</f:view>
```

3. Conversion de la date et de l'heure

a. Propriétés utilisées dans ce type de conversion

Cette conversion s'obtient par l'utilisation d'un convertisseur de type `DateTimeConverter`. Quatre propriétés peuvent être définies depuis la vue **Properties** d'Eclipse pour ce convertisseur :

- `Type` : sert à indiquer si la conversion concerne la date uniquement, l'heure uniquement, ou les deux simultanément.
- `Date Style` : dans le cas d'une conversion de date, cette propriété est utilisée pour spécifier le style d'affichage/de stockage de la date. Il peut s'agir d'une forme courte, complète ou intermédiaire.
- `Time Style` : dans le cas d'une conversion d'heure, cette propriété est utilisée pour spécifier le style d'affichage/de stockage de l'heure. Il peut s'agir d'une forme courte, complète ou intermédiaire.
- `Pattern` : l'utilisation de cette propriété exclut l'emploi des trois autres propriétés précédemment évoquées. Elle permet de mentionner un format particulier d'affichage/de stockage de la date ou de l'heure. L'ensemble des formats autorisés est indiqué au sein d'une zone de liste déroulante.

b. Mode de fonctionnement de la conversion

De la même manière que pour un convertisseur `NumberConverter`, un convertisseur de type `DateTimeConverter` présente un mode de fonctionnement qui diffère, selon qu'il s'occupe de la conversion d'une chaîne en vue du stockage dans une propriété de JavaBean, ou qu'à l'inverse, il s'attache à la conversion du contenu d'une propriété de JavaBean pour en réaliser un affichage dans l'interface utilisateur.

Lors de la conversion d'une chaîne en Object

La chaîne source est analysée afin de tenter sa conversion en objet de type *Date*.

- Si l'objet `String` initial vaut `null`, ou si le nombre de caractères de cette chaîne est nul, la conversion n'est pas en échec, mais l'objet `Date` qui en résulte est `null`.
- Lorsque la propriété `pattern` est spécifiée, sa valeur doit être conforme aux règles mentionnées dans la classe `java.text.SimpleDateFormat`.

Lors de la conversion d'un Object en chaîne de caractères

Dans ce cas de figure, le convertisseur tente la conversion d'une propriété d'un JavaBean en chaîne de caractères. La propriété en question peut être d'un type quelconque compatible avec le type `Date` ou être de type `String`. La démarche suivante est réalisée pour mettre en œuvre la conversion :

- Si la valeur de propriété du JavaBean est égale à `null`, le résultat de la conversion est une chaîne de caractères de longueur nulle.
- Si la propriété du JavaBean est de type `String`, sa valeur est restituée telle quelle par le convertisseur.
- Si la propriété `pattern` est spécifiée, sa valeur doit respecter la syntaxe précisée dans la classe `java.text.SimpleDateFormat`. En cas d'utilisation de cette propriété, toutes les autres propriétés du convertisseur sont ignorées.

© ENI Editions - All rigths reserved

Échec d'une conversion

Lorsque cette situation survient, une exception de type `ConverterException` est générée. La soumission du formulaire dans lequel se trouve le composant associé au convertisseur est interrompue. Un message explicitant le motif de l'échec de la conversion peut être présenté à l'utilisateur si un objet `HtmlMessage` est rattaché au composant en question ou si un objet `HtmlMessages` est présent sur la page web.

c. Exemple d'utilisation d'un DateTimeConverter

À titre d'illustration, on peut imaginer la mise en place d'un formulaire de saisie à partir duquel un internaute mentionne une date quelconque au format *jj/MM/aaaa*. La validation du formulaire doit entraîner l'affichage du nom du jour correspondant à la date fournie (par exemple, *samedi*).

Afin de garantir la saisie de la date selon le format choisi, il est nécessaire d'associer un convertisseur de type `DateTimeConverter` au champ de saisie. Ce convertisseur doit disposer d'une propriété `pattern` correspondant au format à respecter.

Par ailleurs, pour rendre aisé l'affichage du nom du jour de la date saisie, il est très intéressant d'exploiter un JavaBean dont l'une des propriétés correspond précisément à cette date, et de lier cette propriété avec un label disposé sur la page web. L'affichage du nom du jour seul, sans précision de la date elle-même, se fait alors extrêmement simplement par l'emploi d'un `DateTimeConverter` associé au label. Ce convertisseur doit comporter une propriété `pattern` dont la valeur correspond au seul affichage du jour de la semaine, telle qu'elle est précisée dans la classe `SimpleDateFormat`.

Pour récapituler, la page à web exploite un bean managé (à déclarer dans le fichier de description des ressources de l'application web) comportant une propriété de type `Date` qui pourra servir à stocker la date saisie par l'internaute. De plus, la page web doit comporter :

- un champ de saisie `HtmlInputText` lié à la propriété de type `Date` du bean managé, et associé à un premier convertisseur `DateTimeConverter` dont la propriété `pattern` est égale à *dd/MM/yyyy*.
- Un label de type `HtmlOutputText`, également lié à la propriété de type `Date` du bean managé (mais en lecture seule cette fois), associé à un second `DateTimeConverter` dont la propriété `pattern` est fixée à *EEEE*.

Voici le code source d'un JavaBean répondant au besoin exprimé. La création d'une instance de ce JavaBean initialise la propriété de type `Date` avec la date du jour.

```
package premierProjetJSF;

import java.util.Calendar;
import java.util.Date;

public class BeanDateCourante {
     private Date dateCourante;

     public BeanDateCourante()
     {
          setDateCourante(Calendar.getInstance().getTime());

     }

     public Date getDateCourante() {
          return dateCourante;
     }

     public void setDateCourante(Date dateCourante) {
          this.dateCourante = dateCourante;
     }

}
```

La déclaration d'un bean managé s'appuyant sur cette classe peut se faire dans le fichier de description des ressources de l'application par les lignes suivantes :

```
<managed-bean>
          <managed-bean-name>
          beanDateCourante
          </managed-bean-name>
```

© ENI Editions - All rigths reserved

```
          <managed-bean-class>
          premierProjetJSF.BeanDateCourante
          </managed-bean-class>
          <managed-bean-scope>
          session</managed-bean-scope>
</managed-bean>
```

La page web contenant les différents composants JSF peut correspondre au code source ci-dessous. Les champs `<h:inputText>` et `<h:outputText>` sont liés à la propriété `dateCourante` du bean managé par l'intermédiaire de l'attribut de balise `value`. Ce point particulier, illustrant l'association d'un composant avec un JavaBean, est traité plus en détail dans le titre Liaison de composants à des sources de données externes de ce chapitre.

```
<f:view>
     <h:form id="formulaireInscription">
          <br>
          <h:panelGrid border="1" columns="2">
               <h:outputText id="otxtDateDuJour"
                              value="Saisissez une date valide:">
               </h:outputText>
               <h:panelGroup>
               <h:inputText id="itxtDateDuJour"value="#{beanDateCourante.dateCourante}"
               converterMessage="La date indiquée doit être au
          format JJ/MM/AAAA"
               requiredMessage="Vous devez renseigner ce champ!"
               required="true">
                         <f:convertDateTimepattern="dd/MM/yyyy" />
               </h:inputText>
               <h:message id="msgDateDuJour"
               for="itxtDateDuJour"
               style="color: #FF0000; background-color: #FFFF00">
               </h:message>
               </h:panelGroup>
               </h:panelGrid>
          <h:outputText id="otxtDateCourante"
          value="#{beanDateCourante.dateCourante}">
          <f:convertDateTime
               type="date" dateStyle="full"
               pattern="EEEE" />
          </h:outputText><br>
          <h:commandButton id="btnTester"
                         value="Afficher le nom du jour">
          </h:commandButton>
          </h:form>
</f:view>
```


© ENI Editions - All rigths reserved

Association d'écouteurs aux composants présents dans la page

Comme cela a été expliqué à la section Gestion des événements du chapitre Composition d'une application JSF, deux types d'événements peuvent être pris en charge dans le cadre des applications web basés sur la technologie Java Server Faces :

- Les `ActionEvent`, qui peuvent être générés par des composants implémentant l'interface ActionSource (notamment, les composants de type `HtmlCommandButton` ou `HtmlCommandLink`).
- Les `ValueChangeEvent`, qui concernent tous les composants JSF basés sur la classe `UIInput` (champs de saisie).

Ces événements peuvent être traités indifféremment par des écouteurs spécifiques, ou par des méthodes de JavaBeans.

1. Prise en charge des événements par un écouteur

Un écouteur est l'instance d'une classe implémentant l'interface `FacesListener`, ou l'une de ses interfaces dérivées. En l'occurrence, pour qu'une classe puisse traiter un `ActionEvent`, elle doit implémenter l'interface `ActionListener`. De même, une classe implémentant `ValueChangeListener` peut réceptionner des `ValueChangeEvent`.

a. Utiliser un écouteur prenant en charge les ActionEvent

L'exemple suivant montre le code source d'une classe `BeanActionListener` implémentant `ActionListener` : elle présente une méthode `processAction` dont l'argument correspond à un événement de type `ActionEvent`. Les instructions contenues dans cette méthode se contentent d'exploiter cet événement pour en connaître l'origine (la méthode `getComponent()` de la classe `ActionEvent` renvoie en effet la référence du composant ayant provoqué l'événement). L'identifiant de ce composant est ensuite affiché dans la console.

```
package premierProjetJSF.listeners;

import javax.faces.component.UIComponent;
import javax.faces.event.AbortProcessingException;
import javax.faces.event.ActionEvent;
import javax.faces.event.ActionListener;

public class BeanActionListener implements ActionListener {

     public void processAction(ActionEvent arg0) throws AbortProcessing
Exception {
          UIComponent composantSource=arg0.getComponent();
          System.out.println("Vous venez d'utiliser le composant
d'id "+composantSource.getId());
     }
}
```

Compte tenu du fait que cette classe implémente `ActionListener`, celle-ci peut être utilisée au sein d'une application web afin de prendre en charge les `ActionEvent` générés lors de l'exploitation de certains composants graphiques JSF (bouton de commande ou hyperlien).

Réaliser une association entre un composant JSF et un écouteur se fait facilement depuis la vue **Properties** d'Eclipse, lorsque le composant en question est sélectionné. Si ce composant est capable de générer l'un des deux types d'événements prévus par la spécification, une rubrique **Listeners** est en effet disponible. Son utilisation permet d'associer un ou plusieurs écouteurs au composant. Bien sûr, seuls les écouteurs compatibles avec le composant sont proposés en vue d'une association.

Dans le code source suivant, un composant de type `HtmlCommandButton` est présent dans la page web. La balise `<h:commandButton>` qui le représente dispose d'une balise enfant, `<f:actionListener>`. Celle-ci identifie un écouteur de type `ActionListener` associé au bouton ; elle possède un attribut obligatoire `type` permettant de mentionner la classe implémentant l'`ActionListener`. Il s'agit bien sûr ici du nom complet de la classe décrite précédemment : `BeanActionListener`.

```
<f:view>
     <h:form>
```

© ENI Editions - All rigths reserved

```
          <h:outputText id="otxtNom" value="Votre
nom:"></h:outputText>
          <h:inputText id="itxtNom"></h:inputText>
          <h:commandButton id="btnEnregistrer"
                          value="Enregistrer" >
               <f:ActionListener
          type="premierProjetJSF.listeners.BeanActionListener"/>
          </h:commandButton>
     </h:form>
</f:view>
```

L'utilisation de cette page web montre bien que la réalisation d'un clic sur le bouton de commande associé à l'`ActionListener` provoque l'affichage de l'identifiant du bouton dans la console Eclipse.

b. Utiliser un écouteur prenant en charge les ValueChangeEvent

La même démarche peut être utilisée pour illustrer la mise en œuvre d'un écouteur de type `ValueChangeListener` : une classe implémentant cette interface présente une méthode `processValueChange` prenant en charge un objet de type `ValueChangeEvent`. Cet objet correspond à un événement généré, et l'appel de sa méthode `getComponent()` permet d'obtenir la référence du composant qui en est la source.

```
package premierProjetJSF.listeners;

import javax.faces.component.UIInput;
import javax.faces.event.AbortProcessingException;
import javax.faces.event.ValueChangeEvent;
import javax.faces.event.ValueChangeListener;

public class BeanValueChangeListener implements ValueChangeListener
{

     public void processValueChange(ValueChangeEvent arg0)
               throws AbortProcessingException {

               //Affichage du nombre de caractères saisis
               //dans le composant à l'origine de l'évènement.

               UIInput composantSaisie=
                                   (UIInput)arg0.getComponent();
               System.out.println("La valeur du composant
"+composantSaisie.getId()+ " a été modifiée
("+composantSaisie.getValue()+")");
     }
}
```

Un composant JSF s'appuyant sur la classe `UIInput` peut être à l'origine d'un événement de type `ValueChangeEvent`. Pour assurer un traitement spécifique lorsque survient ce type d'événement, il faut associer le composant en question à un écouteur de type `ValueChangeListener`.

Dans l'exemple ci-dessous, un champ de type `HtmlInputTextarea` est positionné sur une page web, côte à côte avec un composant `HtmlCommandButton`. La balise qui le représente comporte une balise enfant `<f:valueChangeListener>`, indiquant qu'il est associé à un écouteur de type `ValueChangeListener`. Cette balise enfant dispose d'un attribut `type` dont la valeur correspond au nom d'une classe capable de prendre en charge les `ValueChangeEvent`. En l'occurrence ici : `BeanValueChangeListener`.

```
<f:view>
     <h:form>
          <h:inputTextarea id="itaCommentaires">
               <f:ValueChangeListener
     type="premierProjetJSF.listeners.BeanValueChangeListener"/>
          </h:inputTextarea>
          <h:commandButton id="btnValider" value="Valider">
          </h:commandButton>
     </h:form>
</f:view>
```

Lors de l'exécution de cette page, on constate qu'un message est indiqué dans la console Eclipse à chaque fois que

© ENI Editions - All rigths reserved

le formulaire de saisie est validé et que le contenu de la zone de texte a été modifié.

2. Associer plusieurs écouteurs à un unique composant

La spécification JSF prévoit la possibilité d'associer plusieurs écouteurs à un unique composant. Il est par exemple réalisable de déclarer N gestionnaires pour l'événement `ActionEvent` pour un bouton de commande particulier. Dans cette situation, l'événement généré est transmis successivement à l'ensemble des gestionnaires.

Le code source ci-dessous illustre ce cas de figure : un bouton de commande dispose de deux écouteurs capables de prendre en charge des `ActionEvent`. À chaque balise `<f:actionListener>` représentant les écouteurs correspond une classe spécifique implémentant l'interface `ActionListener`.

```
<f:view>
     <h:form>
          <h:commandButton id="btnTester"
               value="Tester plusieurs écouteurs">
          <f:actionListener
          type="premierProjetJSF.listeners.BeanActionListener"/>
          <f:actionListener
          type="premierProjetJSF.listeners.EvenementClicListener"/>
          </h:commandButton>
     </h:form>
</f:view>
```

3. Gestion des événements par l'exploitation d'une méthode de JavaBean

Outre la prise en charge des événements par des écouteurs spécifiques de type `ValueChangeListener` ou `ActionListener`, il est également possible de déléguer la gestion événementielle à de simples méthodes de classes.

Dans ce cas, les composants JSF susceptibles d'être la source des événements sont représentés sur les pages web par des balises comportant, selon le cas, soit un attribut `actionListener`, soit un attribut `valueChangeListener`. La valeur de ces attributs fait appel au langage d'expressions spécifique à JSF, et permet d'identifier une méthode de bean managé chargée de traiter l'événement. Si le nom d'une telle méthode peut être choisi librement, le reste de sa signature doit respecter les contraintes suivantes : la méthode doit être publique, de type `void` et accepter un unique paramètre de type `ActionEvent` ou `ValueChangeEvent` (selon le type d'événement à prendre en charge). Les deux lignes de code suivantes correspondent à des signatures de méthodes capables de traiter des événements générés par des composants JSF :

```
public void modificationContenu(ValueChangeEvent evenement)
public void traitementClic(ActionEvent actionEvent)
```

Exemple de prise en charge d'un ActionEvent

Pour illustrer ce concept, imaginons que l'on souhaite mettre en place une page web contenant un bouton de commande dont le label change chaque fois que l'internaute l'utilise (par exemple, le texte affiché sur le bouton passe de la valeur "OUVRIR" à la valeur "FERMER", et réciproquement).

Il faut, dans un premier temps, utiliser un JavaBean comportant une méthode capable de traiter un événement de type `ActionEvent`. Le code source suivant correspond à un tel JavaBean : il comporte en effet une méthode publique de type `void`, dénommée `traitementClic`, acceptant un unique paramètre de type `ActionEvent`. Les instructions contenues dans la méthode en question assurent l'obtention de la référence du composant `UICommand` à l'origine de l'événement, et effectuent une permutation de sa valeur par l'appel des méthodes `getValue` et `setValue` de la classe `UICommand`.

```
package premierProjetJSF.listeners;

import javax.faces.component.UICommand;
import javax.faces.event.ActionEvent;

public class BeanActionListener2 {

     public void traitementClic(ActionEvent actionEvent)
     {
     UICommand actionSource=(UICommand)actionEvent.getComponent();
     String texte=(String)actionSource.getValue();
```

© ENI Editions - All rigths reserved

```
     if (texte.toUpperCase().equals("OUVRIR"))
          actionSource.setValue("FERMER");
     else
          actionSource.setValue("OUVRIR");
     }
}
```

Dans un second temps, il est indispensable de déclarer ce JavaBean en tant que nouvelle ressource de l'application web : il sera alors considéré comme un bean managé et pourra donc être exploité par les pages JSP de l'application grâce au langage d'expressions. Une telle déclaration peut prendre la forme suivante dans le fichier de configuration des ressources de l'application :

```
<managed-bean>
     <description>
     Ce JavaBean comporte une méthode capable
     de traiter les ActionEvent.
     </description>
     <managed-bean-name>
     beanActionListener2</managed-bean-name>
     <managed-bean-class>
     premierProjetJSF.listeners.BeanActionListener2
     </managed-bean-class>
     <managed-bean-scope>
     session
     </managed-bean-scope>
</managed-bean>
```

Il ne reste plus enfin qu'à concevoir la page web elle-même. Celle-ci peut, par exemple, comporter une balise `<h:commandButton>` dont l'attribut `ActionListener` identifie la méthode `traitementClic` du bean managé `beanActionListener2` comme gestionnaire des événements `ActionEvent` qui surviennent sur le bouton.

```
<f:view>
     <h:form>
          <h:commandButton
          id="btnValidation"
          value="OUVRIR"
          actionListener="#{beanActionListener2.traitementClic}">
     </h:commandButton>
     </h:form>
</f:view>
```

À l'exécution, on constate qu'effectivement le bouton change de label à chaque fois que l'utilisateur clique dessus.

Un exemple de prise en charge d'un `ValueChangeEvent` par une méthode de JavaBean est présenté à la section Ajout de composants graphiques à une page - Principaux attributs de balises communs de ce chapitre.

© ENI Editions - All rigths reserved

Liaison de composants à des beans managés

Il existe différents moyens permettant de lier un composant JSF à un bean managé. Le plus courant consiste à associer le composant à une propriété de bean. Mais il est également possible de mettre en place une liaison avec l'une des méthodes du bean.

1. Lier un composant à l'une des propriétés d'un bean managé

Le mécanisme de liaison de composants graphiques JSF avec des propriétés de beans managés est très largement mis en œuvre dans la conception d'applications web basées sur cette technologie. En effet, ce type de liaison est utilisé pour sauvegarder les informations saisies dans un formulaire par l'internaute. Il est aussi exploité pour présenter à l'internaute les valeurs des propriétés de beans managés, en respectant parfois un formatage particulier (exemple : une date dont on ne veut afficher que le numéro du jour).

Concrètement, la plupart des composants graphiques JSF s'appuient sur des classes disposant de deux méthodes particulières : `getValue` et `setValue`. C'est le cas notamment de tous les composants basés sur les classes `UIOutput` et `UICommand`. Comme leurs noms l'indiquent, ces deux méthodes servent respectivement à lire et modifier la valeur du composant.

Bien sûr, le designer des pages web peut aisément faire appel à ces méthodes par l'exploitation de l'attribut `value` des balises représentant les composants. L'usage de cet attribut est particulièrement intéressant lorsqu'il s'agit d'afficher dans l'interface web une information stockée dans un fichier de messages. À ce sujet, de plus amples détails sont fournis dans le chapitre Internationalisation d'une application web de cet ouvrage, concernant l'internationalisation des applications. L'attribut `value` est aussi utilisé pour donner une valeur par défaut au composant (texte d'un label, par exemple). Mais il est aussi très souvent utilisé pour lier la valeur locale d'un composant graphique JSF avec la valeur d'une propriété de bean managé. Dans ce cas, la valeur de cet attribut fait appel au langage d'expression JSF, grâce auquel il est possible de mentionner le nom du bean et celui de la propriété à lier au composant.

a. Liaison en écriture (sauvegarde des données de formulaires)

La mise en place d'une telle liaison consiste à copier la valeur locale du composant graphique dans une propriété de JavaBean. L'application la plus courante de cet artifice technologique concerne la sauvegarde des informations saisies dans un formulaire.

Imaginons par exemple le classique cas du formulaire d'authentification qu'un internaute visualise lors de la consultation de la première page d'une application web protégée : l'internaute y est invité à saisir un login et un mot de passe. Afin de ne pas alourdir la visite du site, ces informations d'authentification doivent être stockées côté serveur. Dans le cadre d'une application basée sur la technologie JSF, ce stockage côté serveur peut être assuré par l'exploitation d'un bean managé de portée session, dont les propriétés sont compatibles avec les informations à sauvegarder. Le code source suivant correspond à un tel JavaBean : ses propriétés `name` et `password`, toutes deux de type `String`, peuvent servir à archiver côté serveur un login et un mot de passe.

```
package premierProjetJSF;

public class LoginBean {
     private String name;
     private String password;

     public LoginBean()
     {
          super();
     }
     public String getName() {
          return name;
     }
     public void setName(String name) {
          System.out.println("Modification du login: "+name);
          this.name = name;
     }
     public String getPassword() {
          return password;
     }
     public void setPassword(String password) {
          System.out.println("Modification du mot de passe:
"+password);
```

© ENI Editions - All rigths reserved

```
          this.password = password;
     }
}
```

Ce JavaBean doit être rendu disponible à l'ensemble de l'application web. Pour cela, il doit être déclaré dans le fichier de configuration des ressources en tant que bean managé. Ce bean devra nécessairement avoir une portée session, de manière à ce qu'il subsiste en mémoire le temps de la visite du site web par l'internaute, et qu'il ne soit pas partagé avec d'autres utilisateurs. Voici les lignes à ajouter au fichier **faces-config.xml** pour assurer cette déclaration :

```
<managed-bean>
          <managed-bean-name>
          loginBean
          </managed-bean-name>
          <managed-bean-class>
          premierProjetJSF.LoginBean
          </managed-bean-class>
          <managed-bean-scope>
          Session
          </managed-bean-scope>
</managed-bean>
```

Ce bean managé peut ensuite être utilisé par les autres ressources de l'application, en utilisant l'identifiant qui lui correspond : `loginBean`. La page web contenant le formulaire d'authentification contient différents composants, dont un champ texte de type `HtmlInputText` et un champ de mot de passe de type `HtmlInputSecret`. Les balises représentant ces composants doivent être liées aux propriétés `name` et `password` de `loginBean`. Pour cela, l'attribut de balise `value` est exploité : sa valeur indique le nom du bean managé et la propriété liée au composant.

Notez également la présence de l'attribut `action` de la balise `<h:commandButton>` représentant le bouton de soumission du formulaire. La valeur de cet attribut correspond à un cas de navigation prévu dans le fichier de configuration des ressources : l'emploi du bouton de soumission entraîne donc l'utilisation de ce cas de navigation, et donc la poursuite de la navigation vers une autre page JSP (dont le rôle peut consister à réafficher les paramètres d'authentification saisis).

```
<f:view>
     <h:form>
          <h:panelGrid border="1" columns="3">
          <h:outputText value="Login:"></h:outputText>
          <h:inputText value="#{loginBean.name}"
                id="champCodeClient">
          </h:inputText>
               <h:message for="champCodeClient"
                         id="messageCodeClient">
               </h:message>
          <h:outputText value="Mot de passe:">
          </h:outputText>
          <h:inputSecret value="#{loginBean.password}"
          id="champMotPasse">
          </h:inputSecret>

               <h:message for="champMotPasse"
                    id="messageMotPasse">
               </h:message>
          </h:panelGrid>
          <h:commandButton value="Valider"
           id="btnLogin" action="authentificationValide">
          <f:actionListener
          type="premierProjetJSF.listeners.EvenementClicListener"
          />
          </h:commandButton>
</h:form>
</f:view>
```

b. Liaison en lecture

Dans ce type de liaison, la valeur d'une propriété de bean managé est lue en vue de son affichage dans une page web. Le plus souvent, la valeur en question est présentée à l'internaute au travers de composants de type `HtmlOutputText` ou `HtmlInputText`. Mais d'une manière générale, elle peut être affichée dans tous les composants

© ENI Editions - All rigths reserved

graphiques instanciés à partir de classes de type `UIOutput` et `UICommand`.

Pour faire suite à l'exemple proposé précédemment pour la liaison en écriture, il est possible de mettre en place la page JSP qui doit être dévoilée à l'internaute après la validation du formulaire d'authentification. Cette page peut par exemple se contenter de réafficher les paramètres fournis par l'utilisateur. Pour cela, la page en question peut comporter deux labels de type `HtmlOutputText` dont les balises qui les représentent disposent d'un attribut `value`. Par utilisation du langage d'expression propre à JSF, la valeur de cet attribut doit référencer le bean managé `loginBean`, ainsi que la propriété adaptée (`name` ou `password`). Voici un exemple de code source correspondant à cette attente :

```
<f:view>
     <h:form>

          <h:outputText id="otxtTitre"
          value="Rappel des paramètres d'authentification saisis:"
          style="text-decoration: underline; font-weight: bold">
          </h:outputText>
          <br>
          <h:outputText id="otxtLabelLogin"
          value="Login: ">
          </h:outputText>
          <h:outputText id="otxtLogin"
          style="color: #FF0000"
          value="#{loginBean.name}">
          </h:outputText>
          <br>
          <h:outputText id="otxtLabelMotPasse"
          value="Mot de passe: ">
          </h:outputText>
          <h:outputText id="otxtMotPasse"
          style="color: #FF0000"
          value="#{loginBean.password}">
          </h:outputText>
          <br>
     </h:form>
</f:view>
```

2. Liaison à des méthodes de beans managés

Outre la liaison avec une propriété de bean managé, il est possible de lier un composant avec l'une des méthodes d'un JavaBean. L'illustration la plus classique de ce concept correspond à la mise en œuvre de la navigation dynamique.

Le formulaire d'authentification précédemment évoqué contient une illustration de la navigation statique entre pages web : la balise représentant le bouton de commande destiné à valider le formulaire dispose en effet d'un attribut `action` dont la valeur est fixée. Cette valeur correspond à l'un des cas de navigation prévus dans le fichier de configuration des ressources de l'application web. Cela signifie que, dans toutes les situations, la validation du formulaire d'authentification implique systématiquement une navigation vers la même page web.

Mais typiquement, il serait souhaitable d'éliminer une telle rigidité en rendant possible une navigation dynamique. En cas d'échec de l'authentification, notamment, il serait intéressant de ne pas autoriser la navigation vers la page suivante. Pour obtenir ce résultat, il est possible de déléguer le test de validité de l'authentification au bean managé `loginBean` contenant les paramètres saisis par l'internaute. Ce test de validité est alors assuré par une méthode spécifique du JavaBean, dont le résultat peut potentiellement servir à orienter dynamiquement la navigation web. Le code source suivant montre une méthode du bean `loginBean` dont le contenu permet de contrôler trivialement les paramètres d'authentification :

```
public String verificationAuthentification()
     {
          //Cette méthode renvoie une chaîne dont la valeur
          //peut servir à orienter la navigation.
          //La valeur de la chaîne dépend du résultat de
          //l'authentification.
          if (getPassword().equals("motdepasse1") && getName().equals
("utilisateur1")) return "authentificationValide";
          else return null;
     }
```

© ENI Editions - All rigths reserved

La méthode `verificationAuthentification` est de type `String`. Elle renvoie la chaîne `authentificationValide` lorsque l'authentification est réussie, ce qui correspond à l'un des cas de navigation prévu. En cas de saisie de paramètres erronés, l'authentification échoue et la méthode renvoie `null`.

Il reste à associer le résultat de l'exécution de cette méthode avec le bouton de soumission du formulaire, pour rendre la navigation dynamique. Cela se fait simplement par le biais de l'attribut de balise `action`, dont la valeur fait référence au bean managé et à la méthode `verificationAuthentification` grâce au langage d'expression.

```
<h:commandButton
     action="#{loginBean.verificationAuthentification}"
     value="Valider"
     id="btnLogin">
</h:commandButton>
```

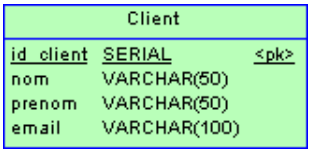

© ENI Editions - All rigths reserved

Liaison de composants à des sources de données externes

La liaison de composants à des propriétés et méthodes trouve également tout son intérêt lorsqu'il s'agit d'exploiter des sources de données externes.

En effet, en suivant une démarche visant à séparer clairement les vues d'une application web de la couche d'accès aux données, les pages web ne doivent jamais disposer d'un accès direct aux sources de données. Elles doivent, dans le cas des applications Java, exploiter des classes qui, elles seules, peuvent atteindre ces sources. En conséquence, lier un composant graphique JSF à une source de données équivaut à lier ce composant à un JavaBean qui, lui seul, exploite les données directement.

Considérons le cas d'une page web chargée de présenter à l'internaute la liste des clients d'une boutique en ligne, dans une zone de liste déroulante. Les informations concernant les clients sont stockées dans une table de base de données. Conformément au modèle physique présenté ci-dessous, chaque client est représenté par son nom, son prénom et son adresse e-mail.



Au niveau de l'interface web, une zone de liste déroulante peut correspondre à un composant JSF de type `HtmlSelectManyListBox`. Un élément particulier de cette zone de liste est, quant à lui, une instance de la classe `javax.faces.model.SelectItem` qui peut être restituée dans les pages web par une balise `<f:selectItem>`. Lorsque la zone de liste contient de nombreux éléments, il est très préférable de regrouper la totalité des items dans une classe de type `ArrayList` dont la restitution dans les pages web se fait par l'intermédiaire d'une unique balise `<f:selectItems>`.

En imaginant que le nombre de clients de la boutique virtuelle est important, le JavaBean chargé d'accéder directement à la base de données peut donc être un représentant d'une classe issue de `ArrayList`. Par l'exploitation d'une requête SQL adaptée, cette classe récupère l'ensemble des informations concernant les clients, puis crée un objet de type `SelectItem` spécifique pour chaque client trouvé. Notez que l'un des constructeurs de la classe `SelectItem` permet de mentionner la valeur et le label de l'élément de liste. Le code source suivant est celui d'une classe répondant à l'ensemble de ces critères.

```
package premierProjetJSF;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;
import java.util.ArrayList;

import javax.faces.model.SelectItem;

public class ArrayPersonnes extends ArrayList {

     private void chargerDonnees()
     {
          String label, value;
          try
          {
          //Tentative de connexion à une base de données
          //Postgresql.
          Class.forName("org.postgresql.Driver");
          Connection cnx= DriverManager.getConnection
     ("jdbc:postgresql://localhost/boutiquevirtuelle","test","test");

          //Récupération de la liste des clients.
          Statement st=cnx.createStatement();
          ResultSet rst= st.executeQuery("select * from clients");

          Personne personne;

          //Parcours du jeu d'enregistrements
          while (rst.next())
               {
               SelectItem element;
```

© ENI Editions - All rigths reserved

```
                //Création d'un SelectItem en précisant
                //sa valeur et son label.
                value=rst.getString("email");
                label=rst.getString("prenom")+
                                      " "+rst.getString("nom");
                element=new SelectItem(value,label);

                //Ajout du SelectItem au ArrayList.
                this.add(element);
                }
           }
           catch (Exception exc)
           {
                System.out.println("Problème lors du chargement
des données");
           }
      }
      public ArrayPersonnes()
      {
           chargerDonnees();
      }
}
```

Ce JavaBean doit être déclaré dans le fichier de configuration des ressources, en tant que bean managé de portée session. Cela le rendra accessible à toutes les autres ressources de l'application.

```
<managed-bean>
     <managed-bean-name>
     listePersonnes</managed-bean-name>
     <managed-bean-class>
     premierProjetJSF.ArrayPersonnes</managed-bean-class>
     <managed-bean-scope>
     session</managed-bean-scope>
</managed-bean>
```

Le bean managé est identifié par le nom `listePersonnes`. La page web chargée de présenter l'ensemble des clients à l'internaute doit comporter une balise `<h:selectManyListBox>` disposant d'une unique balise enfant `<f:selectItems>`. Cette dernière permet de restituer la totalité des éléments de la zone de liste déroulante : la valeur de son attribut `value` doit faire référence au bean managé `listePersonnes` par l'intermédiaire du langage d'expression JSF.

De fait, on constate à l'exécution que la zone de liste dévoile effectivement les informations attendues concernant les clients de la boutique virtuelle. Chaque élément de cette liste est traduit en langage HTML par une balise `<option>` comportant le label et la valeur du `SelectItem` généré dans le bean managé.

```
<f:view>
     <h:form>
          <h:outputText
               value="Liste des clients de la boutique virtuelle"
               style="font-size: 24px; font-weight: bold">
          </h:outputText><br>
          <h:selectManyListbox size="10">
               <f:selectItems value="#{listePersonnes}"/>
          </h:selectManyListbox>
          <br>
          <h:commandButton action="voirLesSelections"
                           value="Valider">
          </h:commandButton>
     </h:form>
</f:view>
```


© ENI Editions - All rigths reserved

Conception d'un convertisseur personnalisé

La spécification JSF propose en standard treize convertisseurs. Deux d'entre eux sont utilisables directement par le designer lors de la conception des pages d'une application web.

- Le premier est issu de la classe `NumberConverter` : son rôle consiste à tenter la conversion de la valeur saisie dans un champ de formulaire pour la rendre compatible avec la propriété de bean managé auquel est associé ce champ de saisie. Ce convertisseur standard réalise également un traitement réciproque, dans lequel la valeur d'une propriété numérique d'un bean managé est convertie en chaîne de caractères afin de pouvoir être présentée dans une page web.

- Le second est de type `DateTimeConverter`. Il assure des tentatives de conversion de la classe `String` vers la classe `Date`, et réciproquement. Il permet de s'assurer que la valeur saisie par un internaute dans un champ de formulaire particulier respecte bien un formatage compatible avec une date, et qu'elle peut donc être stockée en tant que valeur d'une propriété de JavaBean de type `Date`. À l'inverse, ce convertisseur sert aussi à vérifier que qu'une propriété de type `Date` d'un JavaBean peut voir sa valeur affichée dans une page web sous la forme d'une chaîne de caractères.

Vous trouverez de plus amples informations sur les convertisseurs standards à la section Exploitation des convertisseurs standards du chapitre Exploitation des composants standards JSF.

Malgré la souplesse proposée lors de l'exploitation de ces convertisseurs standards et le large champ d'application qu'ils couvrent, ceux-ci peuvent ne pas convenir dans certains contextes. Les traitements qu'ils réalisent peuvent par exemple s'avérer trop succincts, ou au contraire trop poussés. Il faut alors envisager la conception de convertisseurs personnalisés, répondant pleinement à certaines contraintes spécifiques. Pour cela, dans le cadre de la mise en place d'applications web basées sur la technologie JSF, il est nécessaire de développer une classe représentant le nouveau convertisseur, puis de la déclarer en tant que nouveau composant dans le fichier de configuration des ressources de l'application. L'ensemble des autres ressources, en particulier les pages web, peut alors exploiter le nouveau convertisseur.

1. Définition de la classe représentant le convertisseur

Une classe définissant un convertisseur doit impérativement implémenter l'interface `javax.faces.convert.Converter`. Celle-ci prévoit deux méthodes, `getAsString` et `getAsObject`. La première doit permettre d'assurer la conversion d'une chaîne de caractères en une instance de type `Object`, alors que la seconde effectue la conversion inverse. Voici les signatures de ces deux méthodes :

```
Object getAsObject(FacesContext context,
                   UIComponent component,
                   String value)

String getAsString(FacesContext context,
                   UIComponent component,
                   Object value)
```

En dehors du type et du nom de ces méthodes, la signature de celles-ci ne diffèrent que par leur troisième argument qui représente l'objet qui doit être converti. Le premier argument correspond au contexte JSF, c'est-à-dire celui de l'application web elle-même, alors que le second argument est une référence au composant JSF pour lequel la conversion est demandée.

Les instructions contenues dans le corps de ces méthodes peuvent être quelconques. La seule contrainte existante, mis à part le fait que la méthode doit au final renvoyer un objet du type adapté, concerne la situation où la conversion ne peut pas être assurée : il faut dans ce cas introduire une instruction provoquant la génération d'une exception de type `ConverterException`.

Le code source qui suit correspond à la définition d'une classe répondant à tous les critères requis pour constituer un nouveau type de convertisseur. Le rôle de ce nouveau convertisseur est de tenter la conversion d'une chaîne de caractères en valeur numérique entière positive paire. Les situations provoquant un échec de la conversion sont les suivantes :

- La chaîne saisie ne correspond pas à une valeur numérique.

- La chaîne saisie est une valeur numérique négative.

© ENI Editions - All rigths reserved

- La chaîne saisie est une valeur numérique positive impaire.

Lorsque l'une de ces situations est rencontrée, une `ConverterException` est générée : celle-ci est constituée d'un message d'erreur explicite utilisé pour instancier un `FacesMessage` qui peut potentiellement être récupéré au niveau de l'interface web afin d'être présenté à l'internaute.

Dans le cas d'une demande de conversion inverse, c'est-à-dire une conversion du type `Object` vers le type `String`, aucun traitement n'est proposé ici, dans un souci de simplification du code présenté. La méthode `getAsObject` se contente donc de renvoyer la valeur `null`.

```
package premierProjetJSF;

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

public class NombrePairPositifConverter implements Converter {

     public Object getAsObject(FacesContext arg0, UIComponent arg1,
                               String arg2)
{
          //Tentative de conversion de arg2 en objet
          //de type NombrePairPositif.

          int laValeurNumerique;
          NombrePairPositif nombrePair;

          try
          {
               //Tentative de conversion de l'argument arg2
               //en valeur de type int.
               //Cela peut bien sûr provoquer
               //une NumberFormatException.
               laValeurNumerique=Integer.parseInt(arg2);

               //Tentative de conversion en objet
               //de type NombrePairPositif.
               //Là encore, une Exception est susceptible
               //d'être générée.
               nombrePair=new NombrePairPositif(laValeurNumerique);

               //L'argument arg2 est bien un nombre pair positif.
               return nombrePair;

          }
          catch (Exception exc)
          {
               String msg=exc.getMessage();

               //Personnalisation du message en cas
               //de NumberFormatException.
               if (exc.getClass()==NumberFormatException.class)
                           msg="Valeur non numérique";

               //Renvoi d'une ConverterException avec génération
               //d'un FacesMessage contenant le motif de l'échec
               //de la conversion.
               throw new ConverterException(new FacesMessage(msg));
          }
     }

     public String getAsString(FacesContext arg0, UIComponent arg1,Object arg2) {
          return null;
     }
}
```


© ENI Editions - All rigths reserved

Notez enfin que les instructions qui composent ce nouveau convertisseur font appel à une classe `NombrePairPositif`, représentant un nombre pair positif. En réalité, la raison même du nouveau convertisseur est de s'assurer qu'une chaîne de caractère source peut effectivement être utilisée pour créer une instance de la classe `NombrePairPositif`.

Le code source de cette classe est présenté ci-dessous. Il s'agit d'un simple JavaBean réalisant un test trivial au moment de la mise à jour d'une propriété `valeurEntierePositive` et générant une exception le cas échéant.

```
package premierProjetJSF;

public class NombrePairPositif {

     int valeurEntierePositive;

     public NombrePairPositif()
     {valeurEntierePositive=0;}

     public NombrePairPositif(int valeur) throws Exception
     {
          setValeur(valeur);
     }


     public int getValeur() {return valeurEntierePositive;}
     public void setValeur(int val) throws Exception
     {
          if (val<0)
               throw new Exception("Impossible de définir un
nombre pair positif à partir d'une valeur négative.");
          if ((val%2)!=0)
               throw new Exception("Impossible de définir un
nombre pair positif à partir d'une valeur impaire.");
          valeurEntierePositive=val;
     }
}
```

2. Déclaration du nouveau composant

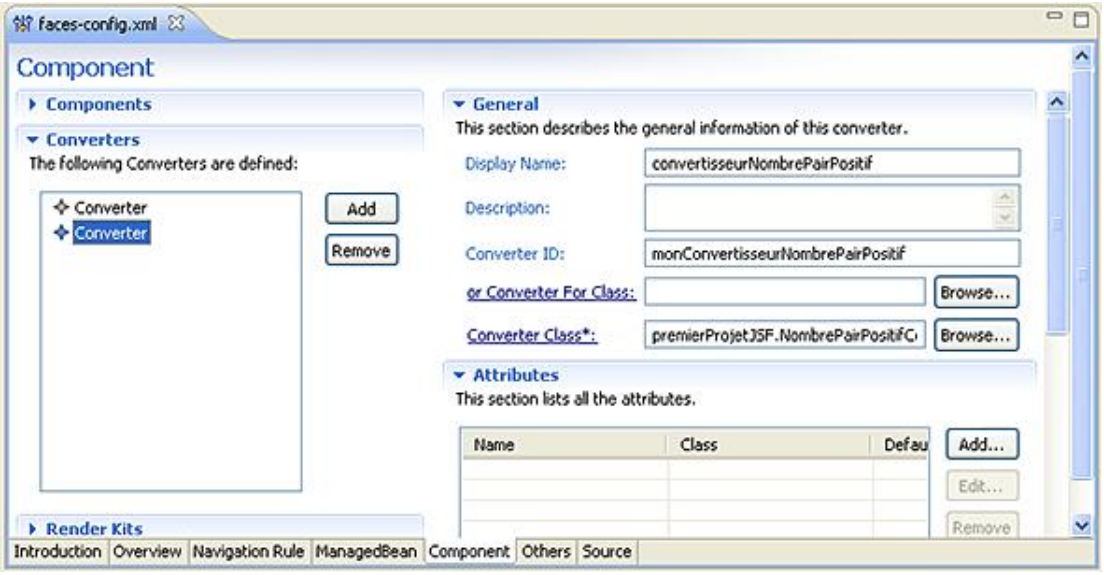
Pour qu'une application web puisse exploiter une classe représentant un nouveau type de convertisseur, cette dernière doit être déclarée explicitement dans le fichier de configuration des ressources. La déclaration se fait par l'intermédiaire de la balise `<converter>` et de deux balises enfants obligatoires `<converter-id>` et `<converter-class>`. Il est possible d'adjoindre d'autres balises enfants pour compléter cette déclaration, mais elles ne sont pas indispensables : celles-ci peuvent notamment à servir à donner un descriptif du convertisseur (balise `<display-name>`), ou à indiquer la valeur par défaut de certains de ses attributs (balises `<attribute>`, `<attribute-class>`, `<attribute-class>` et `<default-value>`).

Le code XML suivant correspond à la déclaration de la classe `NombrePairConverterConverter` en tant que nouveau type de convertisseur d'une application web. Cette déclaration se limite à la précision du nom de la classe (`converter-class`), de l'identifiant utilisable par les autres ressources pour tirer profit de ce convertisseur (`converter-id`) et de sa description (`display-name`).

```
<faces-config
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd"
    version="1.2">
...
<converter>
          <display-name>
          convertisseurNombrePairPositif</display-name>
          <converter-id>
          monConvertisseurNombrePairPositif</converter-id>
          <converter-class>
          premierProjetJSF.NombrePairPositifConverter</converter-class>
     </converter>
...
</faces-config>
```

© ENI Editions - All rigths reserved

Précisons que la déclaration d'un nouveau convertisseur se fait facilement dans l'environnement Eclipse : après avoir affiché le fichier de configuration des ressources dans la zone d'édition, il faut sélectionner l'onglet **Component** puis activer la rubrique **Converters**. Le formulaire de saisie qui se présente alors permet de mentionner tous les paramètres nécessaires pour assurer la déclaration souhaitée.



3. Faire appel au nouveau convertisseur dans les pages web

Une fois que la classe correspondant au nouveau type de convertisseur est développée et déclarée dans le fichier de configuration des ressources de l'application web, il est possible de faire usage du nouveau composant depuis toutes les autres ressources.

Concernant les pages JSP de l'application, la démarche d'utilisation du nouveau convertisseur est identique à celle suivie pour l'exploitation des convertisseurs standards : là où sont utilisées les balises de restitution `<f:convertNumber>` ou `<f:convertDateTime>`, on fait appel à la balise de restitution `<f:converter>` pour associer un convertisseur personnalisé à un composant graphique JSF. Cette balise doit obligatoirement faire mention d'un attribut `converterId` dont la valeur doit correspondre à celle de la balise `<converter-id>`, indiquée dans le fichier de configuration des ressources.

Voici un exemple de page JSP exploitant le convertisseur personnalisé d'identifiant `monConvertisseurNombrePairPositif`. Celui-ci est associé à un champ de saisie de type `HtmlInputText`. Au moment de la validation du formulaire contenant ces deux composants, le convertisseur tentera de réaliser la conversion demandée en s'assurant que la chaîne saisie par l'internaute peut effectivement être utilisée pour constituer une instance de la classe `NombrePairPositif`. Lorsque cette conversion n'est pas possible, une exception est générée par le convertisseur et le motif de l'échec de la conversion est présenté à l'internaute par le bais d'un composant de type `HtmlMessage`.

```
<f:view>
     <h:form>
          <H1><u>
          Test d'un convertisseur JSF personnalisé
          </u></H1>
          <h:outputText
value="Tentative de conversion en objet de type NombrePairPositif"
style="color: #0000FF">
          </h:outputText>
          <br><br>
          <h:message for="txtSaisie"
                style="color: #FF0000; background-color: #FFFF00">
          </h:message>
          <br>
          <h:outputText value="Si vous tapez ici une lettre, un
          nombre négatif, ou un nombre impair, une erreur de
          conversion se produira.">
          </h:outputText>
          <br>
          <h:inputText id="txtSaisie">
```


© ENI Editions - All rigths reserved

```
          <f:converterconverterId="monConvertisseurNombrePairPositif"/>
          </h:inputText>

          <br>
          <h:commandButton
               value="Tester la valeur saisie"
               action="succesConversion">
          </h:commandButton>
     </h:form>
</f:view>
```

© ENI Editions - All rigths reserved

Création d'un validateur personnalisé

Quatre validateurs sont proposés en standard par la spécification JSF, mais seuls trois d'entre eux peuvent être utilisés au cours de la conception d'applications web avec l'environnement de développement Eclipse. Ces validateurs sont les suivants :

- Le validateur de type `DoubleRangeValidator`, qui permet de comparer la valeur d'un composant graphique JSF avec deux valeurs `minimum` et `maximum`. Il prend en charge les nombres entiers, ainsi que les nombres décimaux.
- Le validateur de type `LongRangeValidator`, qui effectue un traitement similaire à celui entrepris par le `DoubleRangeValidator`, en dehors du fait qu'il ne prenne pas en compte les nombres décimaux.
- Le validateur de type `LengthValidator` : il teste la longueur de la chaîne de caractères correspondant à la valeur d'un composant JSF spécifique, en la comparant avec deux limites `maximum` et `minimum`.
- Le validateur de type `MethodExpressionValidator` : son rôle consiste à vérifier le format d'un objet de type `MethodExpression`, en s'assurant qu'il correspond bien à l'appel d'une méthode sur un objet. Ce validateur n'est pas disponible lors de la mise en place d'applications web.

Bien que ces validateurs standards permettent de répondre à un grand nombre de besoins, ils peuvent ne pas être adaptés à certaines situations particulières. Il revient alors aux concepteurs d'une application web d'envisager l'élaboration de validateurs personnalisés, en général chargés de réaliser des vérifications nettement plus poussées sur les valeurs des différents composants JSF disposés sur les pages web. La démarche de création et d'utilisation de validateurs personnalisés est similaire à celle qui a été évoquée à propos des convertisseurs personnalisés : il convient tout d'abord de concevoir la classe Java qui permettra d'instancier le nouveau validateur, puis de la déclarer dans le fichier de configuration des ressources de l'application web. Il reste ensuite à faire appel au validateur personnalisé dans les différentes pages web, par l'exploitation d'une balise de restitution adaptée.

1. Définition de la classe représentant le validateur personnalisé

Un validateur est toujours un objet issu d'une classe implémentant l'interface `javax.faces.validator.Validator`. La constitution d'une classe correspondant à un nouveau type de validateur amène donc nécessairement le développeur à respecter cette contrainte, en l'obligeant à déclarer et implémenter une méthode `validate` dont la signature est la suivante :

```
void validate(FacesContext context,
              UIComponent component,
              Object value)
              throws ValidatorException
```

Les arguments de cette méthode indiquent que la validation porte sur la valeur `value` d'un composant JSF *component*, et qu'elle concerne une application web spécifique, identifiée par le contexte *context*. Par ailleurs, la signature de la méthode `validate` précise également que celle-ci est susceptible de déclencher une exception de type `ValidatorException`. Cela doit effectivement être le cas dans toutes les situations où la validation de la valeur du composant est en échec. À ce sujet, les raisons qui déterminent le succès ou l'échec d'une validation dépendent entièrement du contexte dans lequel vient s'insérer l'application web à mettre en place. Ces raisons sont bien sûr traduites dans l'implémentation de la méthode `validate`.

La classe Java, dont le code source est présenté ci-dessous, correspond à un validateur personnalisé permettant de s'assurer que la valeur d'un composant JSF est une chaîne caractérisée par les éléments suivants :

- Sa longueur est au moins égale à la valeur de la propriété `minimumLength`.
- Elle contient au moins l'un des caractères spéciaux définis dans la propriété `caracteresSpeciaux`.
- Elle contient un nombre de caractères chiffrés au moins égal à la valeur de la propriété `nombreChiffres`.

Typiquement, un tel validateur peut être utilisé pour forcer un internaute à saisir un mot de passe disposant de critères précis. La méthode `validate` de cette classe indique les trois vérifications sont réalisées successivement, et que chacune d'elles peut donner lieu à la génération d'une exception de type `ValidatorException` lorsque le critère testé n'est pas respecté. Dans ce cas, une chaîne de caractères décrivant le motif de l'échec de la validation est utilisé pour instancier un objet de type `FacesMessage`. Cet objet peut potentiellement être récupéré au niveau de la

© ENI Editions - All rigths reserved

page web, pour être présenté à l'internaute, si un composant de type `HtmlMessage` ou `HtmlMessages` y a été placé.

```
package premierProjetJSF;

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

public class ValidatorPassword implements Validator {

     int minimumLength;
     char[] caracteresSpeciaux;
     int nombreChiffres;
     String message_minimumLength;
     String message_caracteresSpeciaux;
     String message_nombreChiffres;

     private boolean checkMinimumLength(String chaine){
          //Renvoie vrai si la longueur minimale est respectée.
          if (chaine.length()<getMinimumLength()) return false;
          else return true; }

     private boolean checkCaracteresSpeciaux(String chaine)
     {
          //Renvoie vrai si au moins l'un des caractères
          //spéciaux est dans la chaîne.
          char[] tableau=getCaracteresSpeciaux();
          char caractere;
          for (int i=0;i<tableau.length;i++)
          {
               for (int j=0;j<chaine.length();j++)
               {
                    caractere=chaine.charAt(j);
                    if (caractere==tableau[i]) return true;
               }
          }
          return false;     }

     private boolean checkNombreChiffres(String chaine){
          //Retourne vrai si la chaîne contient
          //un nombre de chiffres au moins égal au résultat
          //de la méthode "getNombreChiffres()".
          //Retourne false dans le cas contraire.
          int nb=0;
          for (int i=0;i<chaine.length();i++)
          {
               try {
                    Integer.parseInt((new Character(chaine.charAt(i))).
toString());
                    nb++;     }
               catch (NumberFormatException nfe) {}
          }

          if (nb>=getNombreChiffres()) return true;
          else return false;     }

     private String getMessageMinimumLength()     {
          String chaine="Doit contenir au moins
"+getMinimumLength()+" caractère(s)!";
          return chaine;     }

     private String getMessageNombreChiffres()     {
          String chaine="Doit contenir au moins
"+getNombreChiffres()+" chiffre(s)!";
          return chaine;     }

     private String getMessageCaracteresSpeciaux(){
```


© ENI Editions - All rigths reserved

```
          String chaineCarac="";
          char[] tableau=getCaracteresSpeciaux();

          for (int i=0;i<(tableau.length-1);i++)
          {

               chaineCarac+=tableau[i]+", ";
          }
          chaineCarac+="ou "+tableau[tableau.length-1];

          String chaine="Doit contenir au moins l'un des
caractères suivants: "+chaineCarac;
          return chaine;     }

     public ValidatorPassword() {
          //Définition des caractères spéciaux par défaut.
          char[] spec={'#','%','&','$','='};
          setCaracteresSpeciaux(spec);

          //Définition de la valeur minimale par défaut.
          setMinimumLength(6);

          //Définition du nombre de chiffres à utiliser par défaut.
          setNombreChiffres(2);     }

     public int getMinimumLength() {return minimumLength;     }

     public void setMinimumLength(int minimumLength) {
          //Seules les valeurs positives ou nulles sont acceptées.
          if (minimumLength>=0)
          this.minimumLength = minimumLength;     }

     public void validate(FacesContext arg0, UIComponent arg1,
                          Object arg2)
               throws ValidatorException {

          String pwd=arg2.toString();
          if (!checkMinimumLength(pwd)) throw new
ValidatorException(new FacesMessage(getMessageMinimumLength()));
          if (!checkCaracteresSpeciaux(pwd)) throw new
ValidatorException(new FacesMessage(getMessageCaracteresSpeciaux()));
          if (!checkNombreChiffres(pwd)) throw new
ValidatorException(new FacesMessage(getMessageNombreChiffres()));
     }

     public char[] getCaracteresSpeciaux() {
          return caracteresSpeciaux;     }

     public void setCaracteresSpeciaux(char[] caracteresSpeciaux) {
          this.caracteresSpeciaux = caracteresSpeciaux;     }

     public int getNombreChiffres() {return nombreChiffres;}

     public void setNombreChiffres(int nombreChiffres) {
          //Seules les valeurs positives ou nulles sont acceptées.
          if (nombreChiffres>=0)
               this.nombreChiffres = nombreChiffres; }

}
```

2. Déclaration du nouveau composant

Afin de rendre un nouveau validateur disponible à l'ensemble d'une application web, celui-ci doit être déclaré comme tel dans le fichier de configuration des ressources. Cela se fait par l'intermédiaire d'une balise `<validator>` disposant de plusieurs balises enfants permettant de préciser les caractéristiques du validateur :

© ENI Editions - All rigths reserved

- `<validator-id>` : cette balise obligatoire sert à indiquer par quel identifiant le nouveau validateur pourra être invoqué par les différentes ressources de l'application.

- `<validator-class>` : permet de spécifier le nom complet de la classe à partir de laquelle le validateur pourra être instancié. Cette balise est bien sûr obligatoire.

- `<display-name>` : permet de préciser un texte descriptif du validateur.

- `<attribute>`, `<attribut-name>`, `<attribute-class>` et `<default-value>` : sert à définir le nom, la classe et la valeur par défaut d'un attribut spécifique de la balise `<f:validator>` représentant le validateur personnalisé dans la page web.

Compte tenu de ces informations, la déclaration de la classe `ValidatorPassword` comme validateur personnalisé peut être réalisée dans le fichier de configuration des ressources par le code XML suivant :

```
<validator>
     <display-name>
     Validation du mot de passe</display-name>
     <validator-id>
     monValidateurDeMotDePasse</validator-id>
     <validator-class>
     premierProjetJSF.ValidatorPassword</validator-class>
     <attribute>
          <attribute-name>param1</attribute-name>
          <attribute-class>java.lang.Integer</attribute-class>
          <default-value>4</default-value>
     </attribute>
     <attribute>
          <attribute-name>param2</attribute-name>
          <attribute-class>java.lang.Integer</attribute-class>
          <default-value>3</default-value>
     </attribute>
</validator>
```

Depuis l'environnement Eclipse, une telle déclaration se fait simplement lorsque le fichier de configuration des ressources est présenté dans la zone d'édition : il faut sélectionner l'onglet **Component**, puis la rubrique **Validators**, pour visualiser un formulaire de saisie spécifique aux validateurs personnalisés. Par l'utilisation de celui-ci, le concepteur de l'application peut indiquer l'ensemble des caractéristiques du nouveau validateur.

3. Faire appel au nouveau validateur dans les pages web

Lorsque la classe caractérisant le nouveau validateur est développée, et que celle-ci est déclarée dans le fichier de configuration des ressources, le composant personnalisé peut être utilisé dans les différentes pages web d'une application.

Un validateur personnalisé est associé à un composant JSF spécifique par utilisation d'une balise `<f:validator>` disposant d'un attribut `validatorId`. Cet attribut permet de préciser l'identifiant du validateur, tel qu'il est donné dans le fichier de configuration des ressources. D'autres attributs peuvent éventuellement être précisés pour la balise `<f:validator>`, à condition que ceux-ci aient été prévus.

Le code source suivant illustre l'exploitation du validateur personnalisé d'identifiant `monValidateurDeMotDePasse`, issu de la classe `ValidatorPassword`. Ce validateur est ici utilisé pour contrôler la saisie réalisée dans un champ de type `HtmlInputSecret`. Au moment de la soumission du formulaire, le validateur vérifie que tous les critères requis sont respectés (longueur minimale, nombre de caractères spéciaux, nombre de caractères chiffrés). Si la validation échoue, la soumission du formulaire est interrompue et un message correspondant à la `ValidatorException` est présenté à l'internaute par le biais d'un objet `HtmlMessage`.

```
<f:view>
     <h:form>
          <h:panelGrid border="1" columns="3">
          <h:outputText value="Login:">
          </h:outputText>
          <h:inputText value="#{loginBean.name}"
                         id="champCodeClient">
          </h:inputText>
          <h:message for="champCodeClient" id="messageCodeClient">
```


© ENI Editions - All rigths reserved

```
          </h:message>
          <h:outputText value="Mot de passe"></h:outputText>
          <h:inputSecret value="#{loginBean.password}"
                         id="champMotPasse">
          <f:validator validatorId="monValidateurDeMotDePasse" />
          </h:inputSecret>

          <h:message for="champMotPasse" id="messageMotPasse">
          </h:message>
          </h:panelGrid>
          <h:commandButton
               action="#{loginBean.verificationAuthentification}"
               value="Validation
               id="btnLogin">
          </h:commandButton>
     </h:form>
</f:view>
```

© ENI Editions - All rigths reserved

Introduction

Ce chapitre expose la démarche permettant de créer de nouveaux composants graphiques s'appuyant sur la technologie Java Server Faces. Il comporte un exemple illustrant la conception d'un composant personnalisé destiné à réaliser de l'affichage simple, ainsi qu'un exemple montrant comment élaborer un composant personnalisé permettant la saisie d'informations.

La conception et l'utilisation d'un nouveau composant graphique personnalisé requiert plusieurs étapes :

- Conception de la classe régissant le comportement du nouveau composant.
- Déclaration du composant dans le fichier de configuration des ressources.
- Mise en place d'un fichier TLD comprenant les caractéristiques de la balise chargée de représenter le nouveau composant dans les pages JSP de l'application web.
- Définition d'une classe faisant office de `TagHandler` associé à la balise représentant le nouveau composant.
- Si nécessaire, conception d'une (ou de plusieurs) classe(s) chargée(s) d'assurer la restitution graphique du composant au niveau de l'interface web. Lorsque cette possibilité est exploitée, il est indispensable de modifier le fichier de configuration des ressources pour que cette classe de restitution soit exploitable par l'application web.
- Association éventuelle de gestionnaires d'événements, de validateurs et de convertisseurs au nouveau composant.
- Développement des pages JSP exploitant le composant graphique personnalisé.

© ENI Editions - All rigths reserved

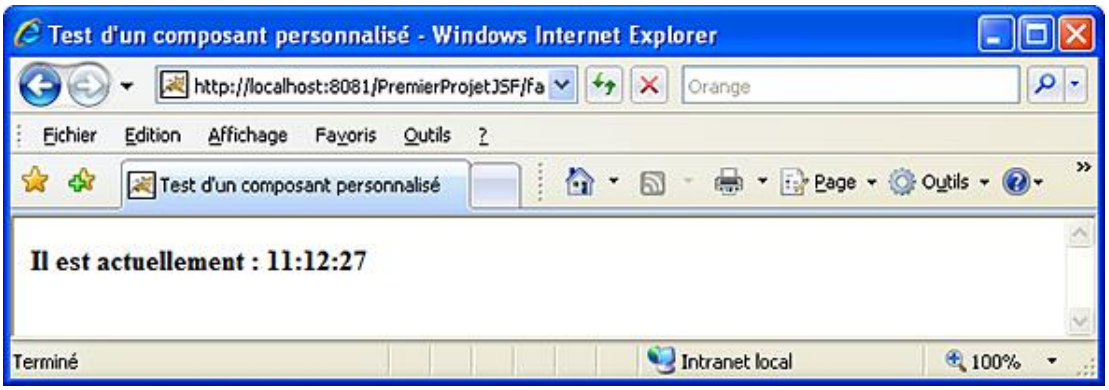
Élaboration des classes régissant le comportement des nouveaux composants

Les actions à entreprendre dans cette étape varient sensiblement compte tenu du type de composant à créer (lecture seule ou saisie autorisée). Des nuances sont également clairement observées si l'on considère que le nouveau composant doit gérer lui-même sa restitution graphique au sein des pages web, ou au contraire la déléguer à un `Renderer` particulier.

Cependant, dans toutes les situations, la classe représentant un nouveau composant graphique personnalisé doit impérativement dériver de la classe `UIComponentBase`. Le nom de la classe à créer peut être choisi librement, en sachant toutefois que les composants destinés à être affichés dans les pages web doivent conventionnellement être définis par des classes dont le nom commence par *Html*.

1. Création d'un composant destiné à l'affichage simple (lecture seule)

Considérons un exemple simple dans lequel un composant graphique personnalisé est chargé de présenter l'heure courante à l'internaute.



a. Si le composant gère lui-même sa restitution graphique

Pour que le composant puisse assurer sa propre restitution graphique, la classe qui le représente doit implémenter l'une des méthodes `encodeBegin` et `encodeEnd`, définies dans la classe parente `UIComponentBase`. Il est également possible de définir ces deux méthodes simultanément, comme le montre le code source présenté ci-dessous.

La méthode `encodeBegin` est invoquée lors de l'interprétation de la balise ouvrante représentant le composant, alors que la méthode `encodeEnd` est exploitée au moment de l'interprétation de la balise fermante correspondante.

```
package premierProjetJSF.composantsPersonnalises;

import java.io.IOException;
import java.util.Calendar;

import javax.faces.component.html.HtmlOutputText;
import javax.faces.context.FacesContext;
import javax.faces.context.ResponseWriter;

public class HtmlHeureCourante extends HtmlOutputText {

     @Override
     public void encodeBegin(FacesContext context)
                                        throws IOException {
          //Récupération de l'heure courante.
          Calendar calendrier=Calendar.getInstance();
          String heureCourante=
                    calendrier.get(Calendar.HOUR)+":"
                    +calendrier.get(Calendar.MINUTE)+":"
```

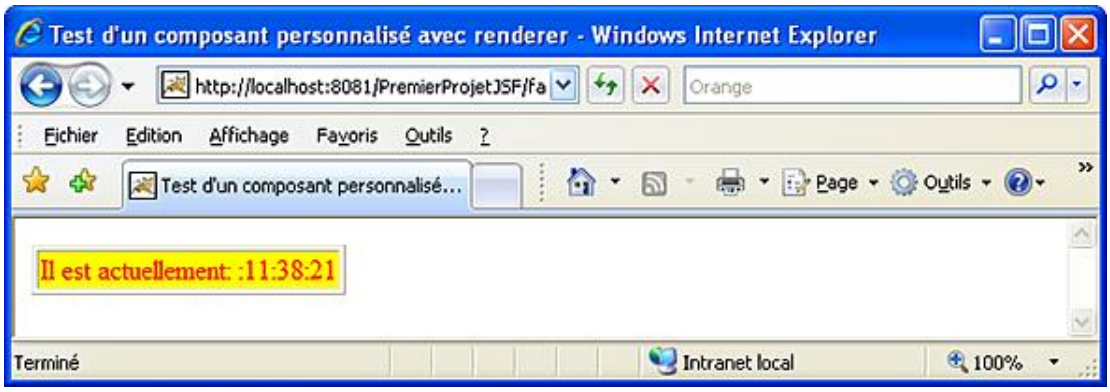
© ENI Editions - All rigths reserved

```
                    +calendrier.get(Calendar.SECOND);

          //Récupération d'un référence au flux de sortie.
          ResponseWriter out=context.getResponseWriter();
          String intitule="Il est actuellement ";
          String reponse="<b>";
          reponse+=intitule+": "+heureCourante;
          out.write(reponse);
     }

     @Override
     public void encodeEnd(FacesContext context)
                                        throws IOException {
          //Récupération d'un référence au flux de sortie.
          ResponseWriter out=context.getResponseWriter();
          String reponse="</b>";
          out.write(reponse);
     }
}
```

Ces deux méthodes acceptent en paramètre une référence au contexte de l'application web. Grâce à celui-ci, une référence au flux de sortie peut être obtenue, puis utilisée pour présenter le message attendu au niveau de l'interface graphique.

b. Si le composant délègue sa restitution graphique à un Renderer spécifique

Dans certaines situations, il peut être intéressant de ne pas fixer définitivement la restitution graphique d'un composant personnalisé : dans ce cas, la charge de la restitution doit être confiée à un `Renderer` particulier. Le designer de l'application web dispose alors de la possibilité de choisir l'apparence du composant, compte tenu de l'évolution des contraintes graphiques à respecter, en sélectionnant un `Renderer` adapté. Le composant précédent peut par exemple présenter le résultat attendu, sous la forme d'un texte encadré écrit en rouge sur fond jaune :



Un composant personnalisé peut déléguer la restitution graphique à un `Renderer` au moment de l'interprétation de la balise ouvrante correspondant au composant, ou au moment de l'interprétation de la balise fermante, ou dans les deux cas. Suivant le choix effectué, les méthodes `encodeBegin` et/ou `encodeEnd` ne doivent pas être implémentées dans la classe représentant le composant : elles doivent l'être dans la classe correspondant au `Renderer`. Par contre, la classe du nouveau composant doit obligatoirement définir une méthode `getFamily`, dont le rôle consiste à renvoyer une chaîne de caractères identifiant le nom de la famille du composant. Ce nom peut être choisi librement, mais doit permettre de sélectionner un `Renderer` particulier : il doit donc correspondre aux informations spécifiées dans le fichier de configuration des ressources, au moment de la déclaration d'un `Renderer` (voir la section Constitution du rendu des composants personnalisés - Déclaration du nouveau Renderer dans le fichier de configuration des ressources de l'application de ce chapitre).

Le code source suivant illustre la création d'un composant `HtmlHeureCourante2`, réalisant la même action que celui présenté précédemment. Ce composant conserve la charge de la restitution graphique de la balise fermante qui le représente (la classe implémente la méthode `encodeEnd`), mais délègue la restitution de sa balise ouvrante à un `Renderer`. Le `Renderer` en question n'est bien sûr pas précisé ici, puisque l'intérêt de cet artifice technologique est de permettre au designer web de choisir librement celui-ci. Par contre, la classe implémente la méthode `getFamily`, qui renvoie la chaîne `heureCourante2` identifiant la famille du nouveau composant : cette chaîne est utilisée par l'implémentation JSF pour identifier un `Renderer` adapté.

```
package premierProjetJSF.composantsPersonnalises;

import java.io.IOException;
```


© ENI Editions - All rigths reserved

```
import javax.faces.component.html.HtmlOutputText;
import javax.faces.context.FacesContext;
import javax.faces.context.ResponseWriter;

public class HtmlHeureCourante2 extends HtmlOutputText {

     public void encodeEnd(FacesContext context)
                                   throws IOException {
          //Surcharge la méthode encodeEnd du Renderer
          //éventuellement associé.
          ResponseWriter out=context.getResponseWriter();
          String reponse="</b>";
          out.write(reponse);
     }

     @Override
     public String getFamily() {
          return "heureCourante2";
     }

}
```

Si les méthodes `encodeBegin` et/ou `encodeEnd` sont définies conjointement avec la méthode `getFamily`, leur implémentation sera utilisée prioritairement à celle proposée dans le `Renderer` associé au composant.

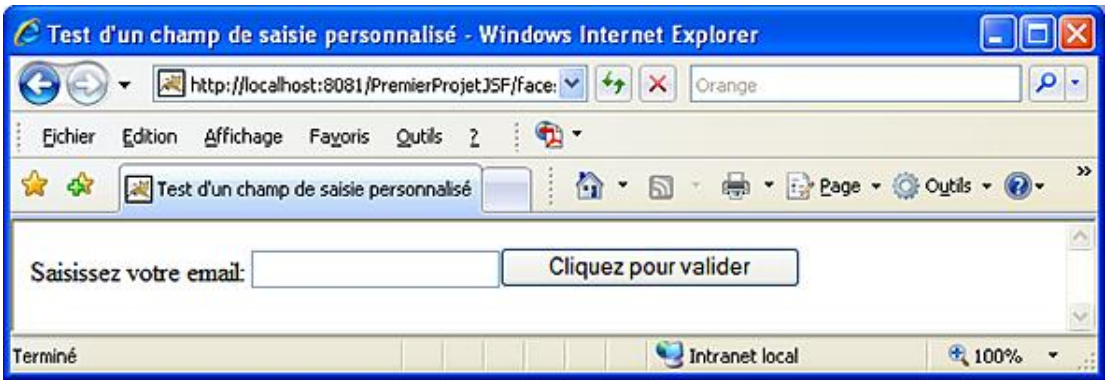
2. Création d'un composant permettant la saisie d'informations

De manière à pouvoir disposer de toutes les caractéristiques liées aux champs de saisies standards de la spécification JSF, les composants de saisie personnalisés doivent obligatoirement être issus de classes dérivées de `UIInput`.

Ces classes doivent implémenter la méthode `encodeEnd`, de manière à assurer la restitution graphique du composant, tout en permettant la prise en charge de convertisseurs éventuellement associés à celui-ci.

Par ailleurs, il convient également de permettre un accès aux paramètres passés dans la chaîne de requête au moment de la soumission du formulaire dans lequel sont insérés les champs de saisie personnalisés. Cet accès est assuré par l'implémentation de la méthode `decode` dans les classes correspondant aux composants.

Imaginons à présent la conception d'un composant graphique plus complexe que celui traité précédemment, composé d'un label, d'une zone de texte et d'un bouton. Ce type de composant peut par exemple servir à la saisie d'une adresse e-mail dans un formulaire web, comme le montre l'illustration suivante :



Le code source qui suit est une implémentation possible pour un tel composant :

```
package premierProjetJSF.composantsPersonnalises;

import java.io.IOException;
import java.util.Map;

import javax.faces.component.UIInput;
import javax.faces.context.ExternalContext;
```

© ENI Editions - All rigths reserved

```
import javax.faces.context.FacesContext;
import javax.faces.context.ResponseWriter;

public class HtmlInputButton extends UIInput {

     private String getIdChampNom(FacesContext contexte)
     {
     return getClientId(contexte)+":email";
     }

     private String getIdSoumission(FacesContext contexte)
     {
     return getClientId(contexte)+":soumission";
     }

     @Override
     public void encodeEnd(FacesContext context) throws IOException {
          //Récupération d'un référence au flux de sortie.
          ResponseWriter out=context.getResponseWriter();

          //Récupération de la valeur éventuellement déjà saisie
          //dans ce composant.
          String laValeur=(String)this.getValue();

          //Définition des labels
          String labelChamp="Saisissez votre email: ";
          String labelBouton="Cliquez pour valider";

          //Définition de la chaîne de caractères restituant le
          //composant au sein du flux de sortie
          //(interface web, généralement)
          String reponse=labelChamp
               +"<input type='text' name='"+getIdChampNom(context)
               +"' value='"+laValeur+"'>";
          reponse+="<input type='submit' value='"+labelBouton+"'
                    name='"+getIdSoumission(context)+"'>";

          //Écriture de la chaîne sur le flux de sortie.
          out.write(reponse);
     }

     @Override
     public void decode(FacesContext context) {
          ExternalContext externalContext
                                   =context.getExternalContext();
          Map<String, String> parametresRequete
                    =externalContext.getRequestParameterMap();

          String idChampSaisie=getIdChampNom(context);

          //Récupération de l'id du champ de saisie
          //dans la requête.
          String champSaisiRequete
                         =parametresRequete.get(idChampSaisie);
          //Mise à jour de la propriété interne correspondant au
          //champ de saisie.
          setSubmittedValue(champSaisiRequete);
     }
}
```

La méthode `encodeEnd` assure la restitution graphique du composant, en renvoyant vers le flux de sortie une chaîne de caractères correspondant à l'ensemble des balises HTML nécessaires. Cette chaîne se compose donc d'un label associé au champ de saisie, d'une balise `<input type='text'>` représentant le champ de saisie lui-même, et du bouton de soumission correspondant à la balise `<input type='submit'>`.

Ces deux balises doivent impérativement disposer d'un attribut `name` afin de pouvoir être prises en charge au moment de la soumission du formulaire. La valeur de ces attributs devant nécessairement être unique au sein du formulaire, celle-ci peut s'obtenir dynamiquement par appel de la méthode `getClientId`.

© ENI Editions - All rigths reserved

Il est par ailleurs souhaitable que l'information saisie par l'internaute puisse être réaffichée dans le composant lui-même, après validation du formulaire, dans le cas où une erreur de validation ou de conversion survient : il faut pour cela faire appel à la méthode `getValue`, chargée de renvoyer la précédente valeur du composant. Il est toutefois nécessaire de préciser que la validité de cette valeur ne peut être garantie que si le composant assure effectivement la prise en charge des informations passées dans la chaîne de requête. Cette prise en charge, réalisée par la méthode `decode` de la classe, consiste à récupérer l'ensemble des paramètres de requête par appel de la méthode `getRequestParameterMap`, puis à isoler le paramètre correspondant au champ de saisie : la valeur de ce paramètre est alors utilisée pour mettre à jour la valeur du composant personnalisé, par l'invocation de la méthode `setSubmittedValue`.

3. Permettre la prise en charge des attributs de balise

Que ce soit pour la conception de composants personnalisés en lecture seule, ou pour celle relative aux composants permettant la saisie, il peut être intéressant de permettre l'utilisation d'attributs de balises pour faciliter la personnalisation des composants graphiques.

L'exemple précédent gagnerait effectivement en souplesse s'il était possible, notamment, de personnaliser le label du champ de saisie ainsi que celui du bouton de soumission, grâce aux attributs de la balise correspondant au composant personnalisé. Pour obtenir un tel résultat, il faut légèrement modifier le code source de la méthode `encodeEnd`, chargée de restituer graphiquement la balise de fermeture du composant. Les modifications doivent porter sur l'attribution de valeurs, initialement fixes, aux variables locales `labelChamp` et `labelBouton` :

```
String labelChamp=
           (String)getAttributes().get("labelChampSaisie");
String labelBouton=
           (String)getAttributes().get("labelBoutonSoumission");
```

L'appel de la méthode `getAttributes` permet de récupérer une collection contenant toutes les propriétés du composant personnalisé. L'obtention d'une propriété particulière est réalisée par appel de la méthode `get` sur la collection, à laquelle le nom de la propriété à trouver est passé en paramètre. Dans l'exemple proposé, les propriétés `labelChampSaisie` et `labelBoutonSoumission` sont recherchées : celles-ci correspondent aux attributs de balise que l'on souhaite utiliser pour définir respectivement le label du champ de saisie, et celui du bouton de soumission. L'instruction suivante est un exemple de balise représentant le composant personnalisé : celle-ci se nomme `composantInputButton` et dispose des deux attributs évoqués.

```
<composantPerso:composantInputButtonlabelChampSaisie="Saisissez votre email: "
                     labelBoutonSoumission="Cliquez pour valider"/>
```

Malgré ces modifications dans le code source de la classe définissant le composant, la prise en charge des attributs de balise ne peut être effective que si le `TagHandler` associé à la balise du composant assure également les opérations nécessaires à leur traitement. La section Définition de balises de composants personnalisés - Définition d'un TagHandler associé à la balise représentant un composant de ce chapitre montre comment réaliser cette action.

4. Déclaration des nouveaux composants dans le fichier de configuration des ressources de l'application

Une fois que la classe correspondant à un nouveau composant graphique est intégralement constituée, celle-ci doit être déclarée au sein du fichier de configuration des ressources, de manière à permettre son utilisation en tant que composant personnalisé dans l'application web.

Cette déclaration se fait à l'aide d'une balise `<component>` directement issue de la balise `<faces-config>`. Elle comporte deux éléments essentiels : `<component-type>`, dont la valeur est utilisée par les `TagHandler` souhaitant s'associer au composant en question, et `<component-class>`, permettant de préciser le nom complet de la classe définissant le composant.

Le code source suivant est un extrait de fichier de configuration des ressources d'une application exploitant les trois composants personnalisés évoqués précédemment dans ce chapitre. Ces composants sont identifiés par les types `HtmlHeureCourante`, `monHtmlHeureCourante2`, et `monHtmlInputButton`. Ils correspondent respectivement aux classes `HtmlHeureCourante`, `HtmlHeureCourante2` et `HtmlInputButton` dont le code source a déjà été donné.

```
<component>
          <description>
          Description du composant personnalisé
          </description>
```

© ENI Editions - All rigths reserved

```
        <display-name>Composant Heure Courante</display-name>
        <component-type>
        HtmlHeureCourante
        </component-type>
        <component-class>
    premierProjetJSF.composantsPersonnalises.HtmlHeureCourante
        </component-class>
    </component>
    <component>
        <description>
        Une partie du rendu de ce composant est délégué à
        un Renderer
        </description>
        <display-name>
        Composant Heure Courante avec Renderer
        spécifique</display-name>
        <component-type>
        monHtmlHeureCourante2</component-type>
        <component-class>
    premierProjetJSF.composantsPersonnalises.HtmlHeureCourante2
        </component-class>
    </component>
    <component>
        <display-name>
        Composant constitué d'un champ de saisie et
        d'un bouton de soumission
        </display-name>
        <component-type>
        monHtmlInputButton</component-type>
        <component-class>
    premierProjetJSF.composantsPersonnalises.HtmlInputButton
        </component-class>
    </component>
```


© ENI Editions - All rigths reserved

Définition de balises de composants personnalisés

L'utilisation de composants personnalisés depuis les pages JSP d'une application web est assurée grâce à des balises spécifiquement conçues. Ces balises personnalisées doivent faire l'objet d'une déclaration au sein d'un fichier TLD, et être associées à des classes tenues de les prendre en charge, les `TagHandler`.

1. Mise en place d'un fichier TLD (Tag Library Descriptor)

Un fichier TLD doit être positionné, de préférence, dans le sous-répertoire WEB-INF/tlds de l'application web : son contenu présente la totalité des balises rattachées à la librairie.

Chaque balise comporte diverses informations permettant notamment :

- d'indiquer le nom de la balise, tel qu'il devra être utilisé dans les pages JSP pour instancier un composant particulier. Cette information, obligatoire, est donnée par l'intermédiaire de la balise `<name>`.
- de mentionner le nom complet de la classe correspondant au composant à instancier (balise `<tag-class>`). La précision de cette information est impérative.
- de préciser si la balise peut ou non disposer d'un corps (balise `<body-content>`). L'absence de cette information indique que la balise supporte effectivement la précision d'un contenu, alors que l'attribution de la valeur `empty` à la balise `<body-content>` empêche cette situation.
- de mentionner la liste de tous les attributs, obligatoires et optionnels, supportés par la balise personnalisée.

Chaque attribut de balise se définit à l'aide d'une balise `<attribute>`, capable de prendre en charge les balises enfant `<name>` et `<required>`. La première de ces balises permet de préciser le nom de l'attribut, tel qu'il devra être mentionné dans les pages JSP de l'application. Quant à la balise `<required>`, elle sert à indiquer si l'attribut en question est obligatoirement requis.

Le code source suivant correspond au contenu d'un fichier TLD nommé **mesPremiersTagsPersonnalises.tld**. Il définit trois balises personnalisées destinées à être associées aux trois composants personnalisés précédemment évoqués. Parmi elles, on retrouve la balise `composantInputButton` et la précision de ces deux attributs obligatoires `labelChampSaisie` et `labelBoutonSoumission`.

```
<?xml version="1.0" encoding="UTF-8" ?>
<taglib version="2.0">
     <tlib-version>1.0</tlib-version>
     <short-name>Mes composants JSF personnalises</short-name>
     <tag>
          <description>
          Composant affichant l'heure courante
          </description>
          <name>heureCourante</name>
          <tag-class>
premierProjetJSF.composantsPersonnalises.HtmlHeureCouranteTag
          </tag-class>

          <attribute>
               <name>id</name>
          </attribute>
          <attribute>
               <name>label</name>
               <required>true</required>
          </attribute>
     </tag>

     <tag>
          <description>
          Composant affichant l'heure courante avec
          un renderer spécifique
          </description>
          <name>heureCouranteAvecRenderer</name>
```

© ENI Editions - All rigths reserved

```
          <tag-class>
premierProjetJSF.composantsPersonnalises.HtmlHeureCourante2Tag
          </tag-class>
          <body-content>empty</body-content>
          <attribute>
               <name>id</name>
          </attribute>
          <attribute>
               <name>label</name>
               <required>true</required>
          </attribute>
     </tag>

     <tag>
          <description>
          Composant constitué d'un champ de saisie
          et d'un bouton de soumission
          </description>
          <name>composantInputButton</name>
          <tag-class>
premierProjetJSF.composantsPersonnalises.HtmlInputButtonTag
          </tag-class>
          <body-content>empty</body-content>
          <attribute>
               <name>id</name>
          </attribute>
          <attribute>
               <name>labelChampSaisie</name>
               <required>true</required>
          </attribute>
          <attribute>
               <name>labelBoutonSoumission</name>
               <required>true</required>
          </attribute>
     </tag>
</taglib>
```

2. Définition d'un TagHandler associé à la balise représentant un composant

La prise en charge d'une balise personnalisée par l'implémentation JSF est assurée par l'intermédiaire d'une classe spécifiquement dédiée. Celle-ci doit obligatoirement dériver de la classe `javax.faces.webapp.UIComponentELTag`, dont la définition indique ces méthodes `getComponentType` et `getRendererType` doivent être redéfinies.

a. Méthodes assurant la liaison avec un composant graphique et un Renderer

La méthode `getComponentType` sert à renvoyer une chaîne identifiant la classe correspondant au composant graphique associé à la balise. La valeur de cette chaîne doit impérativement correspondre avec celle de la balise `<component-type>` définissant un composant personnalisé dans le fichier de description des ressources de l'application.

La méthode `getRendererType` permet de renvoyer une chaîne de caractères identifiant le `Renderer` à associer éventuellement avec la balise. Si aucun `Renderer` ne doit être associé, alors cette méthode doit renvoyer la valeur `null`. Dans le cas contraire, la valeur retournée doit correspondre avec celle de la balise `<renderer-type>` définissant un `Renderer` personnalisé dans le fichier de configuration des ressources.

b. Méthodes permettant la prise en charge des attributs de balise

Comme cela a déjà été précisé, la prise en charge des attributs de balise ne peut être assurée uniquement depuis la classe définissant un composant personnalisé. En effet, à chaque attribut de balise doit correspondre une propriété privée de même nom au sein de la classe définissant le `TagHandler`. Ces propriétés privées doivent impérativement être rendues accessibles en écriture au moyen d'un accesseur `setNomAttribut`. Ainsi, à un attribut de balise `labelChampSaisie` correspond une propriété privée de même nom et un accesseur en écriture nommé `setLabelChampSaisie`.

Par ailleurs, la transmission de la valeur des attributs de balises entre un `TagHandler` et le composant personnalisé

© ENI Editions - All rigths reserved

associé ne peut se faire qu'au travers d'une méthode setProperties : celle-ci dispose d'un argument de type UIComponent correspondant au composant en question.

Le code source suivant correspond à la classe définissant un TagHandler pour la balise composantInputButton, représentant le composant personnalisé de type HtmlInputButton. On constate que la transmission de la valeur des attributs labelChampSaisie et labelBoutonSoumission au composant se fait par appel de la méthode getAttributes sur l'objet correspondant à ce composant. Cette méthode renvoie la liste des attributs, qu'il est possible de mettre à jour grâce à l'invocation d'une méthode put, en mentionnant le nom et la valeur des nouveaux attributs.

```
package premierProjetJSF.composantsPersonnalises;

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

public class HtmlInputButtonTag extends UIComponentELTag {

     private String labelBoutonSoumission;
     private String labelChampSaisie;

     public void setLabelBoutonSoumission(String label)
     {
          //Invoqué lorsque l'attribut de balise
          //labelBoutonSoumission est précisé.
          labelBoutonSoumission=label;
     }

     public void setLabelChampSaisie(String label)
     {
          //Invoqué lorsque l'attribut de balise labelChampSaisie
          //est précisé.
          labelChampSaisie=label;
     }

     @Override
     public String getComponentType() {
          //Renvoie le nom du type de composant à associer
          //à cette balise.
          //Ce nom doit être déclaré dans le faces-config.xml.
          return ("monHtmlInputButton");
          }

     @Override
     public String getRendererType() {
          //Aucun Renderer n'est associé
          return null;
     }


     protected void setProperties(UIComponent composant) {
          //Permet de mettre à jour les propriétés du composant,
          //compte tenu des attributs de balise.
          super.setProperties(composant);

          composant.getAttributes().put("labelChampSaisie",
                                              labelChampSaisie);
          composant.getAttributes().put("labelBoutonSoumission",
                                         labelBoutonSoumission);

     }
}
```

Dans un souci d'exhaustivité, voici le code source d'un TagHandler associé à la balise correspondant au composant personnalisé monHtmlHeureCourante2. Pour rappel, ce composant utilise un Renderer spécifique dont la description est donnée dans le paragraphe suivant. La chaîne identifiant ce Renderer est renvoyée par la méthode getRendererType de la classe.

```
package premierProjetJSF.composantsPersonnalises;

import javax.faces.component.UIComponent;
```

© ENI Editions - All rigths reserved

```
import javax.faces.webapp.UIComponentELTag;

public class HtmlHeureCourante2Tag extends UIComponentELTag {


     private String leLabel;

     public void setLabel(String etiquette)
     {
     //Cette méthode est invoquée lorsque
     //l'attribut de balise 'label' est précisé.
     leLabel=etiquette;
     }


     public String getComponentType() {
          //Renvoie le nom du type de composant à associer
          //à cette balise.
          //Ce nom doit être déclaré dans le faces-config.xml.
          return ("monHtmlHeureCourante2");
     }

     @Override
     public String getRendererType() {
          //Renvoie le nom du type de Renderer à utiliser avec
          //le composant associé.
          //Renvoie null si le composant associé doit gérer
          //lui-même le Rendering.
          return "heureAvecRenderer";
     }


     @Override
     protected void setProperties(UIComponent composant) {
          super.setProperties(composant);

          composant.getAttributes().put("label", leLabel);

     }
}
```


© ENI Editions - All rigths reserved

Constitution du rendu des composants personnalisés

L'objet de ce paragraphe est de décrire la démarche permettant de créer un `Renderer` personnalisé. Le rôle de ce type d'éléments est de permettre la restitution graphique de composants personnalisés qui ne gèrent pas cette action par eux-mêmes.

L'intérêt de définir des classes spécifiquement chargées de la restitution graphique est de permettre à un composant d'être représenté graphiquement de différentes manières. En outre, un `Renderer` particulier n'est pas spécifiquement lié à un type de composant donné : une fois défini, il peut être associé à n'importe quel type de composant graphique rattaché à une famille particulière.

1. Définition d'une classe représentant un Renderer personnalisé

Un `Renderer` personnalisé est simplement une classe dérivée de la classe `javax.faces.render.Renderer`. Les actions assurées pour restituer graphiquement la balise d'ouverture correspondant à un composant sont décrites dans une méthode `encodeBegin`, acceptant en paramètres la référence du contexte de l'application web, ainsi que celle du composant pour lequel la restitution est demandée. Une méthode `encodeEnd`, comportant les mêmes paramètres, est chargée de préciser ce qui doit être fait pour restituer graphiquement la balise de fermeture associée au composant.

Ces méthodes peuvent être implémentées conjointement, mais un `Renderer` peut également ne comporter que l'une d'elles. De plus, si le composant graphique définit lui-même l'une ou l'autre des méthodes `encodeBegin` et `encodeEnd`, l'implémentation précisée dans le `Renderer` est totalement ignorée.

```
package premierProjetJSF.composantsPersonnalises;

import java.io.IOException;
import java.util.Calendar;

import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.context.ResponseWriter;
import javax.faces.render.Renderer;

public class HeureCouranteRenderer extends Renderer {

     @Override
     public void encodeBegin(FacesContext context,
                                UIComponent component)
               throws IOException {
          //Méthode utilisée en début d'encodage,
          //à moins que le composant
          //dispose lui-même d'une méthode encodeBegin.
          super.encodeBegin(context, component);
          System.out.println("encodeBegin du Renderer pour
le composant de classe "+component.getClass().toString());
          Calendar calendrier=Calendar.getInstance();
          String intitule=
               (String)component.getAttributes().get("label");
          String heureCourante=
                         calendrier.get(Calendar.HOUR)+":"
                         +calendrier.get(Calendar.MINUTE)+":"
                         +calendrier.get(Calendar.SECOND);
          ResponseWriter out=context.getResponseWriter();
          out.write("
               <table border='1'>
               <tr bgcolor='yellow'><td>
                    <font color='red'>
                    "+intitule+":"+heureCourante+"
                    </font></td></tr></table>");
     }

     @Override
     public void encodeEnd(FacesContext context,
                         UIComponent component)
               throws IOException {
          //Méthode utilisée en fin d'encodage,
```

© ENI Editions - All rigths reserved

```
          //à moins que le composant
          //dispose lui-même d'une méthode encodeEnd.
          super.encodeEnd(context, component);
          System.out.println("encodeEnd du Renderer pour
le composant de classe "+component.getClass().toString());
     }
}
```

Ce code source correspond à celui d'un `Renderer` nommé `HeureCouranteRenderer`. Sa méthode `encodeBegin` permet d'afficher l'heure courante dans le navigateur web. Cette information est mentionnée à l'aide d'une police rouge sur fond jaune, et se trouve positionnée au sein d'une cellule de tableau HTML.

2. Déclaration du nouveau Renderer dans le fichier de configuration des ressources de l'application

De même que pour les composants personnalisés eux-mêmes, la création d'une classe dérivée de `Renderer` ne suffit pas à rendre un `Renderer` utilisable par une application web. Celui-ci doit en effet être déclaré comme tel au sein du fichier de configuration des ressources.

Cette déclaration doit comporter les informations suivantes :

- Nom complet de la classe implémentant le `Renderer` (balise `<renderer-class>`).
- Nom de la famille de composants à laquelle s'adresse le `Renderer` (balise `<component-family>`). Ce nom correspond à la chaîne de caractères renvoyée par la méthode `getFamily` des composants en question.
- Identifiant du `Renderer` (balise `<renderer-type>`). Celui-ci est utilisé pour associer un `TagHandler` à un `Renderer` donné.

Voici un extrait de fichier de configuration des ressources permettant de déclarer un `Renderer` nommé `heureAvecRenderer`, basé sur la classe `HeureCouranteRenderer`. Ce `Renderer` peut être utilisé avec les composants de la famille `heureCourante2`, ce qui est le cas des composants basés sur la classe `HtmlHeureCourante2` présentée précédemment.

```
<render-kit>
          <renderer>
               <display-name>
               Heure courante avec Renderer</display-name>
               <component-family>
               heureCourante2</component-family>
               <renderer-type>
               heureAvecRenderer</renderer-type>
               <renderer-class>
     premierProjetJSF.composantsPersonnalises.HeureCouranteRenderer
               </renderer-class>
          </renderer>
     </render-kit>
```


© ENI Editions - All rigths reserved

Définition des événements associés aux nouveaux composants

Pour rappel, les événements pouvant être produits par les composants graphiques JSF sont issus de la classe `ActionEvent` ou de la classe `ValueChangeEvent`. Les premiers peuvent être générés par des composants basés sur une classe implémentant l'interface `ActionSource`, alors que les seconds le sont depuis des composants dérivés de `UIInput`. Par ailleurs, la gestion de ces événements peut être assurée par des écouteurs de type `ActionListener` ou `ValueChangeListener`. Elle peut également l'être depuis de simples méthodes de JavaBeans, acceptant obligatoirement en argument l'événement en question.

Une prise en charge correcte des différents événements par des composants personnalisés suppose que le concepteur d'une application web réalise les opérations suivantes :

- Développer la/les classe(s) chargée(s) de gérer les événements.
- Déclarer dans la TLD les attributs de balises qui seront utilisés dans les pages JSP pour identifier les gestionnaires d'événements.
- Modifier le code source du `TagHandler` associé à la balise correspondant au nouveau composant, de manière à permettre la prise en charge de ces attributs de balise.

Pour illustrer ce propos, voyons comment rendre le composant de type `HtmlInputButton` capable de générer des événements.

1. Développement du gestionnaire d'événements

Compte tenu du fait que la classe `HtmlInputButton` dérive directement de `UIInput`, les composants personnalisés qui en sont issus sont en mesure d'être à l'origine d'événements de type `ValueChangeEvent`. La constitution d'une classe capable de gérer ces événements peut simplement se limiter à l'implémentation d'une méthode acceptant un `ValueChangeEvent` en argument. Dans le code source ci-dessous, la méthode `changementValeur` répond à cette caractéristique : son rôle consiste à afficher dans la console le nom du composant, ainsi que sa nouvelle valeur, à chaque fois que survient un `ValueChangeEvent`.

```
package premierProjetJSF.composantsPersonnalises;

import javax.faces.component.UIInput;
import javax.faces.event.ValueChangeEvent;

public class GestionnaireEvenements {
     public void changementValeur(ValueChangeEvent event)
     {
     System.out.println(
           "Un changement a eu lieu dans le composant de type "
           +event.getComponent().getClass().getCanonicalName()
           +".");

     //Si le composant est de type UIInput,
     //il est possible d'afficher sa valeur.
          try
          {
          UIInput champ=(UIInput)event.getComponent();
          System.out.println("Valeur actuelle: "+champ.getValue());
          }
          catch (Exception exc)
          {
          System.out.println("Impossible d'afficher la valeur
          actuelle du composant.");
          }
     }
}
```

Bien sûr, pour que ce gestionnaire d'événements soit exploitable depuis les pages JSP d'une application web, le fichier de configuration des ressources doit être mis à jour par la déclaration d'un bean managé correspondant à la classe `GestionnaireEvenements` :

© ENI Editions - All rigths reserved

```
<managed-bean>
          <managed-bean-name>
          gestionnaireEvenements
          </managed-bean-name>
          <managed-bean-class>
     premierProjetJSF.composantsPersonnalises.GestionnaireEvenements
          </managed-bean-class>
          <managed-bean-scope>
          Session
          </managed-bean-scope>
</managed-bean>
```

2. Modification de la TLD

Pour permettre au designer web de mentionner la référence du gestionnaire d'événements au travers de la balise représentant le composant personnalisé, il est nécessaire de définir un attribut spécifique. Cette déclaration comporte nécessairement le nom de cet attribut, qui, compte tenu de son rôle, peut conventionnellement être `valueChangeListener`. D'autres informations importantes concernant cet attribut doivent également être précisées : son caractère obligatoire (balise `<required>`) et la signature de la méthode chargée de gérer l'événement (balise `<method-signature>`).

```
<attribute>
     <description>
     Permet de spécifier un gestionnaire pour les évènements
     de type ValueChangeEvent.
     </description>
     <name>valueChangeListener</name>
     <required>false</required>
     <deferred-method>
          <method-signature>
     nomDeMethodeQuelconque(javax.faces.event.ValueChangeEvent)
          </method-signature>
     </deferred-method>
     <type>String</type>
</attribute>
```

Notez la particularité de la signature de méthode indiquée par la balise `<method-signature>` : elle ne doit comporter aucun type, et le nom de la méthode peut être tout à fait quelconque. La seule contrainte concerne le type de l'argument de méthode, qui lui, doit obligatoirement correspondre à celui de la méthode chargée de gérer l'événement (`ValueChangeEvent`).

3. Modification du TagHandler

Cette action consiste à déclarer une propriété supplémentaire dans la classe `HtmlInputButtonTag` : son nom doit être identique à celui de l'attribut de balise. Il est également indispensable de créer l'accesseur en écriture correspondant, et d'ajouter quelques lignes à la méthode `setProperties` pour que la valeur de la nouvelle propriété puisse être transmise au composant associé au `TagHandler`.

Étant donné que l'attribut de balise `valueChangeListener` est susceptible de disposer d'une valeur correspondant à une expression de méthode (par exemple `#{ gestionnaireEvenements.changementValeur}`), la nouvelle propriété du `TagHandler` doit être de type `javax.el.MethodExpression`. Naturellement, l'accesseur en écriture correspondant doit disposer d'un argument de même type :

```
private MethodExpression valueChangeListener;
public void setValueChangeListener
           (MethodExpression valueChangeListener)
           {this.valueChangeListener = valueChangeListener; }
```

La méthode `setProperties` doit, quant à elle, récupérer cet objet de type `MethodExpression` pour le convertir en une instance de la classe `MethodExpressionValueChangeListener`. Cette classe, directement dérivée de `ValueChangeListener`, permet en effet de créer des objets qui peuvent ensuite être ajoutés à la liste des ValueChangeListener associés à un composant donné, par appel de la méthode `addValueChangeListener` de la classe

© ENI Editions - All rigths reserved

UIInput.

```
protected void setProperties(UIComponent composant)
{
//...
if (valueChangeListener!=null)
     {
     //Si le composant est issu de la classe UIInput, alors
     //un ValueChangeListener peut lui être attribué.
     try
     {
     //Tentative de conversion du composant en UIInput.
     UIInput commande=(UIInput)composant;

     //Ajout du ValueChangeListener au composant.
     ValueChangeListener vcListener=
     new MethodExpressionValueChangeListener(valueChangeListener);
     if (vcListener!=null)
          {commande.addValueChangeListener(vcListener);}
     else     {
          System.out.println("Attention, l'expression de méthode
utilisée avec l'attribut valueChangeListener ne peut pas être
convertie en ValueChangeListener.");
          }
     }
catch (Exception exc)
     {
     System.out.println("Le composant n'accepte pas de
ValueChangeListener!");
     }
     }
//...
}
```

© ENI Editions - All rigths reserved

Permettre l'utilisation des nouveaux composants dans les pages JSP d'une application

L'utilisation de nouveaux composants dans une page JSP est simple. Il faut tout d'abord indiquer la référence de la TLD décrivant la (ou les) nouvelle(s) balise(s) par le biais d'une directive `taglib`, puis faire usage de ces balises pour exploiter les composants personnalisés.

Il faut toutefois s'assurer que les balises correspondant à ces composants sont bien positionnées entre les balises `<f:view>` et `</f:view>`. En outre, tous les composants susceptibles de permettre la saisie d'informations doivent être intégrés entre des balises de formulaire `<h:form>` et `</h:form>`.

Voici le contenu d'une page JSP faisant usage des deux composants personnalisés de type `HtmlInputButton` et `HtmlHeureCourante2`.

```
<%@taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<%@taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
<%@taglib uri="/WEB-INF/tlds/mesPremiersTagsPersonnalises.tld"
          prefix="composantPerso" %>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-
8859-1">
<title>Test de composants personnalisés</title>
</head>
<body>

<f:view>
<h:form>
<composantPerso:composantInputButton
labelChampSaisie="Saisissez votre email: "
labelBoutonSoumission="Cliquez pour valider"
valueChangeListener="#{gestionnaireEvenements.changementValeur}"
/>

<br>
<composantPerso:heureCouranteAvecRenderer
id="nouveau"
label="Il est actuellement: " />
</h:form>
</f:view></body>
</html>
```

© ENI Editions - All rigths reserved

Mécanismes mis en œuvre dans le cadre du multilinguisme

Dans le domaine du développement logiciel, il est aujourd'hui important de pouvoir mettre en place, dans des délais restreints, des applications qui puissent être facilement vendues et exploitées dans différents pays. Cela suppose, pour les développeurs, de prêter une attention toute particulière aux mécanismes qu'ils mettent en œuvre pour permettre une adaptation simple et rapide du contenu des interfaces graphiques : il est en effet particulièrement confortable, pour un utilisateur final, d'avoir à manipuler des interfaces dont les contenus sont adaptés à la langue employée dans son pays ou sa région.

Pour faciliter un tel travail d'internationalisation, la technologie Java offre depuis longtemps un ensemble de classes spécifiques, qui peuvent être exploitées pour la conception d'applications logicielles basées sur AWT ou SWING, ainsi que pour les applications orientées web. Ces classes sont particulièrement adaptées à la prise en charge de labels spécifiques à un certain nombre de pays. Cette section offre un aperçu de quatre de ces classes : `Locale`, `ResourceBundle`, `NumberFormat`, et `DateFormat`. Il précise également dans quelle mesure ces différentes classes peuvent être exploitées dans le cadre de la création d'applications web, en particulier celles basées sur la technologie Java Server Faces.

1. La classe Locale

a. Description

Cette classe, située dans le package **java.util**, permet d'instancier des objets représentant une langue utilisée par les habitants d'une région ou d'un pays particulier. Compte tenu des signatures des différents constructeurs de la classe, plusieurs informations doivent être fournies pour constituer un objet de type Locale :

```
public Locale(String language)
public Locale(String language, String country)
public Locale(String language, String country, String variant)
```

- L'argument `language` sert à indiquer le code ISO639 de la langue utilisée dans la région que l'on souhaite représenter. Ce code est obligatoirement constitué de deux caractères alphabétiques écrits en minuscules. Par exemple, le français est identifié par le code `fr`, alors que l'anglais l'est par la chaîne de caractères `en`. Pour information, la totalité des codes correspondant aux langues peut être consultée sur le web à l'adresse http://www.loc.gov/standards/iso639-2/langhome.html

- L'argument `country` permet de mentionner le code d'une région ou d'un pays. L'utilisation de ce paramètre impose la précision de la langue utilisée dans la région mentionnée (argument `language`). Cela signifie que la création d'un objet `Locale` réalisée à l'aide d'un code pays permet de prendre en compte la notion de dialecte (une langue donnée peut présenter des variantes suivant les pays où elle est employée : l'anglais du Royaume Uni (GB), par exemple, n'est pas identique à celui utilisé en Afrique du Sud (ZA)). Le code pays doit être fourni en référence à une liste standardisée, référencée ISO3166. Son contenu est visible sur le web à l'adresse http://www.iso.org/iso/country_codes/iso_3166_code_lists/english_country_names_and_code_elements.htm La valeur de l'argument `country` est toujours une chaîne constituée de deux caractères écrits en majuscules.

- L'argument `variant`, indissociable des deux autres, est introduit pour permettre aux éditeurs de logiciels de préciser un code qui leur est spécifique (WIN pour Windows, MAC pour Macintosh, POSIX pour Posix).

Précisons enfin que la classe `Locale` fournit en standard un ensemble de propriétés statiques facilitant l'instanciation d'objets `Locale` particuliers. Ainsi, la variable `Locale.FRANCE` permet d'obtenir simplement un objet `Locale` représentant le français parlé en France.

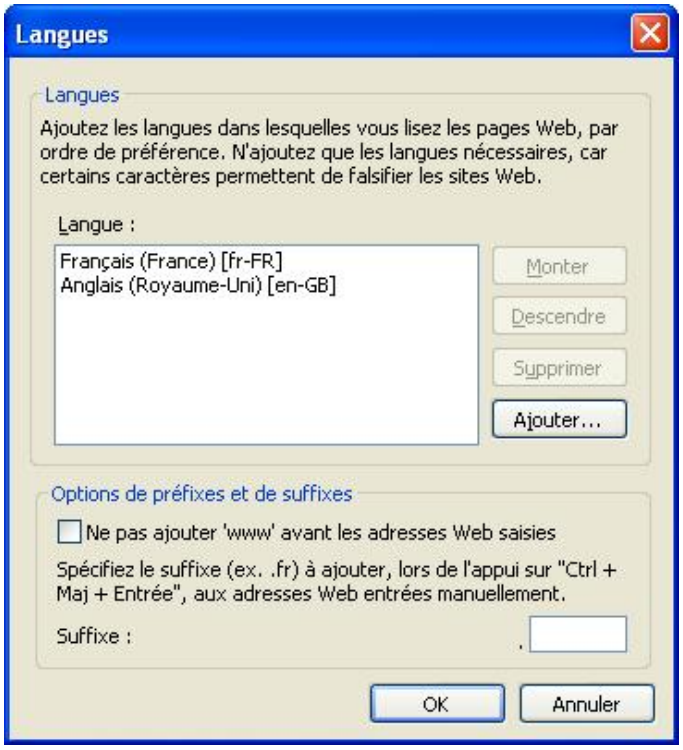
b. Utilisation dans le cadre de la création d'applications web

La classe `Locale` est systématiquement exploitée dans le cadre des applications web basées sur le langage Java (avec ou sans composants JSF). Dès qu'une session est ouverte sur un contexte d'application particulier, le langage défini par défaut dans le navigateur web est recherché, puis utilisé pour instancier un objet `Locale` correspondant aux habitudes linguistiques de l'internaute. Cet objet est stocké par la suite dans la session elle-même, afin de pouvoir être exploité durant toute la visite du site.

Pour rappel, le langage par défaut utilisé par le navigateur se définit simplement par le biais des préférences. Avec Internet Explorer, cette opération est réalisée depuis le menu **Outils - Options Internet** et le bouton **Langues**. La fenêtre qui se présente (voir illustration ci-dessous) indique le nom des langues définies, le pays où

© ENI Editions - All rigths reserved

elles sont utilisées, ainsi que le code ISO correspondant. Ce dernier est l'association entre le code langue et le code pays (fr-FR pour le français utilisé en France). Avec Mozilla Firefox, la même opération peut être réalisée depuis la rubrique **Langues** du menu **Outils - Options**.



Au-delà de la simple application du langage prévu par défaut par le navigateur web, l'objet `Locale` placé dans la session peut être manipulé par programmation. Cela permet au développeur, notamment, de proposer à l'internaute de basculer d'une langue à une autre par l'utilisation de menus constitués précisément dans ce but. Ce cas de figure particulier est étudié dans le détail, en guise d'illustration, dans la suite de ce chapitre.

2. La classe ResourceBundle et ses dérivées

a. Description

La classe `ResourceBundle` est une classe abstraite, rattachée au package java.util, dont les instances servent à représenter un ensemble de ressources utilisées spécifiquement pour une langue et une région particulières. Les ressources en question peuvent être de type quelconque, mais la plupart du temps, un `ResourceBundle` est exploité pour regrouper des chaînes de caractères utilisées par la suite dans diverses interfaces graphiques. Globalement, on peut dire que la notion de `ResourceBundle` se trouve au cœur des problématiques liées à la mise en œuvre des processus d'internationalisation. C'est elle en effet qui rend une application particulièrement adaptable à de nombreux langages.

Création d'un ensemble de ressources

Un `ResourceBundle` se définit le plus souvent à l'aide d'un identifiant particulier, appelé nom de famille, et d'un objet de type `Locale` représentant une langue utilisée dans une région particulière. Lorsque l'objet `Locale` n'est pas précisé par le développeur de l'application, les paramètres locaux définis sur le poste de travail de l'utilisateur sont pris en compte.

La classe `ResourceBundle` étant abstraite, son utilisation peut nécessiter la création de classes qui en dérivent et qui précisent son comportement. Cependant, la classe fournit un jeu de trois méthodes statiques nommées `getBundle`, dont l'utilisation permet d'instancier plus simplement un `ResourceBundle` particulier.

```
public static final ResourceBundle getBundle(String baseName)
public static final ResourceBundle getBundle(String baseName,
                                             Locale locale)
public static ResourceBundle getBundle(String baseName,
                                       Locale locale,
                                       ClassLoader loader)
```


© ENI Editions - All rigths reserved

À la lecture de ces trois constructeurs, on reconnaît les deux principaux composants d'un ResourceBundle : son nom de famille (argument baseName) et la langue représentée (argument locale).

Déclaration des ressources d'un ResourceBundle

Comme indiqué précédemment, un ResourceBundle est défini par un identifiant, qui caractérise la famille à laquelle il est rattaché. Pour être plus précis, une famille regroupe plusieurs ResourceBundle, qui diffèrent tous par la valeur de l'objet Locale qu'ils portent. Par exemple, une famille nommée *MesLabels* peut rassembler deux ResourceBundle, intitulés *MesLabels_fr_FR* et *MesLabels_en_GB* : le premier contient des éléments propres au français parlé en France, alors que le second est composé d'éléments linguistiques propres à l'anglais utilisé au Royaume Uni.

Tous les ResourceBundle d'une même famille doivent impérativement posséder les mêmes éléments, mais la traduction de ces éléments varie suivant la langue définie par l'objet Locale du ResourceBundle. Chaque élément est caractérisé par une paire d'informations de type Clé/Valeur. Ainsi, la clé bouton1 d'un élément peut posséder la valeur *"Se connecter"* pour dans un ResourceBundle correspondant au français. Le même élément se retrouve obligatoirement dans un ResourceBundle propre à l'anglais : il est caractérisé par la même clé (bouton1), mais dont la valeur correspond à la traduction anglaise du verbe *"Se Connecter"* (*login*).

Les deux principales techniques à mettre en œuvre pour déclarer des ressources consistent à utiliser les deux classes directement dérivées de ResourceBundle. Il s'agit des classes PropertyResourceBundle et ListResourceBundle.

- Faire une déclaration à l'aide d'un PropertyResourceBundle.

Un PropertyResourceBundle est un objet prenant en charge un fichier de type texte dans lequel sont déclarés les différents éléments d'un ResourceBundle. L'exemple qui suit illustre le contenu d'un tel fichier, valable pour la langue française : chaque ligne représente un élément particulier, composé de sa clé et de sa valeur.

```
titre1=Bonjour, vous êtes les bienvenus sur notre site!
bouton1=Se connecter
francais1=Version française
anglais1=Version anglaise
```

Il va de soi que les valeurs de chacune des clés est obligatoirement de type String.

Il existe une convention de nommage à respecter pour que des fichiers texte puissent être exploités par un objet de type PropertyResourceBundle : le nom du fichier doit comporter le nom de la famille de ressources, le code ISO de la langue utilisée, auquel est éventuellement joint le code ISO de la région. Ces trois informations sont séparées les unes des autres par le caractère underscore. Enfin, le nom d'un fichier texte exploité par un PropertyResourceBundle doit posséder l'extension *.properties*. Ainsi, les éléments de ResourceBundle présentés ci-dessus peuvent être stockés dans un fichier texte nommé **MesLabels_fr_FR.properties**. Un tel nom montre que les éléments sont membres de la famille *MesLabels*, et concernent des expressions françaises utilisées en France. Les éléments de la même famille, valables pour l'anglais parlé au Royaume Uni, doivent être déclarés dans un fichier texte nommé **MesLabels_en_GB.properties**, dont le contenu peut être le suivant :

```
titre1=Hello, you're welcome on this web site!
bouton1=Login
francais1=French version
anglais1=English version
```

- Faire une déclaration à l'aide d'un ListResourceBundle.

L'usage d'un PropertyResourceBundle peut s'avérer être intéressant, du simple fait de sa simplicité d'utilisation. Cependant, les valeurs des différents éléments d'un ResourceBundle de ce type sont obligatoirement des chaînes de caractères. Pour éliminer cette contrainte, en offrant la possibilité de définir des valeurs d'un type quelconque, il convient de faire usage de la classe ListResourceBundle.

S'agissant d'une classe abstraite, il est nécessaire d'en constituer une classe dérivée dont le nom respecte la convention de nommage expliquée précédemment pour les PropertyResourceBundle, et d'implémenter la méthode getContents dont voici la signature :

```
protected abstract Object[][] getContents()
```

Cette méthode doit renvoyer un tableau bidimensionnel, dont chaque élément est une paire d'instances issues de la classe Object. En réalité, les éléments du tableau renvoyé doivent correspondre aux éléments du ResourceBundle : la première dimension du tableau identifie la clé de l'élément du ResourceBundle et doit donc être de type String. La seconde dimension représente la valeur de l'élément du ResourceBundle, qui peut effectivement être d'un type quelconque.

Si l'on souhaite constituer le ResourceBundle nommé MesLabels_fr_FR grâce à un ListResourceBundle, il faut donc

© ENI Editions - All rigths reserved

développer une classe dont le nom est identique à celui du ResourceBundle et dont le contenu peut être le suivant :

```
package premierProjetJSF;

import java.awt.Button;
import java.util.ListResourceBundle;

public class MesLabels_fr_FR extends ListResourceBundle {

     protected Object[][] getContents() {
          //Définition des éléments.
          Object[][] lesElements=
                    {
                    {"titre1","Bonjour, vous êtes les bienvenus
                         sur notre site!"},
                    {"bouton1","Se connecter"},
                    {"francais1","Version française"},
                    {"anglais1","Version anglaise"},
                    {"bouton2",new Button("Se déconnecter")}
                    };
          return lesElements;
     }
}
```

L'implémentation de la méthode getContents réalise la déclaration de l'ensemble des éléments évoqués précédemment pour ce ResourceBundle. Un nouvel élément a cependant été ajouté : il dispose de la clé nommée bouton2 et possède une valeur de type java.awt.Button représentant un bouton d'interface basée sur la couche graphique AWT.

b. Utilisation dans le cadre de la création d'applications web

De même que les applications web basées sur le langage Java exploitent toutes une instance particulière de la classe Locale, celles-ci peuvent rechercher si un ResourceBundle est disponible par défaut pour le Locale utilisé. Lorsque celui-ci existe, il peut être stocké dans la session, de manière à rester disponible durant toute la visite du site par l'internaute. La librairie standard de balises JSP prévoit par ailleurs la balise <fmt:setBundle> permettant de surcharger le ResourceBundle par défaut, pour en proposer un autre, plus adapté aux attentes.

Dans le cadre des applications web exploitant des composants JSF, un ResourceBundle peut être instancié par l'usage de la balise <f:loadBundle>, issue de la bibliothèque Core. Il est également possible de déclarer des *ResourceBundle* au sein du fichier de configuration des ressources de l'application. Ce point particulier est abordé à la section Déclaration de ResourceBundle associés aux fichiers de libellés de ce chapitre.

3. La classe NumberFormat

a. Description

La classe abstraite NumberFormat est fournie dans le package standard *java.text*. Elle permet d'obtenir un formatage des nombres adapté à une langue et une région géographique particulière. Plusieurs méthodes statiques peuvent être utilisées pour instancier des objets issus de cette classe abstraite. Parmi ces méthodes, les plus importantes sont définies par les signatures suivantes :

```
public static final NumberFormat getInstance()
public static NumberFormat getInstance(Locale inLocale)
public static final NumberFormat getCurrencyInstance()
public static NumberFormat getCurrencyInstance(Locale inLocale)
public static final NumberFormat getPercentInstance()
public static NumberFormat getPercentInstance(Locale inLocale)
```

Ces méthodes disposent d'un rôle spécifique suivant le type de données à formater : getCurrencyInstance fournit un objet NumberFormat capable de s'occuper du formatage des nombres correspondant à des devises et monnaies ; getPercentInstance en instancie un autre en mesure d'assurer le formatage des nombres exprimant des pourcentages. Enfin, la méthode getInstance renvoie un objet NumberFormat prenant en charge les autres types de données.

L'argument inLocale de type Locale, présent dans certaines des signatures de méthodes précédentes, sert à indiquer la langue et la région par rapport auxquelles le formatage est demandé. Lorsque cet argument n'est pas

© ENI Editions - All rigths reserved

mentionné, les paramètres définis localement sur le poste de travail de l'utilisateur sont utilisés pour assurer le formatage.

b. Utilisation dans le cadre des applications web

Le formatage des valeurs numériques selon les usages locaux est bien évidemment d'une importance capitale dans les applications web, en particulier pour toutes celles concernées par le e-commerce.

Les deux balises `<fmt:formatNumber>` et `<fmt:parseNumber>`, issues de la bibliothèque standard *JSTL fmt*, peuvent respectivement être utilisées dans les pages JSP pour formater et réaliser une analyse syntaxique de valeurs numériques qui leur sont fournies en argument sous la forme d'une chaîne de caractères.

Avec la technologie Java Server Faces, le formatage des valeurs numériques se fait au travers de l'usage de convertisseurs associés à des composants graphiques. Ces convertisseurs sont restitués par des balises `<f:convertNumber>` dont l'attribut `type` permet de préciser si le formatage concerne des devises, des pourcentages ou des nombres d'autres types.

4. La classe DateFormat

a. Description

La classe abstraite `DateFormat`, définie dans le package *java.text*, permet d'instancier des objets capables de formater des informations liées à la date et aux horaires, conformément aux habitudes spécifiques d'une région particulière. Ces objets peuvent s'obtenir en constituant des classes dérivées de `DateFormat` (la spécification en définit une, nommée `SimpleDateFormat`). Il est également possible d'en instancier par invocation de méthodes statiques disponibles dans la classe `DateFormat`. Voici les principales signatures de ces méthodes :

```
public static final DateFormat getDateInstance()
public static final DateFormat getDateInstance(int style)
public static final DateFormat getDateInstance(int style,
                                               Locale aLocale)
public static final DateFormat getDateTimeInstance()
public static final DateFormat getDateTimeInstance(int dateStyle,
                                                   int timeStyle)
public static final DateFormat getDateTimeInstance(int dateStyle,
                                                   int timeStyle,
                                                   Locale aLocale)
public static final DateFormat getTimeInstance()
public static final DateFormat getTimeInstance(int style)
public static final DateFormat getTimeInstance(int style,
                                               Locale aLocale)
```

La méthode `getDateInstance` sert à obtenir objet `DateFormat` spécialisé dans le formatage de la date. Elle requiert potentiellement la précision d'un paramètre *style* indiquant sous quelle forme la date en question doit être présentée (courte, longue, ou intermédiaire). Il est également possible de mentionner le pays et la langue pour lesquels le formatage est demandé, par l'utilisation d'un paramètre aLocale de type `Locale`. Si ce dernier paramètre n'est pas fourni, les informations de localisation définies par défaut sur le poste de travail de l'utilisateur sont exploitées.

La méthode `getDateTimeInstance` permet de créer un objet `DateFormat` capable de formater simultanément la date et l'heure. Elle nécessite la précision éventuelle de deux arguments, `dateStyle` et `timeStyle`, dont la valeur permet d'indiquer sous quelle forme la date et l'heure doivent être présentées (courte, longue, ou intermédiaire). L'indication de localisation peut se faire par l'intermédiaire de l'argument `aLocale`.

Enfin, la méthode `getTimeInstance` présente des arguments dont les noms et les rôles sont identiques à ceux décrits pour la méthode `getDateInstance`. La seule spécificité de `getTimeInstance` réside dans le fait qu'elle assure le formatage de l'heure uniquement.

b. Utilisation dans le cadre des applications web

Comme pour les objets de type `ResourceBundle` et `NumberFormat`, il est possible d'exploiter des instances de la classe `DateFormat` depuis les pages JSP d'une application web. Cela peut se faire en utilisant certaines balises définies dans la bibliothèque *JSTL fmt* : il s'agit des balises `<fmt:formatDate>` et `<fmt:parseDate>`. La première permet de réaliser le formatage d'une date fournie en paramètre, alors que la seconde assure l'analyse syntaxique d'une chaîne de caractères représentant une date.

La spécification JSF s'appuie, quant à elle, sur l'usage des convertisseurs pour réaliser le formatage des dates et des

© ENI Editions - All rigths reserved

heures. La bibliothèque *JSF Core* contient pour cela une balise `<f:convertDateTime>`, dont on peut préciser le comportement par le biais de différents attributs, notamment `type` et `pattern`.

Reportez-vous au chapitre Exploitation des composants standards JSF de cet ouvrage pour obtenir de plus amples renseignements concernant l'usage du convertisseur de type `DateTimeConverter`.

© ENI Editions - All rigths reserved

Externalisation des libellés et des messages dans une application web

Dans le cadre de la conception d'applications web basées sur la technologie JSF, la mise en œuvre de l'internationalisation et de la localisation est assurée par l'exploitation d'objets de type `PropertyResourceBundle`. Les libellés et autres messages spécifiques à une famille de `ResourceBundle` sont donc inscrits dans un ensemble de fichiers de type texte, chacun d'eux étant relatif à une langue et une région particulière.

Cette section montre comment il est possible, sur un plan pratique, d'exploiter le mécanisme d'internationalisation : il précise la méthode de déclaration de nouveaux `ResourceBundle` associés à des fichiers de libellés, et propose une illustration de sélection dynamique du langage, selon les souhaits de l'internaute.

1. Définir plusieurs fichiers de libellés

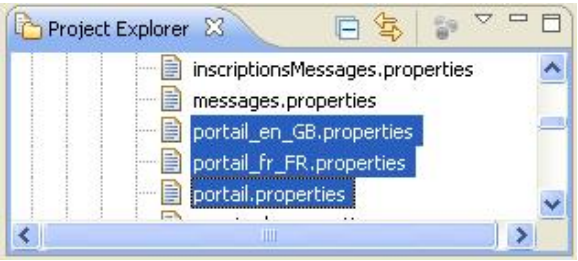
La première étape du travail consiste à élaborer autant de fichiers de libellés que de langues à proposer aux internautes. Ces fichiers doivent être placés préférentiellement dans le sous-répertoire *src* de l'application web. Chaque ligne de ces fichiers contient une paire d'informations représentant la clé et la valeur d'un libellé ou d'un message particulier. Les fichiers d'une même famille de `ResourceBundle` présentent tous les mêmes clés, mais les valeurs de celles-ci diffèrent bien sûr selon la langue. Il est naturellement possible de mettre en place plusieurs familles de `ResourceBundle` dans une même application web.

Tous les fichiers de libellés ou de messages doivent respecter la convention de nommage requise. À défaut, ils ne pourront pas être pris en charge par l'application web. Leur nom doit donc être composé de l'identifiant de la famille, de celui de la langue, et éventuellement de celui de la région. Ils doivent en outre disposer de l'extension *.properties*.

Des exemples de contenu de fichiers de libellés sont présentés dans les sections précédentes de ce chapitre.

2. Utilisation de la langue définie par défaut dans le navigateur

L'image suivante montre le contenu de la vue d'exploration de projets d'Eclipse, pour une application web spécifique :



Plusieurs familles de `ResourceBundle` y sont définies. L'une d'elles, nommée *portail*, permet la prise en charge de l'anglais parlé au Royaume Uni et du français parlé en France. De fait, tous les navigateurs pour lesquels le langage par défaut correspond à l'une de ces langues seront en mesure de proposer à l'internaute des libellés qui lui sont adaptés.

Qu'en est-il pour les autres situations, dans lesquelles le navigateur n'est pas configuré pour exploiter les langues proposées par l'application web ? Il s'agit en effet d'une situation courante, dans la mesure où le concepteur du site est dans l'impossibilité de définir autant de fichiers de libellés qu'il n'y a de dialectes dans le monde. La réponse à cette problématique consiste à créer un fichier de libellés *par défaut* pour chaque famille de `ResourceBundle`. Le nom de ce fichier par défaut est uniquement composé du nom de la famille et de l'extension *.properties*. L'illustration précédente montre que le fichier de libellés par défaut de la famille *portail* est effectivement prévu : son nom est *portail.properties*.

3. Permettre l'exploitation de ces fichiers dans l'application web

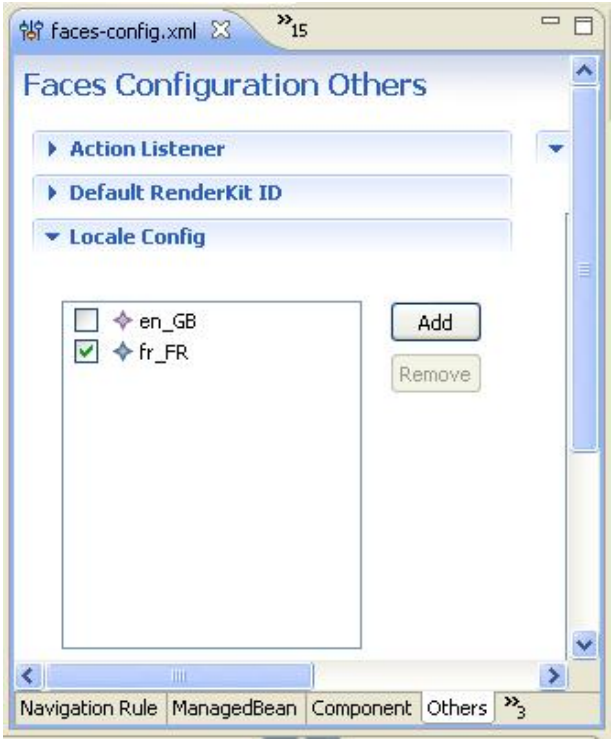
La seule présence des multiples fichiers de libellés dans le sous-répertoire **src** de l'application web ne suffit pas à les rendre exploitables dans les pages web. Il faut en effet préalablement indiquer les dialectes supportés par l'application, et permettre l'instanciation d'un objet de type `ResourceBundle` pour chacune des familles définies par les fichiers de libellés. Ces deux opérations peuvent être assurées par la modification du fichier de configuration des ressources de l'application web.

a. Les dialectes supportés

Après avoir affiché le fichier de configuration des ressources dans la zone d'édition de l'environnement Eclipse, la

© ENI Editions - All rigths reserved

déclaration des dialectes supportés peut se faire depuis la rubrique **Locale Config** située dans l'onglet **Other**. Il est possible de définir une liste de dialectes supportés, et de préciser celui à utiliser par défaut en cochant la case qui lui correspond.



Les changements sont naturellement visibles dans l'onglet **Source** présentant le code source du fichier de configuration des ressources. Les dialectes sont définis par le biais d'une balise `<locale-config>`, sous-élément de la balise `<application>`. Le dialecte par défaut est mentionné par la valeur d'un élément `<default-locale>`, alors que chaque dialecte supplémentaire est défini par une balise `<supported-locale>`.

```
<faces-config>
     <application>
          <locale-config><default-locale>fr_FR</default-locale>
               <supported-locale>en_GB</supported-locale>
               <supported-locale>de_DE</supported-locale>
          </locale-config>
     </application>
     ...
</faces-config>
```

b. Déclaration de ResourceBundle associés aux fichiers de libellés

Deux moyens sont proposés pour déclarer et utiliser de nouveaux `ResourceBundle` associés à des fichiers de libellés : la déclaration au sein des pages web, et la déclaration en tant que nouvelles ressources de l'application web.

Déclaration au sein des pages web

Dans cette situation, les `ResourceBundle` sont instanciés directement dans les pages web par le bais d'une balise `<f:loadBundle>` : ils disposent alors d'une portée de type `request`. Si un tel mode d'utilisation peut convenir dans le cadre d'applications web limitées, il présente une contrainte importante pour les applications plus lourdes, dans la mesure où la balise `<f:loadBundle>` doit être systématiquement répétée dans l'ensemble des pages web qui exploite un `ResourceBundle` particulier.

Le code source suivant illustre la déclaration et l'utilisation d'un `ResourceBundle` au sein d'une page JSP représentant un formulaire d'authentification :

```
<f:view>
     <f:loadBundle
          basename="premierProjetJSF.messages"
          var="mesMessagesPerso"/>
     <h:form>
          <h:panelGrid border="1" columns="3">
```


© ENI Editions - All rigths reserved

```
        <h:outputText value="#{mesMessagesPerso.clientcode}">
        </h:outputText>
        <h:inputText id="champCodeClient"></h:inputText>
        <h:message for="champCodeClient" id="messageCodeClient">
        </h:message>
        <h:outputText value="#{mesMessagesPerso.password}">
        </h:outputText>
        <h:inputSecret id="champMotPasse">
        <f:validator validatorId="monValidateurDeMotDePasse" />
        </h:inputSecret>
        <h:message for="champMotPasse" id="messageMotPasse">
        </h:message>
        </h:panelGrid>
        <h:commandButton id="btnLogin"
             action="#{loginBean.verificationAuthentification}"
             value="#{mesMessagesPerso.login}>
        </h:commandButton>
    </h:form>
</f:view>
```

La balise `<f:loadBundle>` présente un attribut `basename` permettant de préciser le nom de famille du `ResourceBundle` à instancier. Quant à l'attribut `var` de la même balise, il sert à attribuer un identifiant à ce `ResourceBundle`, pour le rendre disponible dans le reste de la page web.

Les étiquettes associées aux deux champs de saisie et au bouton de soumission sont récupérées dans le fichier de libellés correspondant au `ResourceBundle` nommé `mesMessagesPerso`, pour la langue et la région définies par défaut dans le navigateur web. Le fichier en question possède au moins trois éléments dont les clés sont nommées `clientcode`, `password` et `login`. Voici un contenu possible pour ce fichier, ainsi qu'une illustration du résultat obtenu lors de l'exécution de la page :

```
name=Nom
password=Mot de passe
clientcode=Code client
welcome=Bienvenue!
submit=Validez
login=Se connecter
```

Déclaration en tant que nouvelle ressource de l'application

Par ce second moyen, les `ResourceBundle` associés aux fichiers de libellés disposent d'une portée plus importante (session, application) : ils peuvent alors être utilisés dans la totalité des pages web de l'application par l'intermédiaire d'un identifiant unique, sans qu'il soit nécessaire d'avoir recours à la balise `<f:loadBundle>`.

Lorsque le fichier de configuration des ressources est présenté dans la zone d'édition, on constate qu'il n'existe pas de formulaire permettant de déclarer spécifiquement de nouveaux `ResourceBundle` : il est donc nécessaire de modifier directement le code source du fichier **faces-config.xml** pour y ajouter les balises indispensables à ce type de déclaration. Un `ResourceBundle` se caractérise par deux informations : son nom de famille et son identifiant unique.

```
<faces-config>
     <application>
          <resource-bundle><base-name>premierProjetJSF.portail</base-name>
               <var>portail_web</var>
          </resource-bundle>
          ...
     </application>
     ...
```

© ENI Editions - All rigths reserved

```
</faces-config>
```

Comme présenté dans le code ci-dessus, un `ResourceBundle` se déclare à l'aide d'une balise `<resource-bundle>` présentant deux sous-éléments `<base-name>` et `<var>` : le premier permet de mentionner le nom de famille du `ResourceBundle`, alors que le second sert à indiquer l'identifiant unique de cet objet. Dans l'exemple proposé, le nom de famille est portail : il correspond à des fichiers de libellés situés dans le sous-répertoire **src/premierProjetJSF** de l'application web. L'identifiant *portail_web* pourra être utilisé dans les pages web de l'application pour faire référence à ce `ResourceBundle` particulier.

4. Permettre à l'utilisateur de sélectionner dynamiquement un langage

Une application intéressante de l'internationalisation consiste à permettre à l'internaute de sélectionner dynamiquement la langue dans la laquelle il souhaite visualiser les informations proposées par l'application web. Voyons comment obtenir un tel résultat, au travers d'un exemple simple.

La démarche à suivre peut être la suivante :

- Écrire le contenu des fichiers de libellés pour chacune des langues supportées par l'application.
- Construire une page JSP chargée d'afficher les informations et présentant un ou plusieurs composants JSF permettant de choisir la langue à utiliser.
- Développer un bean managé dont l'une des méthodes est chargée de modifier dynamiquement la langue à utiliser dans le contexte de l'application web.

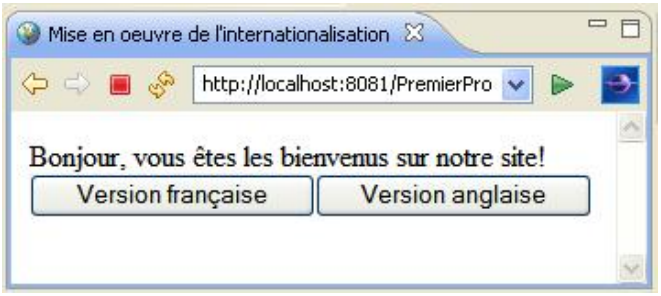
a. Contenu des fichiers de libellés

Cet aspect du travail ne pose aucun problème. La section Mécanismes mis en œuvre dans le cadre du multilinguisme - La classe ResourceBundle et ses dérivées de ce chapitre présente les contenus de deux fichiers nommés **MesLabels_fr_FR.properties** et **MesLabels_en_GB.properties** : ceux-ci sont parfaitement adaptés au cas de figure à développer ici.

b. Construire la page JSP

Voici un exemple de page web contenant des composants JSF faisant référence à un `RessourceBundle` nommé `portailWeb`, déclaré dans le fichier de configuration des ressources de l'application.

```
<f:view>
     <h:form>
          <h:outputText value="#{portailWeb.titre1}">
          </h:outputText><br>
          <h:commandButton
          value="#{portailWeb.francais1}"
          id="fr_FR"
          actionListener="#{beanInternationalisation.choixLangue}">
          </h:commandButton>
          <h:commandButton
          value="#{portailWeb.anglais1}"id="en_GB"actionListener="#{beanInternationalisation.choixLangue}">
          </h:commandButton>
     </h:form>
</f:view>
```



Cette page comporte un champ de type `HtmlOutputText` dont la valeur correspond à la clé *titre1* d'un élément de

© ENI Editions - All rigths reserved

portailWeb. De même, les labels affichés sur les deux boutons correspondent aux clés *français1* et *anglais1* de ce ResourceBundle.

Deux points importants sont à noter à propos de ces boutons : tout d'abord, les balises qui les représentent possèdent un attribut id dont la valeur identifie une langue et une région particulières, selon la codification ISO. De plus, ces balises présentent également un attribut actionListener, dont la valeur référence la méthode choixLangue d'un bean managé nommé beanInternationalisation : cette méthode est donc invoquée lors d'un clic sur les boutons. Celle-ci a pour but de modifier le langage utilisé pour l'affichage des informations.

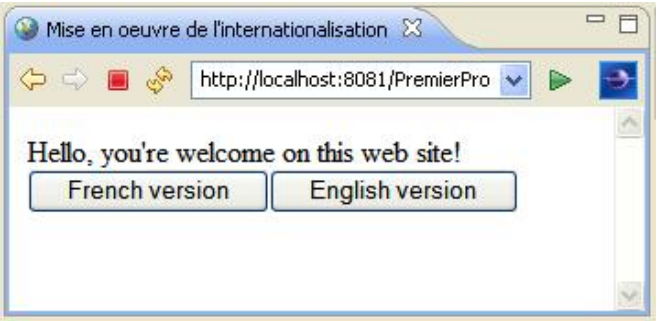
c. Définir une classe contenant une méthode chargée de modifier le langage par défaut

La dernière étape de la démarche consiste précisément à développer le bean managé contenant la méthode choixLangue. Cette méthode doit naturellement accepter un objet de type ActionEvent en argument afin de pouvoir prendre en charge le clic sur des composants de type HtmlCommandButton.

```
package premierProjetJSF;

import java.util.Locale;

import javax.faces.context.FacesContext;
import javax.faces.event.ActionEvent;

public class BeanInternationalisation {
     public void choixLangue(ActionEvent actionEvent)
     {
          Locale locale;
          String idComposant=actionEvent.getComponent().getId();

          //Si l'identifiant du composant est constitué
          //de la langue et du pays,
          //ces deux informations sont séparées.
          int index=idComposant.indexOf("_");
          if (index>0)
          {
               String langue=idComposant.substring(0,index);
               String pays=idComposant.substring(index+1);
               locale=new Locale(langue,pays);
          }
          else
          {
          locale=new Locale(idComposant);
          }

FacesContext.getCurrentInstance().getViewRoot().setLocale(locale);
     }
}
```

Ces instructions illustrent la création d'une instance de la classe Locale à partir de l'identifiant du composant JSF à l'origine du clic. Cette instance est finalement utilisée pour mettre à jour la localisation dans l'application web.

À l'exécution, on constate bien un basculement entre les deux langues (anglais et français) par l'utilisation du bouton adapté.



© ENI Editions - All rigths reserved

Formatage des nombres et des dates

Comme précisé précédemment dans ce chapitre, le formatage des nombres et des dates, selon les conventions propres à une langue ou une région, est assuré par les composants JSF de type `NumberConverter` et `DateTimeConverter`. Ces composants doivent être associés à d'autres composants graphiques, et sont utilisables dans les pages web grâce à l'exploitation des balises `<f:convertNumber>` et `<f:convertDateTime>`.

Les sections Conversion des valeurs numériques et Conversion de la date et de l'heure du titre Exploitation des convertisseurs standards du chapitre Exploitation des composants standards JSF de cet ouvrage présentent dans le détail le mode de fonctionnement de ces convertisseurs, et proposent des exemples pouvant tenir lieu d'illustrations dans le cadre de l'internationalisation. Le lecteur intéressé par cette thématique peut donc se référer à ces sections pour obtenir les renseignements qu'il recherche.

© ENI Editions - All rigths reserved