20/06/2017

Victor Vogel ; Cédric Welty

Rapport LO54

Notice PrimeFaces

Table des matières

[I. Introduction 2](#_Toc485827199)

[A. JavaServer Faces 2](#_Toc485827200)

[B. PrimeFaces 2](#_Toc485827201)

[II. Tutoriel 2](#_Toc485827202)

[A. Intégration 2](#_Toc485827203)

[B. Première approche 3](#_Toc485827204)

[C. Composition 3](#_Toc485827205)

[D. Composants 4](#_Toc485827206)

[III. Expérience personnelle 7](#_Toc485827207)

[A. Problèmes rencontrés 7](#_Toc485827208)

[B. Attraits de la technologie 7](#_Toc485827209)

[IV. Bilan 7](#_Toc485827210)

# Introduction

## JavaServer Faces

Actuellement dans sa version 2.3, JavaServer Faces (abrégé JSF) est un framework Java EE servant au développement d'applications web qui se base sur une notion de composants. L’idée est que l’état de chaque composant est enregistré au rendu de la page puis restauré au retour de la requête.

## PrimeFaces

Uniquement doté de deux jeux de composants standard, JSF seul peut s'avérer assez limité. Pour contrer à cela il existe un certain nombre de jeux de composants additionnels, dont PrimeFaces en est le principal représentant. PrimeFaces, en version 6.1 depuis Janvier 2017, est développé par PrimeTek qui le distribue sous licence Apache 2.0. De par ses qualités autant au niveau esthétique des composants, richesse de la bibliothèque, ou simplicité d’implémentation, PrimeFaces est devenu une solution viable pour développer la couche présentation.

# Tutoriel

## Intégration

<**dependency**>  
 <**groupId**>org.primefaces</**groupId**>  
 <**artifactId**>primefaces</**artifactId**>  
 <**version**>6.1</**version**>  
</**dependency**>

L'intégration du composant PrimeFaces peut se faire via Maven :

<**dependency**>  
 <**groupId**>com.sun.faces</**groupId**>  
 <**artifactId**>jsf-api</**artifactId**>  
 <**version**>2.2.2</**version**>  
</**dependency**>  
<**dependency**>  
 <**groupId**>com.sun.faces</**groupId**>  
 <**artifactId**>jsf-impl</**artifactId**>  
 <**version**>2.2.2</**version**>  
</**dependency**>

Ce composant étant dépendant de JSF, son intégration est également nécessaire :

## Première approche

PrimeFaces fonctionne sur la base de JSF, nous utiliserons donc des fichiers de type \*.jsf ou \*.xhtml. L'architecture d'une simple page affichant un traditionnel "Hello World!" se présentera comme suit :

**<!DOCTYPE html*>***<**html xmlns="http://www.w3.org/1999/xhtml"  
 xmlns:h="http://java.sun.com/jsf/html"  
 xmlns:c="http://java.sun.com/jsp/jstl/core"  
 xmlns:f="http://java.sun.com/jsf/core"  
 xmlns:ui="http://java.sun.com/jsf/facelets"  
 xmlns:p="http://primefaces.org/ui"**>  
  
 <**h:head**>  
 </**h:head**>  
  
 <**h:body**>  
 <**h:outputText value="Hello World!"** />  
 </**h:body**>  
</**html**>

## Composition

Afin de réaliser un site internet avec un template de page récurrent, JSF propose un système de balises de compositions. Ainsi, on peut aisément définir un fichier template qui contiendra toute l'architecture de base de nos pages en indiquant l'emplacement du contenu par des balises *<ui:insert>.* La page devant être affichée pour alors être définie dans une balise *<ui:composition>* puis par des *<ui:define>* pour chaque partie à définir. L'attribut *target* dans certaines balises permet également d'indiquer l'emplacement où JSF devra replacer la balise. Notre page "Hello World!" ressemblera alors à ça :

***template.xhtml***

**<!DOCTYPE html*>***<**html xmlns="http://www.w3.org/1999/xhtml"  
 xmlns:h="http://java.sun.com/jsf/html"  
 xmlns:ui="http://java.sun.com/jsf/facelets"**>  
  
 <**h:head**>  
 <**ui:insert name="header"** />  
 </**h:head**>  
  
 <**h:body**>  
 <**ui:insert name="content"** />  
 </**h:body**>  
</**html**>

***index.xhtml***

<**ui:composition xmlns="http://www.w3.org/1999/xhtml"  
 xmlns:ui="http://java.sun.com/jsf/facelets"  
 xmlns:h="http://java.sun.com/jsf/html"  
 template="/template.xhtml"**>  
  
 <**ui:define name="content"**>  
 <**h:outputText value="Hello World!"** />  
 </**ui:define**>  
  
</**ui:composition**>

## Composants

PrimeFaces propose plus d’une centaine de composants, de complexités diverses, néanmoins l’ajout d’un composant reste plutôt simple puisqu’il ne nécessite que l’ajout du composant lui-même dans le xHTML avec quelques tags pour le configurer ainsi que l’implémentation de méthodes spécifiques dans la managed bean. Voici quelques exemples d'implémentation :

**Chips**

<**p:chips id="searchChips" value="#{**searchBean.chips**}" onkeyup="if**(!(event.keyCode !== 8 || $(**this**)[0].value.length !== 0)) { hiddenSubmitButton(); search(); } **"**>  
 <**f:ajax event="itemSelect" render="@form" onevent="search()"**/>  
 <**f:ajax event="itemUnselect" render="@form" onevent="search()"**/>  
</**p:chips**>

<**p:remoteCommand name="hiddenSubmitButton" update="searchForm"**/>

Le composant *chips* peut être très simple d'utilisation pour peu qu'il soit dans un formulaire et avec un bouton de validation. À ce moment-là, une simple balise <p:chips> avec un tag *value* associé à un bean suffit pour avoir des puces fonctionnelles. Seulement, dans notre cas nous avons souhaité une mise à jour dynamique en fonction de l'ajout ou de la suppression d'une puce. Pour ce faire, un certain nombre de modifications ont été nécessaires. Tout d'abord l'utilisation des événements *itemSelect* et *itemUnselect* qui sont levés respectivement lors de l'ajout et du retrait d'une puce. Ces événements sont à utiliser dans des balises *ajax* à l'intérieur de la balise *chips*. La levée d'un de ces événements activera alors ajax et lui demandera de mettre à jour une partie du DOM (ici le formulaire) via le tag *render* et d'exécuter une fonction JavaScript grâce au tag *onevent*. Seulement, l'événement de suppression d'une puce n'est pas levé si celle-ci est supprimée via un appui sur la touche retour arrière. Il a donc fallu gérer ce cas particulier via du code JavaScript qui, sur l'appui de la touche retour arrière, va appeler un composant PrimeFaces qui va forcer la mise à jour du formulaire.

La gestion des valeurs du composant se fait via un *bean*, c’est-à-dire un composant JavaServer Faces qui permettra une communication entre le client et le serveur. Ce composant se présente sous cette forme :

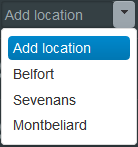
Il s'agit d'une simple classe Java munie d'une annotation *@ManagedBean* et d'une annotation de scope (ici *@ViewScoped*). Cette classe va contenir des variables munies d'accesseurs, ce qui permet au client de récupérer ou d'attribuer ses valeurs.

@ManagedBean  
@ViewScoped  
**public class** SearchBean {  
  
 **private** List<String> **chips** = **new** ArrayList<>();  
  
 **public** List<String> getChips() {  
 **return chips**;  
 }  
  
 **public void** setChips(List<String> chips) {  
 **if**(chips != **null**) {  
 **this**.**chips** = chips;  
 }  
 **else** {  
 **this**.**chips**.clear();  
 }  
 }  
}

**SelectOneMenu**

<**p:selectOneMenu id="location" value="#{searchBean.location}"**>  
  
 <**f:selectItem** **itemLabel="#{msg['search.addLocation']}"** **itemValue=""**/>

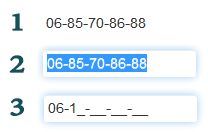
<**f:selectItems** **value="#{searchBean.getAvailableLocations()}"**/>  
  
</**p:selectOneMenu**>

L’implémentation d’un menu déroulant pour effectuer un choix unique se fait à peu près de cette manière. La balise *selectOneMenu* annonce l’utilisation du composant en question, elle possède ici deux tags : *id* qui correspond à son identifiant et *value* qui indique ici la variable liée. La première sous-balise, *selectItem* créé le choix *Add location* que l’on voit sur la gauche. Ensuite la deuxième sous-balise va permettre de remplir le menu déroulant avec les valeurs d'emplacements disponibles. Ici, la requête sur les emplacements est effectuée auprès du bean *searchBean* qui retourne une liste de String, contenant dans notre cas Belfort, Sévenans et Montbéliard. Dans cet exemple, la sélection d’un item mettra à jour la variable location de type String du *searchBean*. Par conséquent, la sélection de *Add location* revient à réinitialiser la variable.

**Inplace associé à un inputMask**

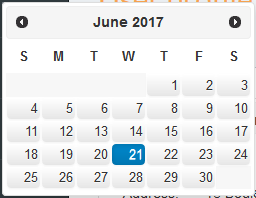
<**p:inplace id="phone" label="#{clientManagementBean.getPhone()}"**

**effectSpeed="fast" event="dblclick"**>  
  
 <**p:inputMask** **value="#{clientManagementBean.phone}" mask="99-99-99-99-99"**/>  
  
</**p:inplace**>

L’*inplace* est un composant qui permet d’afficher du texte au même titre qu’un label, mais également d’éditer ce texte grâce à un évènement. Dans cet exemple, l’*inplace* a pour id 'phone', il affiche la valeur retournée par la méthode *getPhone()* du *clientManagementBean* (visible en 1). L’évènement d’édition est défini grâce au tag *event*: un double clic. Ainsi en effectuant un double clic sur le numéro en 1 l’affichage se transforme en celui présenté en 2. C’est ici qu’intervient l’*inputMask*, qui force l’input à être formaté suivant un certain pattern. Dans ce cas, le pattern est défini par le tag *mask*, et demande une série de chiffre, les tirets étant automatiquement ajoutés comme présenté en 3. La valeur résultant de cette édition sera retournée sur l’attribut phone du bean.

**Calendar**

<**p:calendar id="date" value="#{searchBean.date}" showOn="button"/**>

Le calendrier est un composant qui permet de choisir une date. Même si le principe est simple, la création d’un calendrier à partir de rien ne l’est pas pour autant. Grâce à PrimeFaces, il suffit simplement d’ajouter la balise ci-dessus pour avoir le calendrier présenté à droite. Grâce au tag *showOn*, on définit que le calendrier n’est pas ouvert pas défaut, mais est représenté sous forme d’un bouton (à gauche). Au clic sur ce bouton, le calendrier apparait. La date sur laquelle l’utilisateur clique sera envoyée sur la variable date de searchBean.

# Expérience personnelle

## Problèmes rencontrés

Au cours de ce projet, un certain nombre de problèmes liés à la technologie JSF / PrimeFaces ont été rencontrés. Tout d'abord, il faut savoir que JSF va interpréter les pages JSF/XHTML et les retranscrire en langage HTML/JavaScript. Cependant, absolument tout est retranscrit, commentaires inclus. Ainsi, il peut arriver qu'une ligne commentée bouleverse totalement le rendu final car interprétée par JSF.

Aussi, certains comportements sont parfois difficiles à implémenter. Notre projet nécessitait un certain nombre de changements dynamiques à la sélection d'une valeur dans une liste déroulante ou un champ texte. Seulement, sans submit explicite du formulaire, cette mécanique de mise à jour peine à fonctionner. Ainsi, malgré des composants censés intégrer Ajax et permettre des événements servant justement à ce type de situations, il a fallu à de nombreuses reprises "tricher" en forçant la main à Ajax.

Enfin, cette version de JSF présente certains bugs connus. Nous avons notamment rencontré un bug qui apparaissait dans le cas particulier où l'on essayait de se connecter/déconnecter après avoir changé la langue d'affichage. En effet, lors de la mise à jour du formulaire de connexion, celui-ci perd la capacité à appeler une action côté serveur. Il a été possible de résoudre ce problème grâce à un script trouvé sur internet.

## Attraits de la technologie

La technologie JSF et sa bibliothèque de composants additionnels PrimeFaces comportent nombre d'avantages. En effet, il n'a jamais été aussi simple d'intégrer à l'interface client des valeurs et des actions issues du serveur. De plus, PrimeFaces embarque nombre de composants intéressants et relativement simples d'utilisation mais aussi compréhensibles et peu coûteux en terme de programmation. De plus les composants proposés sont modernes et les designs agréables.

# Bilan

Malgré certains défauts (qui de plus sont amenés à être corrigés dans les versions à venir), la technologie JSF est intéressante. Elle est un choix tout à fait envisageable pour la conception d'une application web professionnelle.