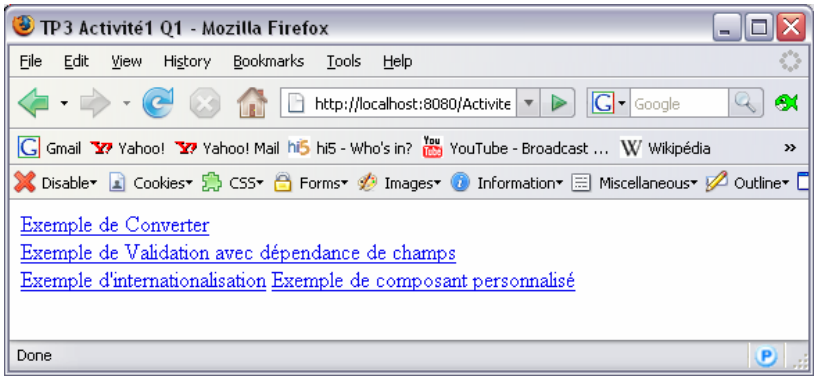
Atelier 3 C

JSF – Aspects avancés

Au cours de cet atelier, nous allons aborder la technologie JSF, qui est, de nos jours, entrain de gagner du terrain et prendre de l'avantage sur la technologie Struts. Nous allons toucher aux aspects et aux facilités apportés par cette technologie, à savoir les contrôles, les validations, l'internationalisation, et le développement de composants personnalisés._Nous avons organisé ces différents exemples suivant un menu permettant de tester les différentes réalisations. Voici un aperçu :



Programmation de Convertisseurs personnalisés :

Dans la section suivante nous allons nous intéresser aux convertisseurs (Converters) qui font partie de la librairie JSF. A la section qui suit, nous allons présenter comment satisfaire notre propre code de validation.

La conversion standard (standard converters) : conversion de dates et de nombres

Une application web enregistre des données de différents types, mais l'interface utilisateur web utilise exclusivement le type chaîne de caractères (String).
Supposons que l'utilisateur éprouve le besoin un objet Date qui est enregistré dans la logique métier. En premier lieu, la date est convertie en une chaîne qui va être envoyée au browser du client afin d'être affichée dans un champ de texte. L'utilisateur modifie alors la date et le résultat est retourné au serveur où il serait reconvertit en objet date. La même situation peut se présenter au types entier, double précision ou booléen. Grâce aux Converters, la valeur récupérée dans le champ de texte sera convertie en date avant l'envoi par le container JSF.

Soit l'application suivante :

On considère un formulaire d'ajout de facture. La facture est caractérisée par un montant, un numéro de carte client (acheteur) et une date. Nous utiliserons des converters pour le montant et la date.

Nous attachons le premier champ au montant et nous indiquons qu'au minimum on a 3 chiffres après la virgule.

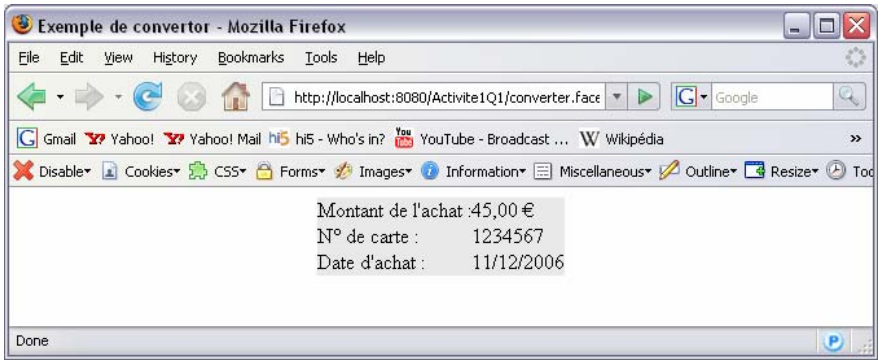
```
<h:inputText id="montant" value="#{achatBean.montant}" required="true">
      <f:convertNumber minFractionDigits="3"/>
</h:inputText>
```

Le convertisseur « f:convertNumber » est l'un des convertisseur standard supportés par l'implémentation.

En ce qui concerne la date, nous avons imposé un schéma (pattern) qui est le suivant : jj/mm/aaaa :

```
<h:inputText id="date" value="#{achatBean.date}" required="true">
      <f:convertDateTime pattern="dd/MM/yyyy"/>
</h:inputText>
```

Voici le résultat que nous obtenons :



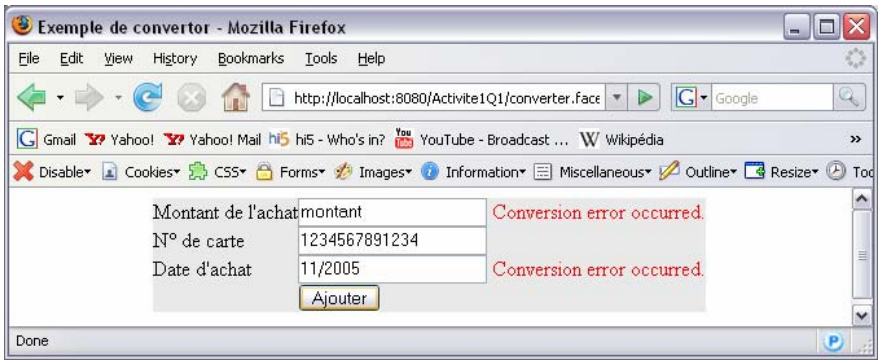
L'unité monétaire est ajoutée grâce à :

```
<h:outputText value="#{achatBean.montant}">
      <f:convertNumber type="currency"/>
</h:outputText>
```

Remarques:

- Pour que les convertisseurs fonctionnent, il faut que les attributs de achatBean aient les mêmes types respectifs que les champs auxquels ils sont attachés.
- Si les champs n'ont pas le même type que celui de la conversion ou si la valeur introduite ne correspond au schéma attendu l'action ne sera pas exécutée

Si jamais l'utilisateur introduit par mégarde de fausses valeurs qui ne correspondent pas aux schémas attendus, il doit être avertit de l'erreur et de sa localisation pour ça nous utilisons les messages :



La conversion perso :

Pour se faire, nous sommes appelés à développer notre propre classe CarteConversion qui implémenterait l'interface Converter. Cette implémentation nous amène à développer les deux méthodes :

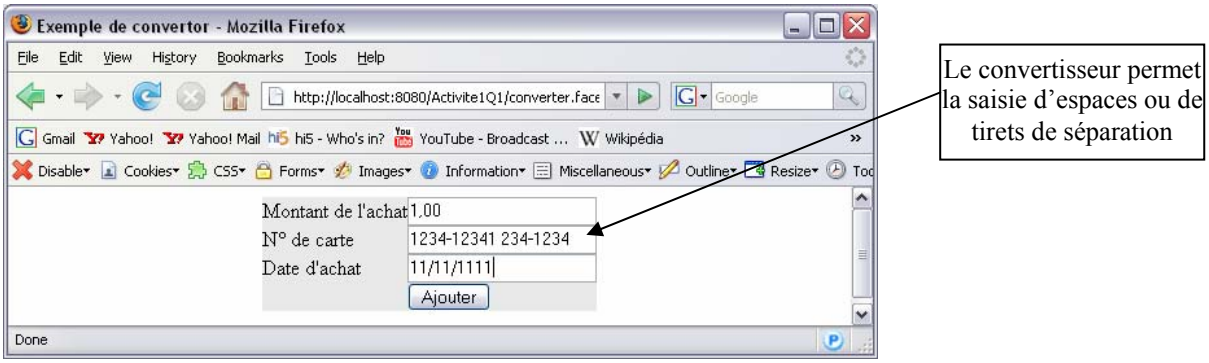
```
public Object getAsObject(FacesContext context, UIComponent
component,String newValue) throws ConverterException
```

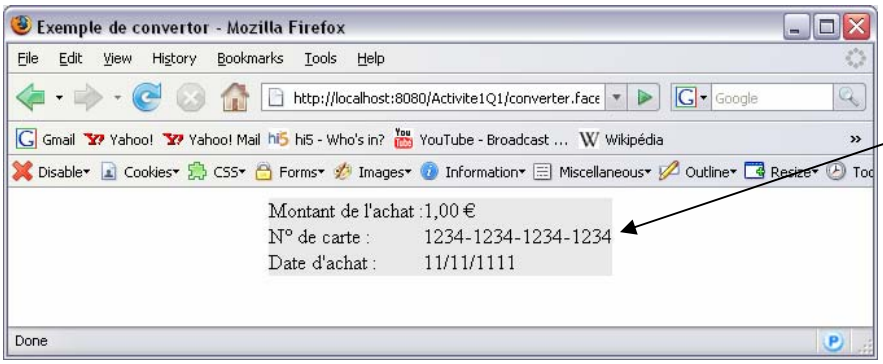
```
public String getAsString(FacesContext context, UIComponent
component,Object value) throws ConverterException
```

La première méthode, getAsObject permet de récupérer le contenu du champ de texte et de retourner un objet dont le type est celui de l'attribut en question.

La deuxième permet de formaliser l'affichage de l'objet dans une page résultat, ET ceci en renvoyant une chaîne de caractères.

Nous avons créée un nouveau type pour l'attribut carte de la classe AchatBean, il est désormais un objet de type Carte. Cet objet sera renvoyé par le convertisseur CarteConvertor lors de la requête d'envoi.





Le convertisseur ainsi créer doit être ajouté à faces-config.xml :

```
<converter>
      <converter-id>converter.Carte</converter-id>
      <converter-class>converter.CarteConverter</converter-class>
</converter>
```

Voici la classe CarteConverter :

```
package converter;
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.convert.Converter;
import javax.faces.convert.ConverterException;
import bean.Carte;
public class CarteConverter implements Converter
{
      public Object getAsObject(FacesContext context, UIComponent
component,String newValue) throws ConverterException
      {
            StringBuffer buffer = new StringBuffer(newValue);
            boolean foundInvalidCharacter = false;
            //char invalidCharacter = '\0';
            int i = 0;
            while (i < buffer.length() && !foundInvalidCharacter){
                  char ch = buffer.charAt(i);
                  if (Character.isDigit(ch)){i++;     }
                  else  {
                        if (Character.isWhitespace(ch) || ch =='-'){
                              buffer.deleteCharAt(i);
                        }
                        else  {
                              foundInvalidCharacter = true;
                              //invalidCharacter = ch;
                        }
                  }
            }
            if (foundInvalidCharacter)    {
                  throw new ConverterException("Exception : Caractère
invalide");
            }
            return new Carte(buffer.toString());
      }

      public String getAsString(FacesContext context, UIComponent
component,Object value) throws ConverterException {
            String v = value.toString();
```

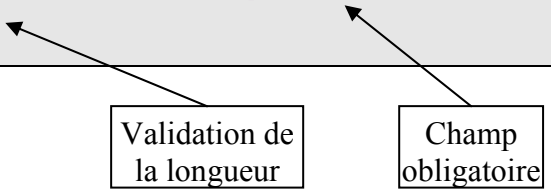
```
            int[] boundaries = null;
            boundaries = new int[]{ 4, 8, 12 };

            StringBuffer result = new StringBuffer();
            int start = 0;
            for (int i = 0; i < boundaries.length; i++){
                  int end = boundaries[i];
                  result.append(v.substring(start, end));
                  result.append("-");
                  start = end;
            }
            result.append(v.substring(start));
            return result.toString();
      }
}
```

Validation :

Utilisation des validateurs standard :

Les validateurs standard que nous avons utilisé sont : la définition des champs obligatoire et la définition de la longueur du numéro de la carte de crédit.

```
<h:inputText id="carte" value="#{achatBean.carte}" required="true">
      <f:validateLength minimum="16"/>
</h:inputText>
```



Nous avons aussi fixé une plage de prix pour lesquels nous pouvons valider un achat : Le montant minimum est 1 et le maximum est de mille.

```
<f:validateDoubleRange minimum="1" maximum="1000"/>
```

Utilisation de validateur perso :

Nous allons supposer que notre application d'achat concerne une zone bien déterminée, et que l'on a attribué à cette zone des numéros de cartes bancaires du type :

1111-xxxx-xxxx-xxxx

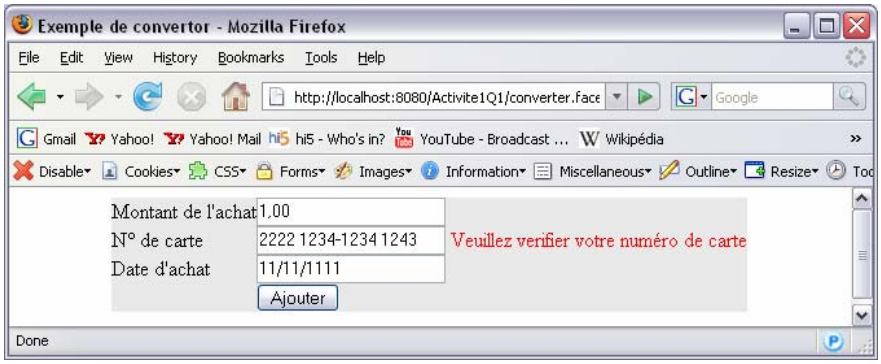
Donc, la validation portera sur ces quatre premiers chiffres. Pour ce faire, nous développons une nouvelle classe CarteValiator qui implémentera l'interface Validator. L'implémentation de cette classe demande la redéfinition de la méthode :

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

Si l'on introduit un mauvais numéro, l'envoi est bloqué, cependant aucune erreur n'est signalée : ce qui représente un problème de détection de l'erreur et de localisation de celle-ci.

Nous allons donc permettre à la classe CarteConverter de lever une exception de nature ValidatorException qui prend en argument un message d'erreur.

Voici un aperçu :



Le nouvel validateur doit être mentionné dans le fichier faces-config.xml comme suit :

```
<validator>
      <validator-id>validator.Carte</validator-id>
      <validator-class>validator.CarteValidator</validator-class>
</validator>
```

Voici un aperçu de la classe CarteValidator :

```
package validator;
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;
import bean.Carte;
public class CarteValidator implements Validator{

      public void validate(FacesContext context, UIComponent
component,Object value)
      {
            if(value == null)
                  return;

            String cardNumber;
            if (value instanceof Carte)
                  cardNumber = value.toString();
            else
                  cardNumber = getDigitsOnly(value.toString());

            if(!CartCheck(cardNumber))
            {
                  FacesMessage message =
util.Messages.getMessage("util.messages", "badLuhnCheck", null);
                  message.setSeverity(FacesMessage.SEVERITY_ERROR);
                  throw new ValidatorException(message);
            }
      }
      private boolean CartCheck (String x)
      {
            char[] verif=new char[] {'1','1','1','1'};
            boolean y=true;
            for (int i=0;i<4&&y;i++)
            {
                  char c=x.charAt(i);
                  if (c!=verif[i])
```

Une exception est levée en cas d'erreur

Méthode de validation de l'entête du numéro

```
                  {
                        y=false;
                  }
            }
            return y;
      }
      private static String getDigitsOnly(String s)
      {
            StringBuffer digitsOnly = new StringBuffer ();
            char c;
            for(int i = 0; i < s.length (); i++)
            {
                  c = s.charAt (i);
                  if (Character.isDigit(c))
                  {
                        digitsOnly.append(c);
                  }
            }
            return digitsOnly.toString ();
      }
}
```

Voici un aperçu de la classe messages :

```
package util;
import java.text.MessageFormat;
import java.util.Locale;
import java.util.MissingResourceException;
import java.util.ResourceBundle;
import javax.faces.application.Application;
import javax.faces.application.FacesMessage;
import javax.faces.component.UIViewRoot;
import javax.faces.context.FacesContext;

public class Messages {
      public static FacesMessage getMessage(String bundleName, String
resourceId,Object[] params)
      {
            FacesContext context = FacesContext.getCurrentInstance();
            Application app = context.getApplication();
            String appBundle = app.getMessageBundle();
            Locale locale = getLocale(context);
            ClassLoader loader = getClassLoader();
            String summary = getString(appBundle, bundleName,
resourceId,locale, loader, params);

            if (summary == null)
                  summary = "???" + resourceId + "???";

            String detail = getString(appBundle, bundleName, resourceId +
"_detail",locale, loader, params);
            return new FacesMessage(summary, detail);
      }
      public static String getString(String bundle, String
resourceId,Object[] params)
      {
            FacesContext context = FacesContext.getCurrentInstance();
            Application app = context.getApplication();
            String appBundle = app.getMessageBundle();
            Locale locale = getLocale(context);
```

```
            ClassLoader loader = getClassLoader();
            return getString(appBundle, bundle, resourceId, locale, loader,
params);
      }
      public static String getString(String bundle1, String bundle2,String
resourceId, Locale locale, ClassLoader loader,Object[] params)
      {
            String resource = null;
            ResourceBundle bundle;
            if (bundle1 != null)
            {
                  bundle = ResourceBundle.getBundle(bundle1, locale,
loader);
                  if (bundle != null)
                        try
                        {
                              resource = bundle.getString(resourceId);
                        }
                        catch (MissingResourceException ex)
                        {}
            }

            if (resource == null)
            {
                  bundle = ResourceBundle.getBundle(bundle2, locale,
loader);
                  if (bundle != null)
                        try
                        {
                              resource = bundle.getString(resourceId);
                        }
                        catch (MissingResourceException ex)
                        {
                        }
            }

            if (resource == null)
                  return null; // no match

            if (params == null)
                  return resource;

            MessageFormat formatter = new MessageFormat(resource, locale);
            return formatter.format(params);
      }

      public static Locale getLocale(FacesContext context)
      {
            Locale locale = null;
            UIViewRoot viewRoot = context.getViewRoot();
            if (viewRoot != null)
                  locale = viewRoot.getLocale();
            if (locale == null)
                  locale = Locale.getDefault();

            return locale;
      }
      public static ClassLoader getClassLoader()
      {
            ClassLoader loader =
Thread.currentThread().getContextClassLoader();
```

```
            if (loader == null)
                  loader = ClassLoader.getSystemClassLoader();
            return loader;
      }
}
```

Le fichier messages.propoerties, qui contient le message d'erreur spécifié pour un numéro de carte erroné doit être aussi indiqué dans le fichier faces-config.xml :

```
<application>
      <message-bundle>util.messages</message-bundle>
</application>
```

Validation avec dépendance de champs :

Le mécanisme de validation avec JSF est décrit pour valider un seul composant. Cependant, dans la pratique, vous avez souvent besoin d'assurer que différents composants ont des valeurs raisonnables.

Considérons l'exemple d'introduire deux valeurs, et n'envoyer ces valeurs que dans le cas où la seconde valeur soit supérieure à la première.

Dans le cas où le premier nombre serait supérieur au second, un message d'erreur s'afficherait.

Voici la classe de validation :

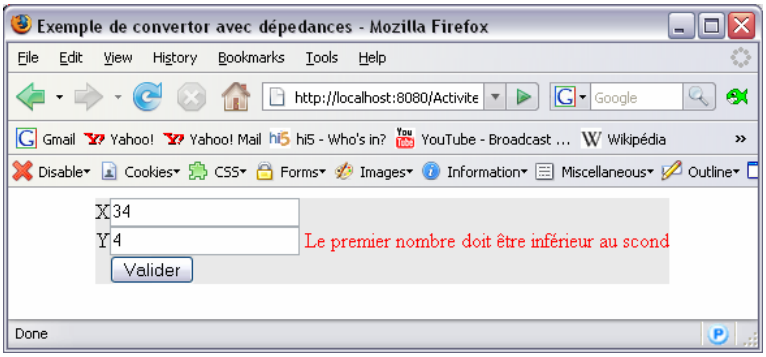
```
package bean;
import javax.faces.application.FacesMessage;
import javax.faces.component.UIComponent;
import javax.faces.component.UIInput;
import javax.faces.context.FacesContext;
import javax.faces.validator.ValidatorException;
public class ValeurBean {
        int a;
        int b;
        UIInput aInput;
        UIInput bInput;
        public ValeurBean() {
                super();
        }
        public ValeurBean(int a, int b, UIInput aInput, UIInput bInput) {
                super();
                this.a = a;
                this.b = b;
                this.aInput = aInput;
                this.bInput = bInput;
        }
        public int getA() {return a;}
        public void setA(int a) {this.a = a;}
        public UIInput getAInput() {return aInput;}
        public void setAInput(UIInput input) {aInput = input;}
        public int getB() {return b;}
        public void setB(int b) {this.b = b;}
```

```
        public UIInput getBInput() {return bInput;}
        public void setBInput(UIInput input) {bInput = input;}

        public void validate(FacesContext context, UIComponent component,Object value)
        {
                int x = a;//((Integer) aInput.getLocalValue()).intValue();
                int y = b;//((Integer) bInput.getLocalValue()).intValue();

                if (!isValidNumbers(x,y))
                {
                        FacesMessage message =
util.Messages.getMessage("util.messages","invalidNumbers", null);
                        message.setSeverity(FacesMessage.SEVERITY_ERROR);
                        throw new ValidatorException(message);
                }
        }
        private boolean isValidNumbers(int x, int y) {
                return (x<y);
        }
}
```

Voici la page JSP :

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<f:view>
<html>
   <head>
      <title>Exemple de convertor avec dépedances</title>
   </head>
   <body>
      <h:form>
         <table align="center" bgcolor="#e9e9e9" border="0" cellspacing="0"
         cellpadding="0">
            <tr>
               <td>
                  <h:outputText value="X"/>
               </td>
               <td>
                  <h:inputText id="x" value="#{valeurBean.a}"
                  required="true" binding="#{valeurBean.aInput}"/>
               </td>
            </tr>
            <tr>
               <td>
                  <h:outputText value="Y"/>
               </td>
               <td>
                  <h:inputText id="y" value="#{valeurBean.b}"
                   required="true" binding="#{valeurBean.bInput}"
                   validator="#{valeurBean.validate}" />
                  <h:message for="y" style="color:red" />
               </td>
            </tr>
```

```
            <tr>
               <td>
               </td>
               <td>
                  <h:commandButton value="Valider" action="verifier"/>
               </td>
            </tr>
         </table>
      </h:form>
   </body>
</html>
</f:view>
```



Internationalisation :

JSF propose des fonctionnalités qui facilitent l'internationalisation d'une application. Il faut définir un fichier au format properties (que nous avons déjà utilisé pour les messages d'erreurs des validations personnalisées) qui va contenir la définition des chaînes de caractères. Un tel fichier possède les caractéristiques suivantes :

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

Voici un exemple de contenu d'un fichier properties :

```
login_titre=Application JSF
login_identification=Identification
login_nom=Nom
login_mdp=Mot de passe
login_Login=Valider
```

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

Exemple :

```
msg.properties
msg_en.properties
msg_de.properties
```

Dans chacun de ces fichiers, nous devons garder les mêmes clefs et les différentes valeurs seront écrites dans le langage en question.
Les langues qui seront disponibles pour l'application, devront être précisées dans le fichier de configuration faces-config.xml.

```
<faces-config>
...
  <application>
    <locale-config>
      <default-locale>fr</default-locale>
      <supported-locale>en</supported-locale>
      <supported-locale>fr</supported-locale>
      <supported-locale>ar</supported-locale>
    </locale-config>
  </application>
...
</faces-config>
```

La langue chargée par défaut

L'ensemble des langues supportées

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

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

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

Le chargement de la page, utilisera automatiquement la langue spécifiée comme étant langue par défaut sera chargée. Cependant nous pouvons forcer la page à utiliser une autre langue supportée, bien entendu, par l'application grâce à la méthode suivante :

```
<f:view locale="ar">
```

Sinon, nous pouvons régler le choix de la langue par la navigation :
Le bloc de code suivant spécifie les actions de changement de langue, et l'appel de méthode correspondant :

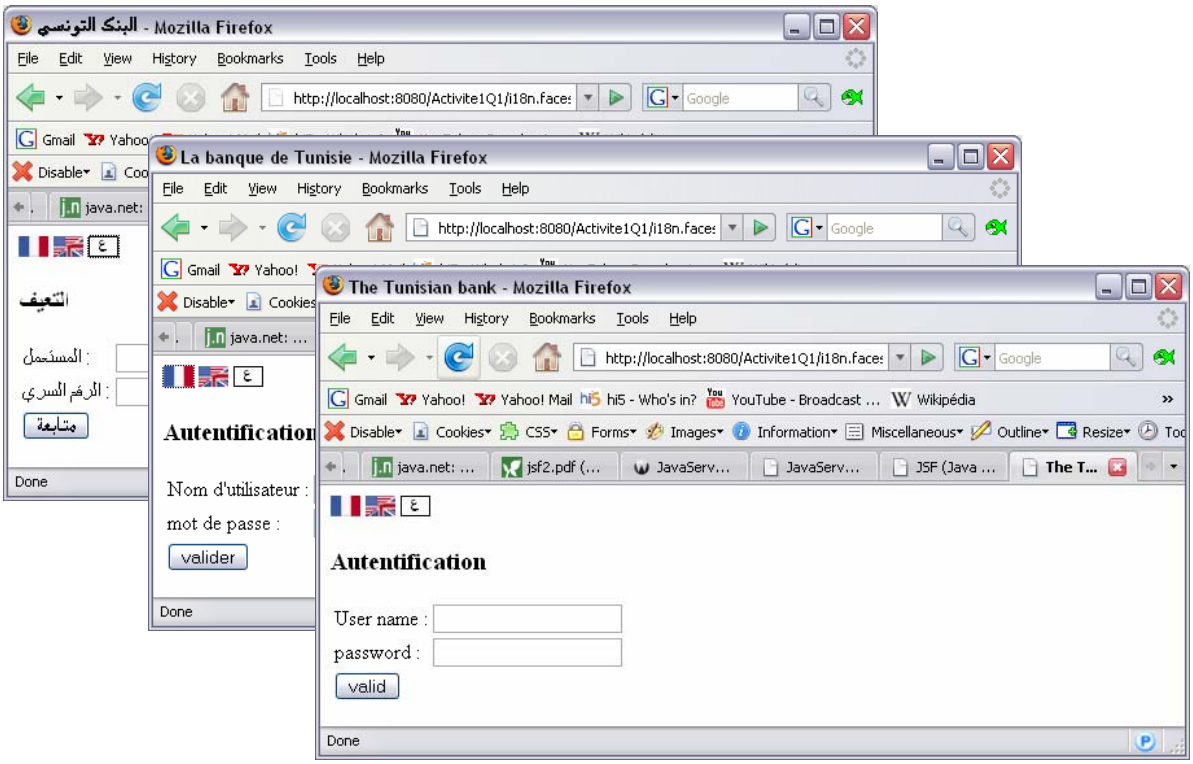
```
<h:commandLink id="fr" immediate="true" action="#{langueApp.activerFR}">
      <h:graphicImage value="fr.jpg" style="border: 0px"/>
    </h:commandLink>
    <h:commandLink id="en" immediate="true"
action="#{langueApp.activerEN}">
        <h:graphicImage value="en.jpg" style="border: 0px"/>
    </h:commandLink>
    <h:commandLink id="ar" immediate="true"
action="#{langueApp.activerAR}">
      <h:graphicImage value="aar.jpg" style="border: 0px"/>
    </h:commandLink>
```

Ces appels utilisent le bean suivant :

```
package bean;
import java.util.Locale;
import javax.faces.context.FacesContext;
public class LangueApp {
  public String activerFR() {
    FacesContext context = FacesContext.getCurrentInstance();
```

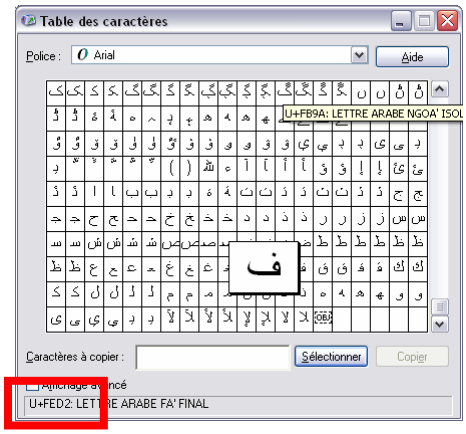
```
    context.getViewRoot().setLocale(Locale.FRANCE);
    System.out.println("okFR");
    return null;
  }
  public String activerEN() {
    FacesContext context = FacesContext.getCurrentInstance();
    context.getViewRoot().setLocale(Locale.CHINA);
    System.out.println("okEN");
    return null;
  }
  ...
}
```

Et voici un aperçu de la navigation :



Remarque :

Pour le fichier properties de la langue arabe, les caractères doivent être écrits en unicode. Nous avons extrait ces informations à partir de la table de caractères de Windows, mais la liste est entièrement disponible à www.unicode.org



Voici un aperçu du fichier :

```
titre=\u0627\ufedf\ufe92\ufee7\ufb8e
\u0627\ufedf\ufe97\ufeee\ufee7\ufeb3\ufbe5
msg_bienvenue=\ufee3\ufead\ufea3\ufe91\ufe8e
aut=\u0627\ufedf\ufe98\ufecc\ufbff\ufed2
utilisateur=\u0627\ufedf\ufee3\ufeb3\ufe97\ufecc\ufee3\ufede
pass=\u0627\ufedf\ufeae\ufed7\ufee2 \u0627\ufedf\ufeb3\ufeae\ufbe4
valid=\ufee3\ufe98\ufe8e\ufe91\ufecc\ufe94
```

La création de composants personnalisés :

Pour la création d'un composant personnalisé (custom component), nous allons développer un simple tag qui affiche un message muni de la date et l'heure courante.

Avant de montrer comment établir un composant personnalisé HelloWord, passons rapidement en revue comment l'employer. Pour appeler le composant nous avons une page de JSP permise par JSF avec une directive supplémentaire de taglib indiquant une bibliothèque faite sur commande d'étiquette (tld) qui contient l'étiquette faite sur commande, le « jsfhello », qui est associé à notre composant personnalisé UI. Pour employer l'étiquette de jsfhello, nous la plaçons simplement dans le corps du JSP et plaçons un attribut facultatif « hellomsg ».

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<%@ page contentType="text/html;charset=windows-1252"%>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<%@ taglib uri="monURI" prefix="cf"%>
<f:view>
  <html>
    <head>
      <meta http-equiv="Content-Type" content="text/html; charset=windows-
1252"></meta>
    </head>
    <body>
     <h:form>
     <p>A simple HelloWorld Custom Component:</p>
     <p>The HelloWorld UI Component:</p>
     <cf:jsfhello hellomsg="Hello world! This is output from a custom JSF
Component!"   />
     </h:form>
    </body>
  </html>
</f:view>
```

1ère étape : Construction du composant personnalisé UIComponent:

Pour notre exemple, le UIComponent va hériter de la classe abstraite UIComponentBase qui fait partie de la spécification JSF.

Voici notre classe HelloUIComponent

```
package customComponent;
import java.util.Date;
import javax.faces.component.UIComponentBase;
import javax.faces.context.FacesContext;
import java.io.IOException;
import javax.faces.context.ResponseWriter;
public class HelloUIComp extends UIComponentBase
{

  public void encodeBegin(FacesContext context) throws IOException
  {
   ResponseWriter writer = context.getResponseWriter();
                                                          Récupération du contexte
                                                          pour préparer le retour

String hellomsg = (String)getAttributes().get("hellomsg");

                                                          Récupération de l'attribut
   writer.startElement("h3", this);
   if(hellomsg != null)
     writer.writeText(hellomsg, "hellomsg");
   else
     writer.writeText("Hello from a custom JSF UI Component!", null);
   writer.endElement("h3");
   writer.startElement("p", this);
   writer.writeText(" Today is: " + new Date(), null);
   writer.endElement("p");
  }

 public String getFamily()                                Le rendu de la méthode
 {
  return "HelloFamily";
 }
}
```

Comme vous pouvez le remarquer, ce composant UI retourne un message formaté en utilisant la méthode **encodeBegin()**. Notez aussi que nous avons définit la méthode **getFamily()**. Ceci est nécessaire pour toutes classe héritant de UIComponentBase. Pour cet exemple nous n'allons par créer une nouvelle famille de composant, donc la méthode getFamily() peut simplement retourner une chaîne de caractères.

2ème étape : Signaler le composant dans le fichier de configuration faces-config.xml:

```
<component>
      <component-type>customComponent.JSFHello</component-type>
      <component-class>customComponent.HelloUIComp</component-class>
</component>
```

3ème étape : construction d'une librairie personnalisée du tag:

Afin de pouvoir utiliser notre tag personnaliser, nous devons développer un fichier de configuration de tag personnalisé TLD et une classe de traitement (Handler).

Construction du Handler:

Pour le développement de composants JSF, la classe de traitement du tag est dérivée de la classe javax.faces.webapp.UIComponentTag. Le but est de :

1. Associer un tag et un composant UI.
2. Associer une classe de rendu séparée et un composant UI.
3. Fixer les propriétés à partir de l'attribut du tag au composant UI.

Voici la source de notre handler :

```
package customComponent;
import javax.faces.application.Application;
import javax.faces.webapp.UIComponentTag;
import javax.faces.component.UIComponent;
import javax.faces.el.ValueBinding;
import javax.faces.context.FacesContext;
public class FacesHelloTag extends UIComponentTag
{
  // Declare a bean property for the hellomsg attribute.
  public String hellomsg = null;

  // Associate the renderer and component type.
  public String getComponentType() { return "customComponent.JSFHello"; }
  public String getRendererType() { return null; }

  protected void setProperties(UIComponent component)
  {
    super.setProperties(component);

    // set hellomsg
    if (hellomsg != null)
    {
      if (isValueReference(hellomsg))
      {
        FacesContext context = FacesContext.getCurrentInstance();
        Application app = context.getApplication();
        ValueBinding vb = app.createValueBinding(hellomsg);
        component.setValueBinding("hellomsg", vb);
      }
      else
        component.getAttributes().put("hellomsg", hellomsg);
    }
  }

  public void release()
  {
    super.release();
    hellomsg = null;
  }

  public void setHellomsg(String hellomsg)
  {
    this.hellomsg = hellomsg;
  }
}
```

La première chose que vous allez remarquer, c'est l'attribut hellomsg muni de ses accesseurs. Il s'agit en faite d'un attribut du tag JSP :

```
<sf:jsfhello hellomsg="Hello world!"../>
```

Les deux premières methods **getComponentType()** et **getRendererType()** permettent d'associer le handler avec notre composant UI enregistré sous

« customComponent.JSFHello ». Nous n'avons pas de classe de rendu séparée, c'est pour cette raison que la deuxième méthode renvoi la valeur nulle.

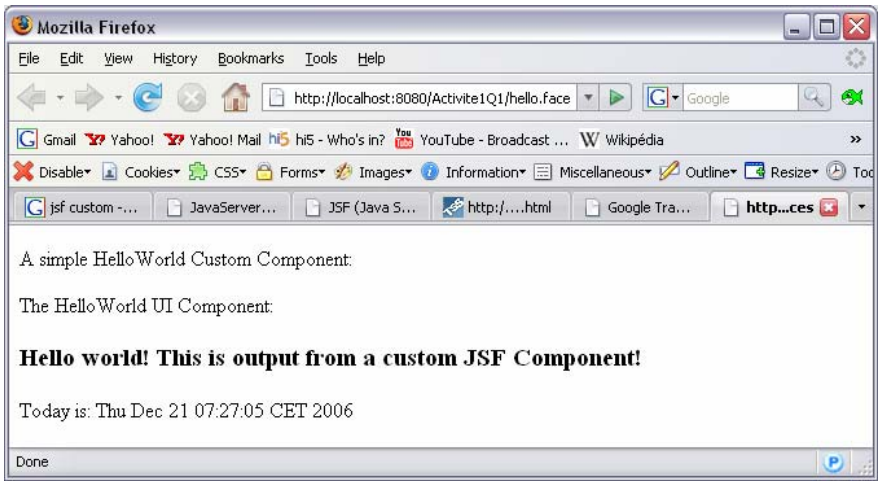
La méthode **setProperties**, comme son nom l'indique, permet de faire un bind entre la valeur passée par le tag JSP et l'attribut. La méthode release() a pour rôle de reinitialiser le bean.

4ème étape : la TLD

Le fichier TLD doit être situé sour le répertoir Web-inf.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
  PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
  "http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">
<taglib>
   <tlib-version>0.01</tlib-version>
   <jsp-version>1.2</jsp-version>
   <short-name>simple</short-name>
   <uri>monURI</uri>
   <description>Un simple exemple.</description>
   <tag>
      <name>jsfhello</name>
      <tag-class>customComponent.FacesHelloTag</tag-class>
      <attribute>
         <name>binding</name>
         <description>
            A value binding that points to a bean property
         </description>
      </attribute>
      <attribute>
         <name>id</name>
         <description>The client id of this component</description>
      </attribute>
      <attribute>
         <name>rendered</name>
         <description>Is this component rendered?</description>
      </attribute>
      <attribute>
         <name>hellomsg</name>
         <description>a custom message for the Component</description>
      </attribute>
   </tag>
</taglib>
```

Et voilà le résultat:



Nous allons maintenant étendre notre application, afin de réaliser un composant personnalisé. Nous allons passer pas les étapes précédentes pour produire le tag du composant, cependant nous allons apporter des changements majeurs à la classe de l'UI.

Nous allons présenter la nouvelle classe HtmlInputSolde qui va hériter de UIInput car cette classe va accepter des valeurs en entrée.

Ce composant va avoir une méthode d'encodage plus détaillée que le helloWord présenté. Cette fois ci, la méthode encodeBegin() va déléguer ses traitements à trois nouvelles méthodes. Ces méthodes s'occuperont du champ d'entrée, du bouton d'envoi et du message à afficher. Vous allez noter que nous avons joint l'identifiant ".inputbutton" et ".submitbutton" à l'identifient du client. Ceci est effectué pour garantir l'unicité des noms des champs utilisés pour les cas où le composant à créer va être utilisé plus d'une fois dans une même page JSP.

```
package customComponent;
import java.io.IOException;
import java.util.Map;
import javax.faces.component.UIInput;
import javax.faces.context.FacesContext;
import javax.faces.context.ResponseWriter;
public class HtmlInputSolde extends UIInput{

      public void encodeBegin(FacesContext context) throws IOException {
          ResponseWriter writer = context.getResponseWriter();
          String clientId = getClientId(context);
          encodeInputField(writer, clientId+".inputbutton");
          encodeSubmit(writer, clientId+".submitbutton");
          encodeOutputField(context);
         }

      private void encodeInputField(ResponseWriter writer, String clientId)
throws IOException {
            // render a standard HTML input field
            writer.startElement("input", this);
            writer.writeAttribute("type", "text", null);
            writer.writeAttribute("name", clientId, "clientId");

            Object v = getValue();
            if (v != null)
               writer.writeAttribute("value", v.toString(), "value");

            writer.writeAttribute("size", "6", null);
            writer.endElement("input");
         }
```

Ajout du champ d'entrée

```
      private void encodeSubmit(ResponseWriter writer, String clientId)
throws IOException {
            // render a submit button
            writer.startElement("input", this);
            writer.writeAttribute("type", "Submit", null);
            writer.writeAttribute("name", clientId, "clientId");
            writer.writeAttribute("value", "Enter Stock Symbol", null);
            writer.endElement("input");
         }
      public void encodeOutputField(FacesContext context) throws
IOException
        {

         ResponseWriter writer = context.getResponseWriter();
         String symbol = (String)getAttributes().get("value"); // The
"value" attribute is used to pass the stock symbol.

         //get stock price
         String stockprice = Compte.getSolde(symbol);

         writer.startElement("p", this);
         if(symbol!=null)
           writer.writeText("The current price for " + symbol + " is: " +
stockprice + ".", null);
         else
           writer.writeText("", null);
         writer.endElement("p");
        }

      public void decode(FacesContext context) {
            Map requestMap =
context.getExternalContext().getRequestParameterMap();
            String clientId = getClientId(context);
            try {
            String string_submit_val = ((String)
requestMap.get(clientId+".inputfield"));
            setSubmittedValue(string_submit_val);
            setValid(true);
            }
            catch(NumberFormatException ex) {
               setSubmittedValue((String) requestMap.get(clientId));
            }
       }
}
```

Ajout du bouton d'envoi

Nous travaillons avec une classe Compte

Gestion du message de sortie

La méthode decode() est responsable du parsing des valeurs récupérées et des actions à appliquer sur ces derniers.

Voici un aperçu de la méthode setProperties de la nouvelle classe du tag, FacesHtmlTag :

```
protected void setProperties(UIComponent component)
  {
  super.setProperties(component);
  if (value != null)
    {
     if (isValueReference(value))
      {
        FacesContext context = FacesContext.getCurrentInstance();
        Application app = context.getApplication();
        ValueBinding vb = app.createValueBinding(value);
```

```
            component.setValueBinding("value", vb);
        }
        else
            component.getAttributes().put("value", value);
    }
}
```

Nous mettons à jour notre fichier TLD.

La classe Compte utilisé dans la classe de l'UI, est en fait une simple classe qui nous permet de récupérer un solde en entrant un numéro de compte, notre exemple est simple, mais cette classe peut implémenter toute une logique d'accès à une base donnée, ce qui va donnée en gros : un composant qui récupère des résultats à partir d'une base de comptes ou bien exploiter un service web.

Voici un aperçu :