Introduction à java server page (jsf) avec éclipse et couplage avec Hibernate via les annotations JAVA

Kissangou_julis@jkissangou.fr

Février 2011

Version 1.0

Table des matières

Préambule

Le présent tutoriel est une introduction de JSF avec couplage Hibernate, cette introduction se fera via un exercice, le but de celui-ci est de permettre de se familiariser à l'utilisation des dits Framework en ayant une bonne base pratique de démarrage. Il n'a nullement pour objectif de faire de quiconque un expert, une connaissance en JAVA et des fondamentaux des technos Web et JAVAEE est en revanche indispensable.

II-JSF

II-1.0 JSF qu'est-ce à dire ?

Java Server Page est un Framework ayant pour objectif la création des interfaces Web, il se repose sur l'architecture MVC (Model Vue Contrôleur) et les technologies JAVA d'applications web (JSP et Servlets) et bénéficie par ailleurs des apports du Framework STRUTS dont le concepteur a coécrit les spécifications.

II-1.1 Pourquoi utiliser JSF ?

Par le concept de composants JSF permet aux développeurs d'applications web de créer rapidement des interfaces utilisateurs et être ainsi très réactifs. Nous pouvons par ailleurs mettre de plus à son actif la facilité d'accès aux Beans ainsi que les propriétés de ces derniers, exit les classes action, les mapping et les transferts manuels entre les propriétés de Beans actionForm et ceux du Bean modèles comme c'est le cas avec STRUTS.

JSF permet :

– une séparation nette entre la couche de présentation et les autres couches (les couches métier, services et persistance des données)

– le mapping HTML/Objet (un tag jsf est aussi bien utilisé comme Objet ou composant java et simple balise pour le rendu)

Ex : un tag < h :dataTable > représente un objet **UIData** de la classe **javax.faces.component.UIData** mais contribue aussi au rendu de cette table de données sur une page web

– un modèle riche de composants graphiques réutilisables

– une gestion de l'état de l'interface entre les différentes requêtes par le paramètre

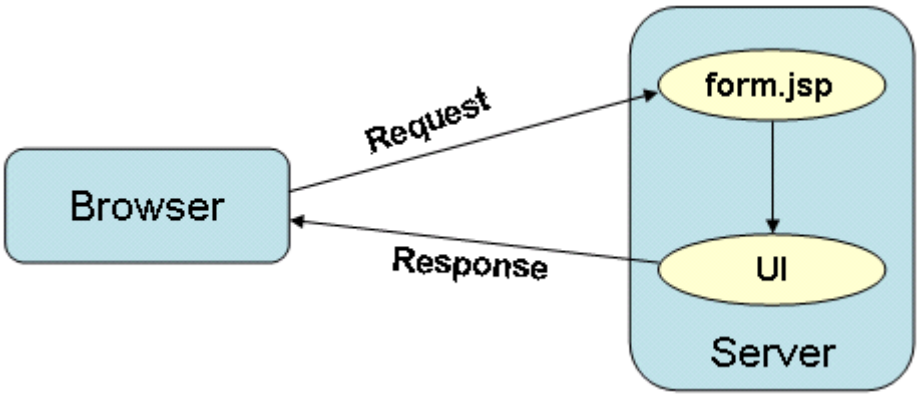
javax.faces.STATE_SAVING_METHOD (la valeur par défaut est « client»)

– une liaison simple entre les actions côté client de l'utilisateur et le code Java correspondant côté serveur

– la création de composants graphiques personnalisés grâce à une API

– le support de différents clients (HTML, WML, XML, ...)

Relation entre un client, un serveur et JSF(les composant **UI** s'exécutent sur le serveur)



La page JSP fait office d'interface utilisateur en utilisant la bibliothèque de balises personnalisées de JSF. Quand on accède à la page JSP ces balises sont interprétées et un arbre à composants dont la racine de type **javax.faces.component.UIViewRoot** est créé cotés serveur cet arbre comporte des composants de type **javax.faces.component.UIComponent**. Les composants UI de l'application gèrent les objets rendus par la page JSP.

II-1.2 Entrée en matière

Ce tutoriel fait usage d'un exercice permettant d'entrée dans le vif du sujet, cet exercice est la création d'un formulaire d'entrée de données liées à l'état civil d'un individu. Au fur et à mesure de l'évolution de l'exercice nous exposerons les éléments évoqués dans table des matières.

Les points qui seront traités dans l'exercice seront :

-l'affichage des données, la validation, la conversion de ces derniers

-l'enregistrement des données

-la lecture des données enregistrées

-la suppression des données

Nous nous concentrerons premièrement sur l'affichage, la validation, la conversion des données. Le reste des points seront évoqués dans la partie Hibernate

II-1.2.0 Installation de l'environnement de travail

Notre choix sur l'implémentation JSF se portera sur celui de Sun, nous utiliserons par ailleurs l'IDE éclipse et le moteur de servlets TOMCAT, l'installation est effectuée sur Windows.

Ci-dessous les liens de nos outils :

1- Les « .jar » du **Framework JSF** (**jsf-api.jar** et **jsf-impl.jar**) sont téléchargeable sur : http://javaserverfaces.java.net/download.html ils sont dans un dossier compressé du style mojarra-xxxx-FCS-binary.

En plus de l'usage de sa bibliothèque de balises personnalisées JSF fait usage de **la bibliothèque des balises standards de jsp** nous compléterons alors les « .jar » suivants **jstl-api.jar** et **jstl-impl.jar** téléchargeables sur http://jstl.java.net/download.html

2- **Tomcat** est téléchargeable à cette adresse : http://tomcat.apache.org/

Nous avons pour notre tutoriel utilisé la version 6 de Tomcat.

3- **Eclipse** est téléchargeable à l'adresse suivante : http://www.eclipse.org/downloads/

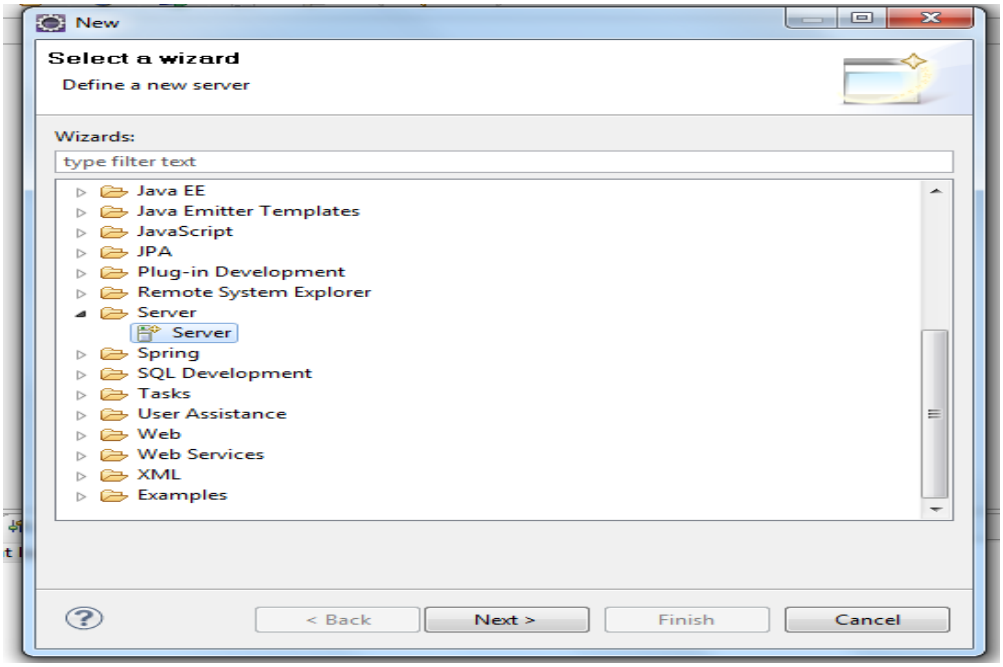
Nous avons pour notre tutoriel utilisé le la version d'éclipse suivante **eclipse-jee-galileo-SR2-win32**.

REMARQUE : Nous vous conseillons de déposer tous les « .jar » dans le dossier **/WEB-INF/lib** ou directement dans le répertoire **lib** de Tomcat.

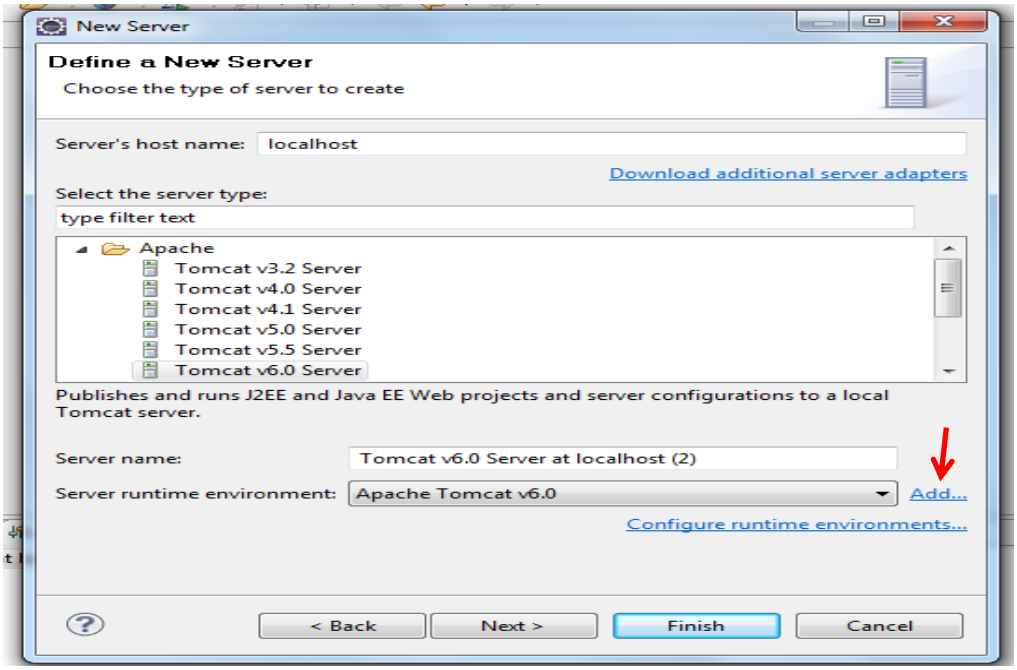
II-1.2.0.1 Configuration de l'environnement

Après avoir effectué le téléchargement et extrait les dossiers compressés suivez les instructions suivantes :

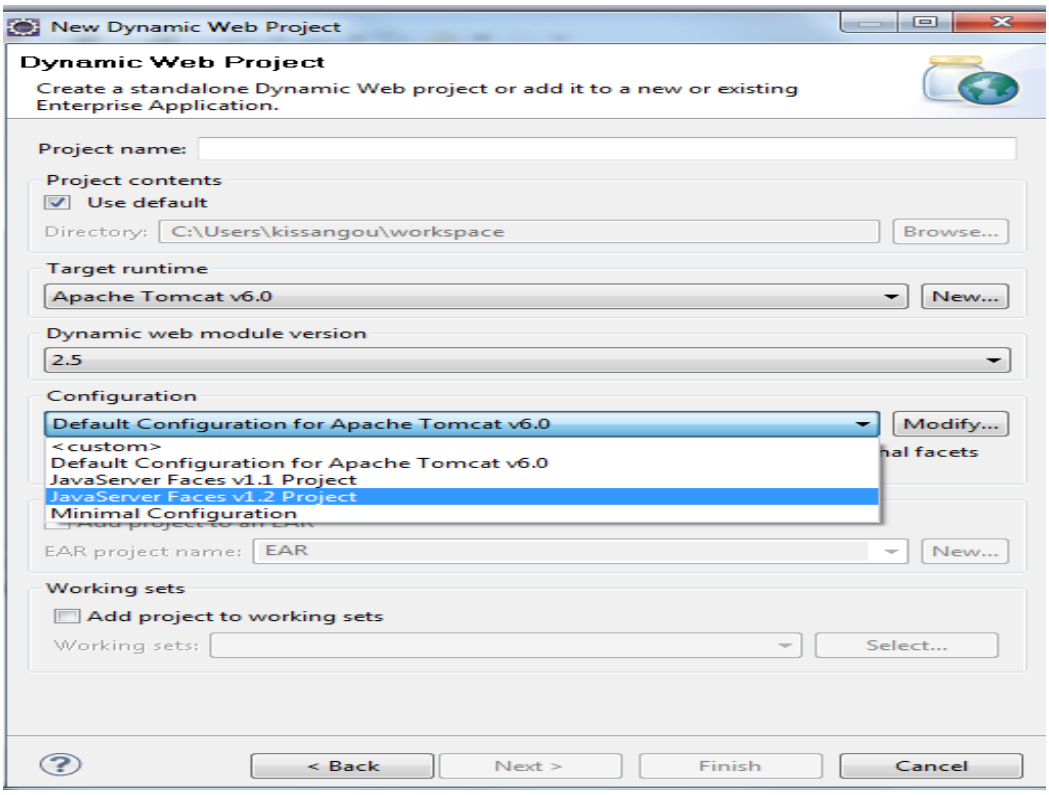
1- lancez l'exécutable **eclipse.exe** se trouvant dans le dossier éclipse puis choisissez un espace de travail (workbench) que vous désirez
2- créez un projet serveur en y associant Tomcat comme serveur d'application, pour le faire allez dans **File→New→Other→Server→Server→**cliquez sur **Next**



Dans la fenêtre qui suit choisissez la version du serveur Tomcat, puis cliquez sur **add** afin de spécifier L'emplacement du dit serveur et de lui donner un nom évocateur puis terminer votre la procédure

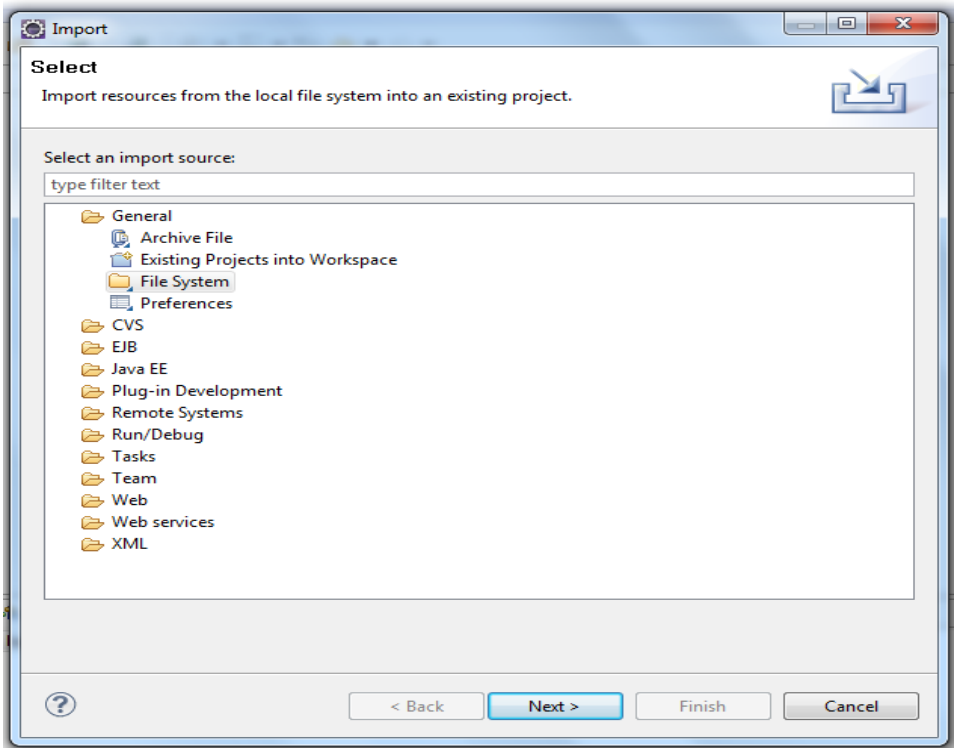


3- créez un projet web dynamique en choisissant javaServer Face v1.2 pour la partie **configuration**, pour le faire allez dans **File→New→Other→Web→Dynamic web project**



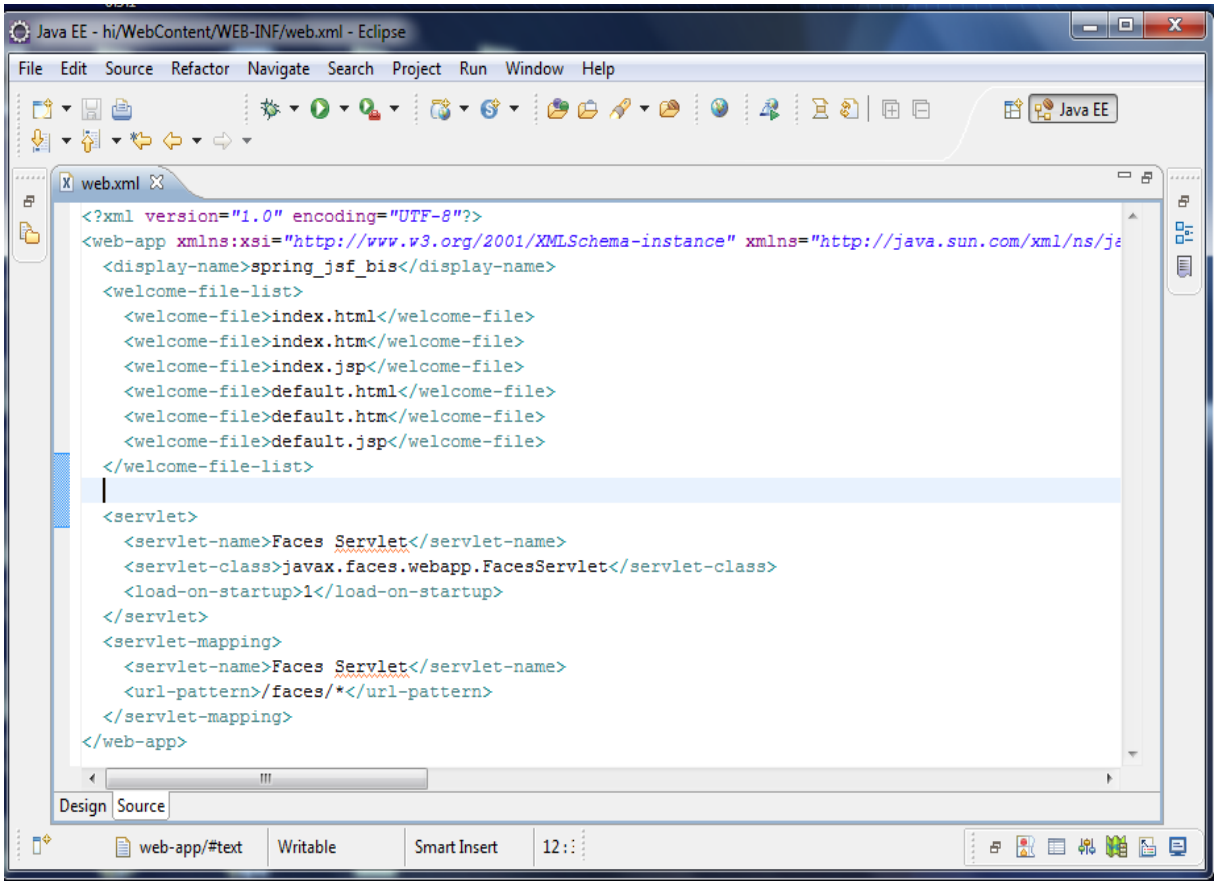
donnez un nom à votre projet puis cliquez sur finish, nous l'appellerons **jsfHibernate**

4- importer les .jar dans le dossier **WebContent/WEB-INF/lib** de votre projet, pour le faire allez dans **File→Import→**choisissez **File System→**puis cliquez sur **Next**



choisissez l'emplacement de vos librairies et importez les

5- configurer le fichier Web.xml, par défaut il est déjà configuré pour utiliser JSF.



Vous pourrez ajouter les lignes suivantes

```
<context-param>
<param-name>javax.faces.STATE_SAVING_METHOD</param-name><param-value>server</param-value>
</context-param
```

si vous voulez que l'état de la vue soit conservé sur le serveur entre les différentes requêtes, par contre cela risque de vous couter en espace mémoire sur le serveur, la valeur par défaut **client**.

Nous remarquons dans le fichier web.xml la présence de la servlet principale **javax.faces.webapp.FacesServelet** responsable de l'administration de tous les composants dans l'architecture MVC2.
Nous constatons de même que cette servlet est identifiée par un mapping qui a pour préfixe **/faces/*,** ainsi tous les appels à celle-ci seront du genre **http://localhost:8080/nom du contexte/faces/page. jsp**. Toutefois ce préfixe peut être changé par de une simple extension ***.faces ou *.jsp** on aura donc comme appel **http://localhost:8080/nom du contexte/page. Jsp** ou **http://localhost:8080/nom du contexte/page. faces**

II-1.2.1 Les composants graphiques

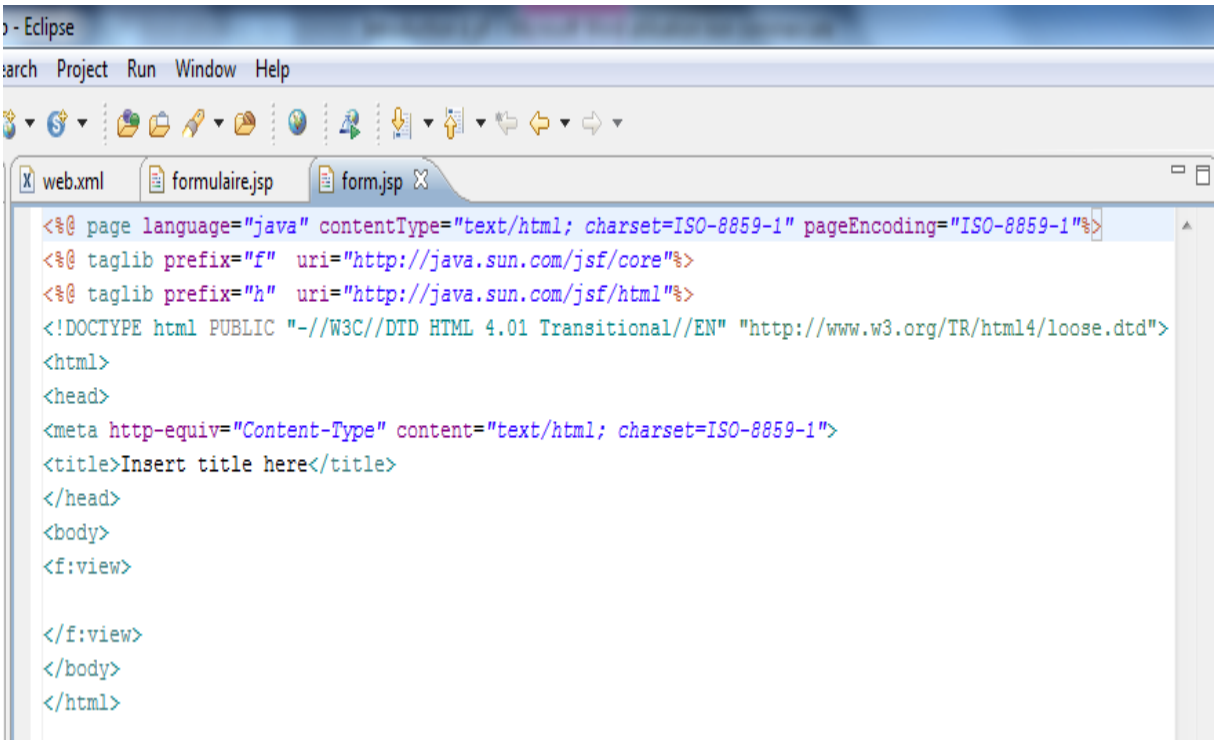
Les composants graphiques JSF sont des tags(ou balises) de la librairie personnalisée de JSF qui permettent de créer une interface web, à la différence des tags de langage de balise tel que html ils font aussi office d'objets JAVA étendant la classe **javax.faces.component.UIComponent**.

Après avoir créé notre projet (**jsfHibernate**) et effectué sa configuration, nous créerons ici notre première page JSF, cette page sera le formulaire permettant la saisie de données d'état civil. Nos pages JSF seront insérées dans le dossier **WebContent** de notre projet.

Pour le faire allez dans votre projet (**jsfHibernate**) et suivez les instructions suivantes :

Faites un clic droit sur dossier **WebContent**→pointez le curseur sur **New**→Cliquez sur **JSP**→Entrez le nom de votre page→Cliquez sur **Next**→puis choisissez **New JavaServer Face(JSF) Page (html)**→puis cliquez sur **Finished.**

Vous aurez le résultat suivant qui est le squelette d'un fichier utilisant JSF:



Nous remarquons la présence des instructions suivantes tout en haut de la page:

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

Ces instructions permettent l'insertion des librairies de tags(ou balises) JSF via la directive **<%@ taglib%>**. Les préfix peuvent être changés, le « h » est pour les adaptations de balises HTML et le « f » pour ceux directement liés à JSF.

Nous remarquons aussi la présence dans le corps du fichier du tag **<f:view> </f:view>,** ceci pour la simple raison que toutes les autres tags JSF doivent être compris dans ledit tag.

Pour créer le formulaire nous utiliserons le tag **<h :form></h :form>**) à l'intérieur duquel en insérera les tags suivants :
<h:panelGrid ></h:panelGrid > (pour la création d'un tableau)
<h:outputText></h:outputText> (pour l'affichage de textes)
<h:inputText ></h:inputText > (pour la saisie de texte dans la zone de saisie)
<h:message ></h:message> (pour l'affichage des erreurs ou de messages d'information)
<h:selectOneMenu></h:selectOneMenu> (pour un choix dans un menu)
<h:selectOneRadio></h:selectOneRadio> (pour les boutons radio)
<h:commandButton> </h:commandButton> (pour la validation du formulaire)
<**f:selectItem />** ce tag sera utilisé pour les choix d'item dans les tags **<h :selectOneXXXXXXX>**

Voici un extrait de code du fichier " **formulaire.jsp"** et ses explications plus bas, le reste pourra être téléchargé sur mon site

```
<f:view>
<h:form>
<h:panelGrid columns="3">

                    <!- - ENTRER VOTRE NOM -->
            <h:outputText value="Nom : "></h:outputText>
             <h:inputText id="idname"
                         value=""
                         required="true"
                         requiredMessage="ce champ de doit pas être vide" >
              </h:inputText>
               <h:message for="idname"></h:message>

              <!- - CHOIX DE SEXE -->

<h:outputText value="Sexe : "></h:outputText>
<h:selectOneRadio id="idChoixSexe" value=""
                  required="true"
                 requiredMessage="faites un choix de genre">
<f:selectItem itemLabel="Féminin" itemValue="Féminin" id="itemsexe1"/>
<f:selectItem itemLabel="Masculin" itemValue="Masculin" id="itemsexe2"/>
</h:selectOneRadio>
<h:message for="idChoixSexe"></h:message>
          <!- - CHOIX D'ACTION A EXECUTER (ENREGISTRER OU EFFACER) -->

<h:outputText value="Que faire : "></h:outputText>
<h:selectOneMenu id="idChoixMenu" value="" reqired="true"
requiredMessage="faite un choix d'action">
<f:selectItem itemLabel="" itemValue="" id="itemaction1"/>
<f:selectItem itemLabel="Enregistrer" itemValue="1" id="itemaction2"/>
<f:selectItem itemLabel="Effacer" itemValue="2" id="itemaction3"/>
</h:selectOneMenu>
<h:message for="idChoixMenu"></h:message>
</h:panelGrid > <br>
<h:commandButton value="valider" action="confirmerDonnee">
</h:commandButton>
</h:form>
</f:view>
```

Nous avons bien évidement le tag <f :view> tout au début pour permettre l'insertion des autres tags JSF
Le tag `<h:panelGrid columns="3">` permet d'afficher un tableau de trois colonnes, cela est spécifié par l'attribut **colums**. Nous aurons donc 3 cellules par ligne, ceci est un détail important car chaque cellule ne peut contenir qu'un seul composant graphique. Les cellules de chaque ligne contiendront les composants `<h:outputText>, <h:inputText> et <h:message>.`

Le tag `<h:outputText value="Nom : ">` permet d'afficher via l'attribut value le texte « Nom : » sur la page web

Le tag `<h:inputText id="idname" value="" required="true" requiredMessage="ce champ de doit pas être vide" >` permet d'afficher une zone de saisie.
- l'attribut **id** représente l'identifiant unique du tag.
- l'attribut **value** permet de spécifier une propriété d'un **bean (bean managé)**qui contiendra la valeur du nom entré, il n'est pas utilisé pour l'instant , nous y reviendrons dans la section des beans managés.
- l'attribut **required** avec la valeur **true** oblige à saisir un texte
-l'attribut **requiredMessage** permet de spécifier un message d'alerte pour obliger la saisie d'un texte il est conjointement utilisé avec l'attribut **required** qui doit avoir la valeur **true.**

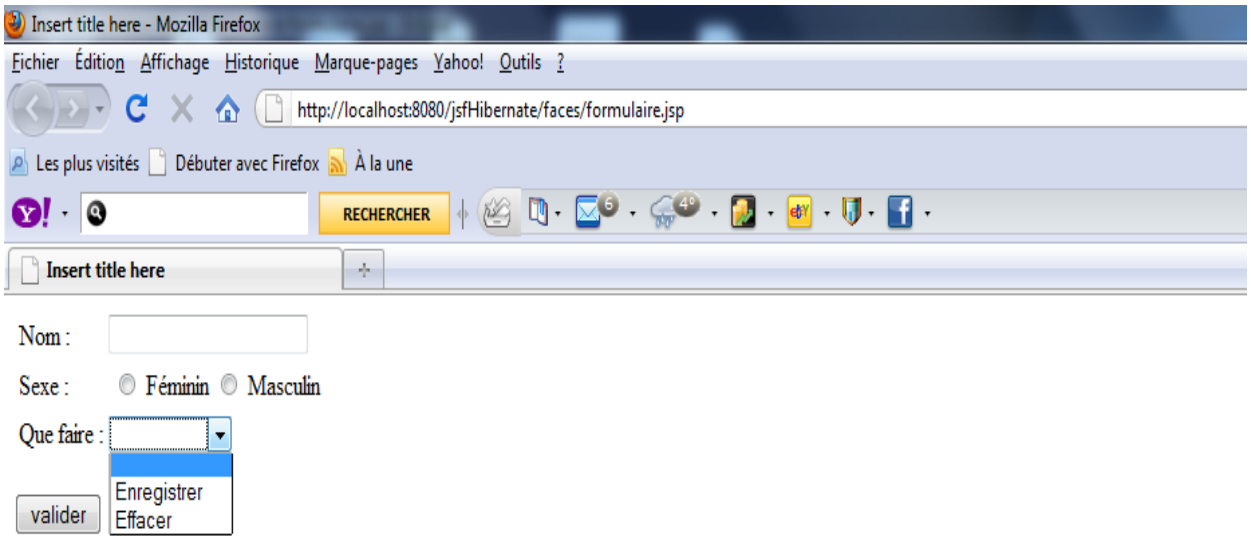
Le tag `<h:message for="idname">` permet d'afficher le message spécifié dans l'attribut **`requiredMessage`**, ici le message se rapporte au composant comportant l'**id** (l'identifiant) **idname .**
Le tag `<h:selectOneRadio>` permet d'afficher les boutons radios.

Le tag `<f:selectItem itemLabel="Féminin" itemValue="Féminin" id="itemsexe1"/>` associé conjointement avec le tag **<h:selectOneRadio>** permet de d'afficher les genres Féminin(ou Masculin) sur la page web via l'attribut **itemLabel**, l'attribut **itemValue** est la valeur qui sera mise dans la propriété d'un bean pour connaitre le genre du sexe.

Le tag `<h:selectOneMenu>` permet d'afficher un menu déroulant avec une seule possibilité de choix.

Le tag `<h:commandButton value="valider" action="confirmerDonnee">` permet de valider les données saisies, l'attribut value contient le texte qui sera afficher sur le bouton de validation, l'attribut action contient l'alias du nom du fichier de la page web suivante à afficher ou une fonction qui conduira vers cette page .

Pour exécuter le code précédent **faites un clic droit de souris sur le fichier ''formulaire.jsp''**, aller sur **Run as→Run on Server→finish.**
L'exécution du code précédent donnera le résultat suivant (voir illustration).



II-1.2.2 Les beans managés

La question que chacun peut se poser est de savoir comment les données sont transmises aux propriétés de beans ou comment peut-on solliciter une fonction via une interface utilisant la techno JSF, nous essaierons d'apporter dans cette section la réponse à cela.

Un bean managé est en réalité une classe java comportant des méthodes et des propriétés inter-réagissant avec les tags JSF, celui-ci doit être configuré dans le fichier **faces-config.xml** (**fichier de configuration JSF automatiquement généré, se trouvant dans /WEB-INF)** et est accessible via les expressions langages **(EL)**. Les EL sont des termes syntaxiquement définis par **#{}** et **${}.** Le premier terme **(#{})** permet ,outre les appels aux méthodes, un accès en lectures et en écriture sur les propriétés de bean , tandis que le second permet uniquement des accès en lecture sur les propriétés de bean et les appels des méthodes ayant la nomenclature **getXXXXX().**
Les accès aux propriétés et les appels de méthodes respectent les syntaxes suivantes :

- **#{nomDuBean.propriété}** pour un accès en lecture et écriture via un getter ou setter, **#{nomDuBean.méthode}**
- **${nomDuBean.propriété}** pour un accès en lecture seule via un getter**, ${nomDuBean.méthode}** entraine une erreur si la méthode ne commence pas par **« get »**

Remarque :

Les getters et les setters sont des fonctions respectant la nomenclature suivante getXX () et set XX() ou XX représente le nom de la propriété. Pour les deux cas nul n'est besoin de préfixer les appels de méthodes par le préfixe « get », le suffixe suffit
Ex : la méthode getNom () d'un bean nommé **personne** sera appelée par les expressions **#{personne.nom}** ou **${personne.nom }**

II-1.2.2.1 Configuration du bean managé

Commençant par définir notre classe java représentant le bean, dans notre exercice nous utilisons un formulaire, nous allons donc créer un bean ayant les propriétés **nom**, **prenom**, **age**, **sexe** et **date de naissance et autres propriétés indispensable à notre exercice**. Cette classe java sera définie dans un package appelé **model.personne,** et portera le nom **PersonneDTO.** Cette classe comportera des getters et setters associés à chaque propriété ainsi qu'un constructeur vide sans argument. Le code de la classe est t téléchargeable sur mon site.
Voici un extrait de configuration du bean dans le fichier **faces-config.xml**, celui-ci se trouve dans le dossier **WebContent/WEB-INF** du projet

```
<faces-config …….>
……
……..
<managed-bean>
  <managed-bean-name>personneDTO</managed-bean-name>
  <managed-bean-class>model.personne.PersonneDTO</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>XXXXXXXXXX</value>
                </managed-property>
</managed-bean>
……
……
</faces-config>
```

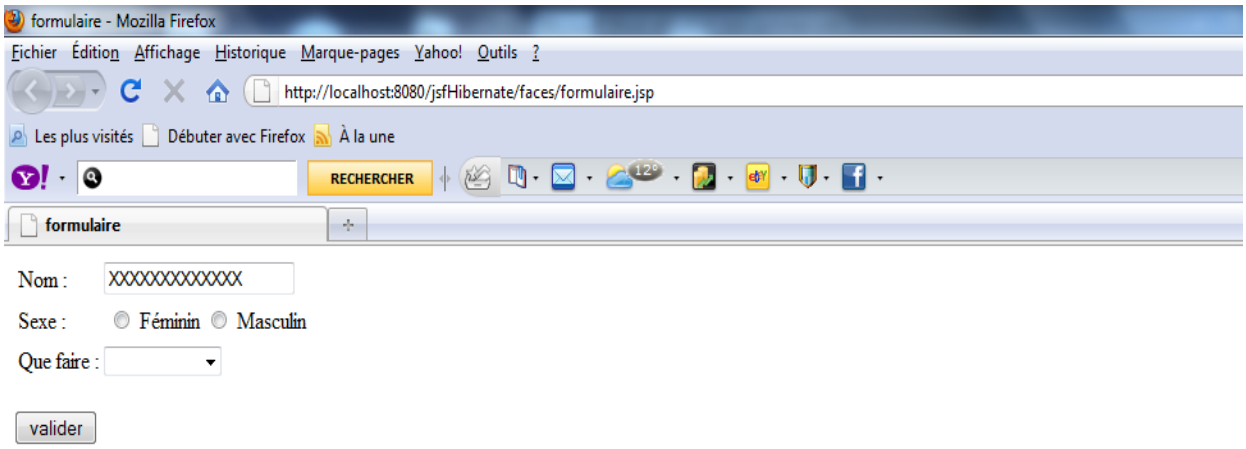
Petite explication :

L'élément `<faces-config>` **représente la racine ou l'élément parent de tous les éléments entrant dans la configuration de JSF**.
L'élément `<managed-bean>` permet de définir le bean, les élément enfants `<managed-bean-name>`, `<managed-bean-class>`, `<managed-bean-scope>` désignent respectivement son nom, sa classe et sa portée dont la valeur est **session** , l'initialisation par défaut des propriétés est faite dans la balise `<managed-property>`, ici la propriété **nom** à la valeur **XXXXXXXXX** par défaut.

Nous pouvons maintenant compléter le code de notre formulaire en affectant à l'attribut **value** des tags JSF les expressions suivantes :

```
<h:inputText id="idname"  value="#{personneDTO.nom}".....>
<h:selectOneRadio id="idChoixSexe" value="#{personneDTO.sexe}" .....>
<h:selectOneMenu id="idChoixMenu" value="#{personneDTO.action}".....>
```

Après avoir configuré notre bean et modifié notre code en exécutant notre code on aura l'illustration suivante



Nous remarquons bien la valeur par défaut affectée au Nom

II-1.2.3 Les règles de navigation

Elles permettent de définir la navigation d'une page web à une autre dans une application web. Une fois de plus le fichier de configuration sera sollicité pour configurer la navigation entre les différentes pages.

Revenons au code de notre formulaire, nous constatons que le tag de validation `<h:commandButton value="valider" action="confirmerDonnee">` dispose d'un attribut **action** avec la valeur **"confirmerDonnee"**, cette valeur est aussi appelée **outcome** dans le jargon JSF, elle est liée à un nom de fichier de page web. Ainsi quand on clique sur le bouton de validation on est redirigé vers la dite page.
Nous pouvons aussi affecter comme valeur à l'attribut **action** une fonction sans argument ayant une valeur de retour de type **String** qui sera bien sur **l'outcome** ou la page web à venir.

II-1.2.3.1 Configuration des règles de navigation

Notre formulaire départ a pour nom de fichier ''**formulaire.jsp'',** après la saisie des informations sur celui-ci nous aimerions que l'utilisateur ait une idée des données entrées. Ainsi après validation du formulaire il sera redirigé vers une autre page web, que nous appellerons **''confirmerDonnee.jsp''**, pour voir sa saisie et confirmer ses données.

Voici la configuration de cette navigation dans le fichier **faces-config.xml**

```
<navigation-rule>
            <display-name>formulaire</display-name>
            <from-view-id>/formulaire.jsp</from-view-id>
            <navigation-case>
                  <from-outcome>confirmerDonnee</from-outcome>
                  <to-view-id>/confirmerDonnee.jsp</to-view-id>
            </navigation-case>
</navigation-rule>
```

Petite explication :
L'élément `<navigation-rule>` permet de définir une règle de navigation. Les éléments enfants `<display-name>`, `<from-view-id>` désignent respectivement le nom d'affichage du formulaire et le point de départ(**le fichier formulaire.jsp**).
L'élément enfant `<navigation-case>` défini un cas de navigation, les éléments `<from-outcome>`, `<to-view-id>` désignent respectivement l'alias du nom du fichier de la page web à venir et le dit fichier. Donc nous partirons de **''formulaire.jsp''** vers **''confirmerDonnee.jsp''.**
L'outcome **''confirmerDonnee''** correspond à la valeur de l'attribut **action** dans le tag `<h:commandButton>`

Extrait du code de la page web « confirmerDonnee.jsp »

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<f:view>
<b>VOICI LES DONNEES SAISIES</b>
<hr>
<h:form>
 <h:panelGrid columns="2">
      <h:outputText value="Nom :"></h:outputText>
      <h:outputText value="#{personneDTO.nom}"></h:outputText>
```

……

……

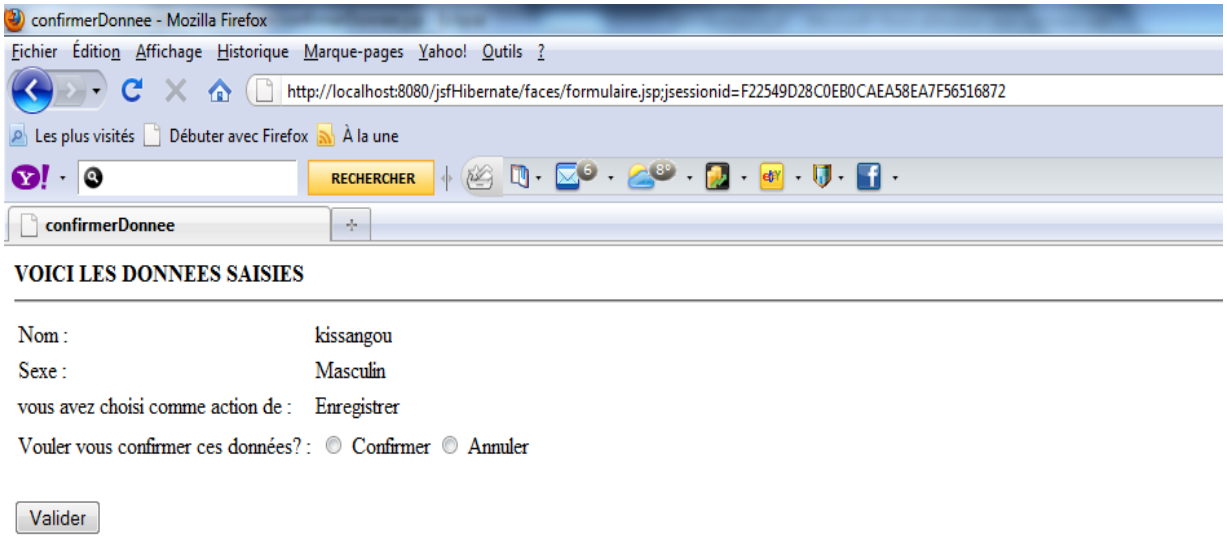
```
<h:outputText value="vous avez choisi comme action de :"></h:outputText>

      <c:choose >
            <c:when test="${personneDTO.action=='2'}">
               <h:outputText value="Effacer"></h:outputText>
            </c:when>
            <c:otherwise>
                <h:outputText value="Enregistrer"></h:outputText>
            </c:otherwise>
      </c:choose>
 </h:panelGrid>
```

……

L'élément nouveau est l'introduction de la bibliothèque **core** de **JSTL (JavaServer Pages Standard Tag Library)** pour prendre en compte les tags préfixé par **''c'',** le tag **<c :choose>** est conjointement utilisé avec les tags **<c :when>** et **<c :otherwise>** pour effectuer des test de condition, ainsi si la propriété **action** du bean **personneDTO** est égale à 2 on affiche « Effacer » sinon « Enregistrer »

Voici une illustration du code précédent après avoir saisi « kissangou » comme nom et validé le formulaire, on est redirigé vers la page « confirmerDonnee ».



II-1.2.4 La validation de données

Quand un utilisateur saisie une donnée qui n'est pas conforme avec ce que l'application attend, il bien normal de le lui faire savoir, aussi les règles de validation sont là pour permettre la validation des informations entrées.

La validation est faite par des validateurs, ce sont des classes java qui implémentent l'interface **javax.faces.Validator,** ou des classes java**(bean managé)** définissant une fonction au prototype ci-après :
nomDeFonction (FacesContex contex**, UIComponent** component**, Object** value**) ; .**

On distingue dans JSF les validateurs standards (natifs de JSF et directement utilisable) et les validateurs personnalisés (implémentés par un développeur).

Dans notre tutoriel nous implémenterons un validateur dont le but sera de vérifier les caractères saisies, ceux-ci doivent être conformes aux caractères alphabétiques du code ASCII. Nous allons par ailleurs faire usage d'un validateur standard(**LengthValidator**), défini dans le code par le tag **<f :validateLength>,** qui obligera l'utilisateur d'entrer au minimum deux caractère.

Une fois de plus le fichier **face-config.xml** sera solliciter pour, car l'utilisation d'un validateur personnalisé impose la configuration de ce fichier.

II-1.2.4.1 Configuration d'un validateur

La classe définissant notre validateur sera créée dans le package **model.personne.validator** elle aura pour nom **NameValidator,** le code de cette classe est téléchargeable sur mon site.

Ci-dessous la configuration du validateur dans le **faces-config.xml**

```
<validator>
      <display-name>nameValidator</display-name>
      <validator-id>nameValidator</validator-id>
<validator-class>model.personne.validator.NameValidator</validator-class>
</validator>
```

Petite explication :
L'élément `<validator>` permet de définir un validateur. Les élément enfants `<display-name>, <validator-id>,<validator-class>` désignent respectivement le nom d'affichage utilisé pour reconnaitre le validateur, l'identifiant du validateur utilisé dans le code pour faire appel à ce validateur, la classe de définition du validateur

Extrait de code du formulaire modifié

```
<f:view>
<h:form>

<h:panelGrid columns="3">

                    <!- - ENTRER VOTRE NOM -->
            <h:outputText value="Nom : "></h:outputText>
<h:inputText id="idname"  value="#{personneDTO.nom}"
```

```
                        required=”true”
                        requiredMessage=”ce champ de doit pas être vide” >
        <f:validateLength minimum=”2”/>
        <f:validator validatorId=”nameValidator”/>
</h:inputText>
……
……
```

II-1.2.5 Les convertisseurs

Les convertisseurs sont des classes java implémentant l'interface **javax.faces.Converter**
Ils interviennent quand on désire établir une adéquation entre une donnée saisie et le type d'une propriété d'un Bean qui recevra la dite donnée.
Nous distinguons des convertisseurs standards et ceux qui peuvent être personnalisés. Dans Notre tutoriel nous utiliserons un convertisseur standard(**DateTimeConverter**), permettant la conversion d'une chaine de caractères particulière en format de date, défini dans le code par le tag (**<f :convertDateTime>**), puis un autre que nous allons implémenter qui converti une chaine de caractère définissant l'âge d'une personne en entier. Notre cher fichier faces-config.xml nous sera encore utile ici.

II-1.2.5.1 Configuration d'un convertisseur

Nous nommerons notre classe **AgeConverter,** celle-ci sera créée dans le package **model.pesonne.converter,** le code de cette classe est téléchargeable sur mon site.

Ci-dessous la configuration du convertisseur dans le fichier faces-config.xml

```
<converter>
      <display-name>ageConverter</display-name>
      <converter-id>ageConverter</converter-id>
<converter-class>model.personne.converter.Ageconverter</converter-class>
</converter>
```

Petite explication :
Les explications sont les mêmes que celles du validateur, excepté le fait que le mot validateur est remplacé par convertisseur.

Extrait de code du formulaire modifié

```
<f:view>
<h:form>
<h:panelGrid columns="3">
```

……

……

```
                <!- - ENTRER VOTRE Age -->
<h:outputText value="Age : "></h:outputText>

      <h:inputText id="idAge" value="#{personneDTO.age}">
            <f:converter converterId="ageConverter"/>
      </h:inputText>
```

……

……

```
            <!- - ENTRER VOTRE DATE DE NAISSANCE -->
<h:outputText value="Date de naissance : "></h:outputText>
<h:inputText id="idBirthDay" value="#{personneDTO.dateDeNaissance}"
            converterMessage="votre date n'est pas au format dd/mm/yyyy">
      <f:convertDateTime  type="date" dateStyle="short"/>
</h:inputText>
```

……

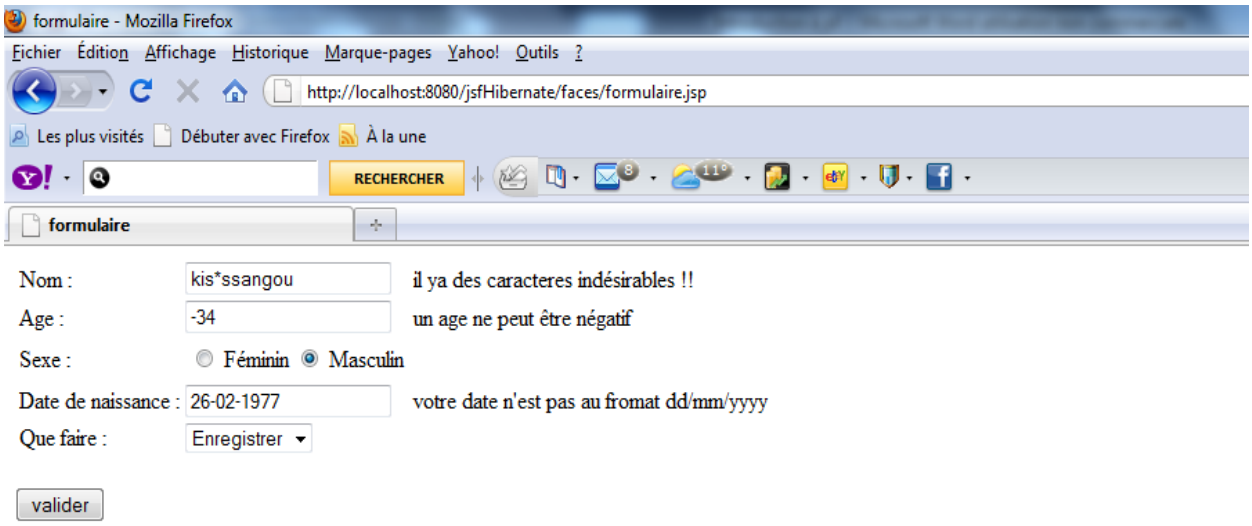
……

Petite explication:

L'attribut **converterMessage** du tag **<h :inputText>** permet d'afficher un message d'erreur lié à un mauvais format de date entré, ce message est envoyé par le convertisseur **DateTimeConverter,** défini ici par le tag **<f :convertDateTime>.**

L'attribut **type** informe qu'on attend une date **(l'heure ne sera donc pas affiché).** L'attribut **dateStyle** nous impose à rentrer une date au format **dd/mm/yyyy.**

Voici une illustration après validation du formulaire quand des données non désirées sont entées.


formulaire - Mozilla Firefox
Fichier Édition Affichage Historique Marque-pages Yahoo! Outils ?
http://localhost:8080/jsfHibernate/faces/formulaire.jsp
Les plus visités Débuter avec Firefox À la une
RECHERCHER
formulaire
Nom : kis*ssangou il ya des caracteres indésirables !!
Age : -34 un age ne peut être négatif
Sexe : Féminin Masculin
Date de naissance : 26-02-1977 votre date n'est pas au fromat dd/mm/yyyy
Que faire : Enregistrer
valider

II- 1.2.6 Le backing bean

Le **backing bean** est un **bean** comportant des propriétés de type **UIComponent,** via ces propriétés nous pouvons directement avoir accès aux références de composants graphiques représentés par des tags, ce principe est aussi appelé **binding** (liaison d'un composant graphique avec une propriété d'un Bean de type **UIComponent**).

Pour avoir accès aux références de composants nous devons passer par l'intermédiaire d'un bean managé, nous définirons donc un bean managé qui fera office de validateur, son rôle sera de vérifier la conformité entre l'âge et l'année entrée.

Ce bean sera nommé **AgeYearValidator**, il sera créé dans le package **model.personne.validator**, il comportera une propriété de type **UIInput** pour le binding avec le composant graphique permettant la saisie de l'âge. Le code de ce bean est téléchargeable sur mon site.

Pour la configuration d'un bean managé dans le faces-config.xml voir la section traitant les beans managés.

Extrait de code du formulaire modifié

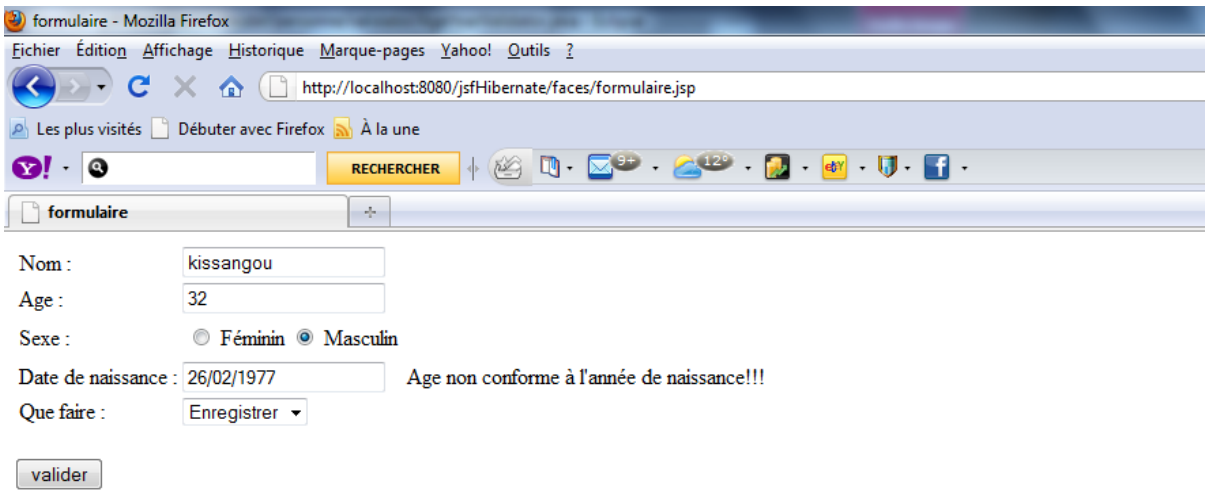
……..

……..

```
<!- - ENTRER VOTRE Age -->
<h:outputText value="Age : "></h:outputText>
      <h:inputText id="idAge" value="#{personneDTO.age}"
      binding="#{ageYearVailidator.age}">
            <f:converter converterId="ageConverter"/>
      </h:inputText>

……..
……
            <!- - ENTRER VOTRE DATE DE NAISSANCE -->
<h:outputText value="Date de naissance : "></h:outputText>
<h:inputText id="idBirthDay" value="#{personneDTO.dateDeNaissance}"
            converterMessage="votre date n'est pas au format dd/mm/yyyy"
            validator="#{ageYearVailidator.compareAgeAnnee}">
      <f:convertDateTime  type="date" dateStyle="short"/>
</h:inputText>
```

Voici une illustration quand l'âge entrée n'est pas conforme à l'année de naissance après validation du formulaire



II- 1.2.7 L'internationalisation

C'est le principe qui consiste à rendre son application web multilingue, pour le faire il faudra définir une famille de fichiers textes(ou **RessourceBundle** dans le jargon JSF) avec un identifiant représentant la famille, puis les code ISO appropriés des langues et des pays et un « **.properties** » comme extension des fichiers textes.

Un nom de fichier définissant un **RessourceBundle** respecte donc la syntaxe suivante :

IdentifiantDefamille_codeISO639Langue_codeISO3166Pays.properties

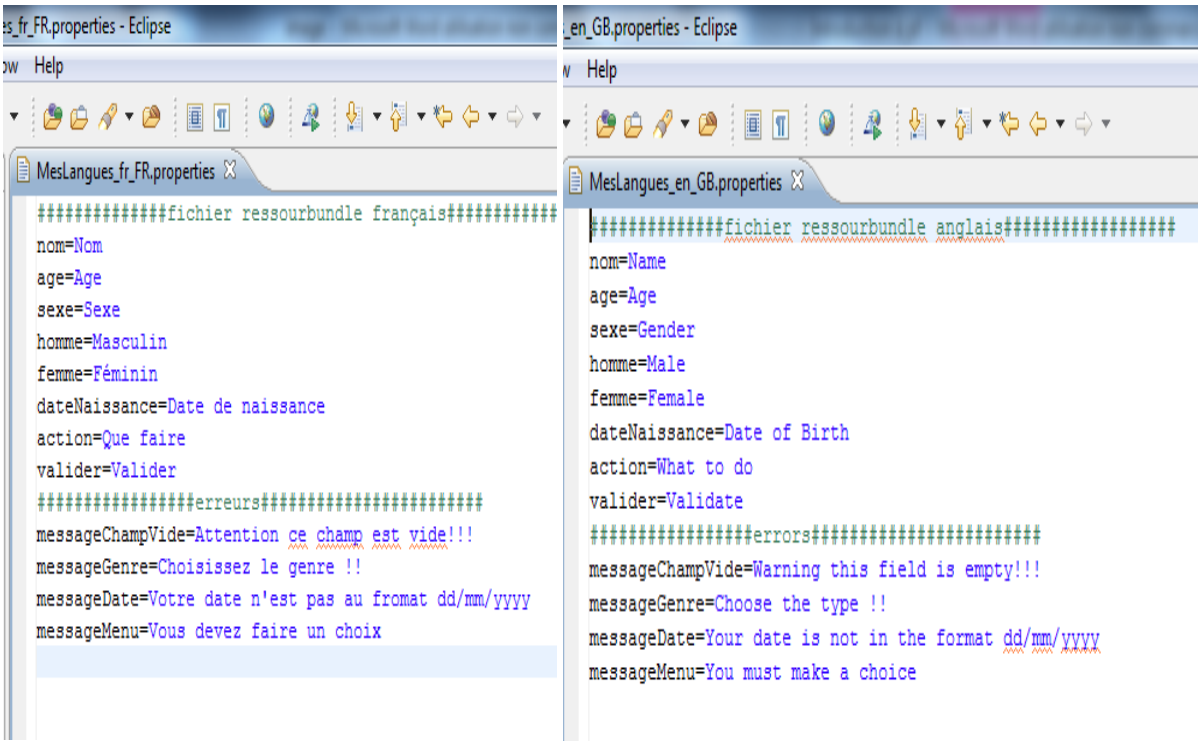
- L'identifiant de la famille correspond à une chaine de caractères quelconque.
- Le code ISO639 représente la langue, par exemple le français a pour code ISO639 **« fr ».**
- Le code ISO3166 représente un pays ou une région, par exemple la France a pour code ISO3166 **« FR ».**

Les RessourceBundle de même famille ont un même identifiant
EX : **MesLangues_fr_FR.properties** et **MesLangues_en_GB.properties**

Une fois les familles définies, les fichiers doivent contenir des **clés** et des **valeurs**. Les clés dans les fichiers d'une même famille doivent être identiques, les valeurs par contre doivent correspondre à la langue spécifiée dans le nom du fichier.
On peut comparer une clé à une variable contenant une valeur.

EX :
Le fichier **MesLangues_fr_FR.properties** contient le texte suivant : **nom**=**Nom**.
Le fichier **MesLangues_en_GB.properties** contient celui-ci : **nom**=**Name**
nom est la clé, elle est identique pour les deux fichiers**. Nom** et **Name** sont les valeurs de la clé selon la langue.

Voici une illustration



Après avoir défini les clés et les valeurs nous devons enregistrer les fichiers dans un dossier source et configurer **le faces-config.xml** pour les utiliser.
Les tests seront effectués en modifiant les préférences du navigateur au niveau des langues.

Dans notre tutoriel notre application sera bilingue, parlant le français et l'anglais. Nous créerons deux RessourceBundle (MesLangues_fr_FR.properties et MesLangues_en_GB.properties) , un dossier source **ressources** et un package nommé **langue** qui contiendra les RessourceBundle**.**

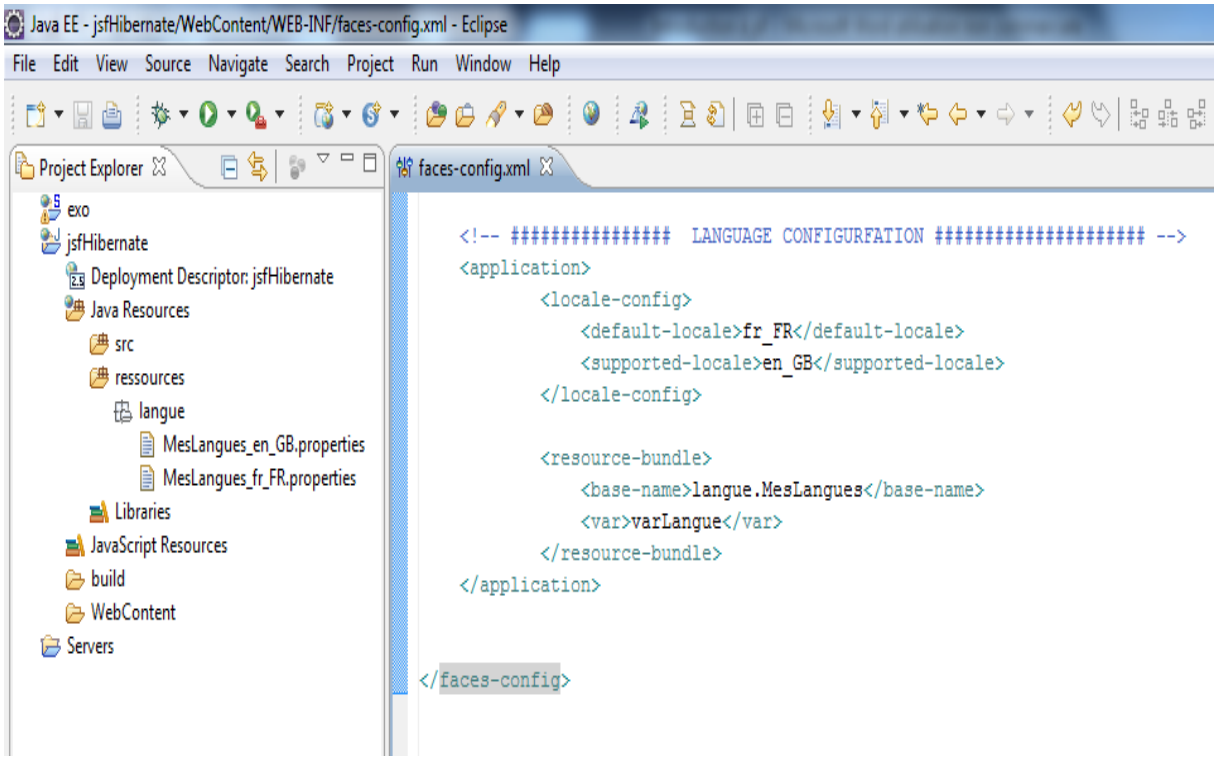
II- 1.2.7.1 Configuration d'un RessourceBundle

Extrait du fichier **faces-config.xml**

```
<application>
                  <locale-config>
                        <default-locale>fr_FR</default-locale>
                        <supported-locale>en_GB</supported-locale>
                  </locale-config>

                  <resource-bundle>
                        <base-name>langue.MesLangues</base-name>
                        <var>varLangue</var>
                  </resource-bundle>
</application>
```

Illustration de la configuration



Petite explication :

Toute la configuration doit être insérée dans l'élément `<application>`.
La langue par défaut est le français d'où l'utilisation de **fr_FR** dans l'élément `<default-locale>`, la langue supportée est l'anglais d'où le **en_GB**. L'élément `<base-name>` désigne l'emplacement de la resource, vous remarquerez que seul le nom de famille est utilisé(MesLangues), l'élément `<var>` permet de définir une sorte de variable donnant accès aux clés des fichiers représentant les RessourceBundle.

Extrait de code du formulaire modifié

……..

……..

```
<!- - ENTRER VOTRE Age -->
<h:outputText value="#{varLangue.nom} : "></h:outputText>
      <h:inputText id="idAge" value="#{personneDTO.age}"
      binding="#{ageYearVailidator.age}">
            <f:converter converterId="ageConverter"/>
      </h:inputText>

……..
……..
            <!- - ENTRER VOTRE DATE DE NAISSANCE -->
<h:outputText value="#{varLangue.dateNaissance} : "></h:outputText>
<h:inputText id="idBirthDay" value="#{personneDTO.dateDeNaissance}"
            converterMessage="#{varlangue.messageDate}"
            validator="#{ageYearVailidator.compareAgeAnnee}">
      <f:convertDateTime  type="date" dateStyle="short"/>
</h:inputText>
```

nous remarquons que l'accès au clés se fait, par la **''variable'' « varLangue** » définie dans le fichier **faces-config.xml,** via la syntaxe **#{varLangue.Clé}**. Ainsi l'affichage des messages sera automatique, et les langues seront modifiées automatiquement selon la configuration dans les préférences de langue dans votre navigateur.

III-Couplage avec Hibernate

III-1.0 Hibernate qu'est à dire ?

Est un Framework permettant le mapping objet relationnel, les données enregistrées dans une base donnée relationnelle sont donc considérées comme des objets, il crée une abstraction entre les couches métiers et les couches de persistance de données. Les accès aux informations sont exposés aux autres couches comme des services.

III-1.1 Entrée en matière

III-1.1.1 Installation d'Hibernate

- Téléchargez l'archive du module principal (**Hibernate core**) à l'adresse suivant : http://sourceforge.net/projects/hibernate/files/hibernate3/

Nous avons pour notre tutoriel téléchargé l'archive ci-après : **hibernate-distribution-3.5.0-CR-2-dist.**

Récupérez après avoir décompressé l'archive les librairies suivantes : **hibernate3.jar**, **slf4j-log4j12.jar**, **log4j-1.2.15.jar**, et tous les **".jar "** contenus dans le sous-dossier **/lib/required.**
Par ailleurs nous aurons aussi besoin de la librairie **jpa-api-2.0-cr-1.jar** se trouvant à cette adresse :

http://repository.jboss.org/maven2/org/hibernate/java-persistence/jpa-api/2.0-cr-1/

Toutes les librairies doivent être placées dans le répertoire **/WEB-INF/lib** de notre projet

- Pour le confort du travail dans éclipse (**pour la version galiléo SR2**) nous aurons aussi besoin de **Hibernate Tools**, l'archive est téléchargeable à l'adresse suivante: http://www.jboss.org/tools/download/stable/3_1_GA.html

Nous avons pour notre tutoriel téléchargé l'archive ci-après : **HibernateTools-3.3.1.v201006011046R-H111-GA**

Cette archive contient deux sous-dossiers (**plugins** et **features**), leur contenu doit être placé respectivement dans les sous-dossiers identiques se trouvant dans l'installation de l'atelier d'éclipse.

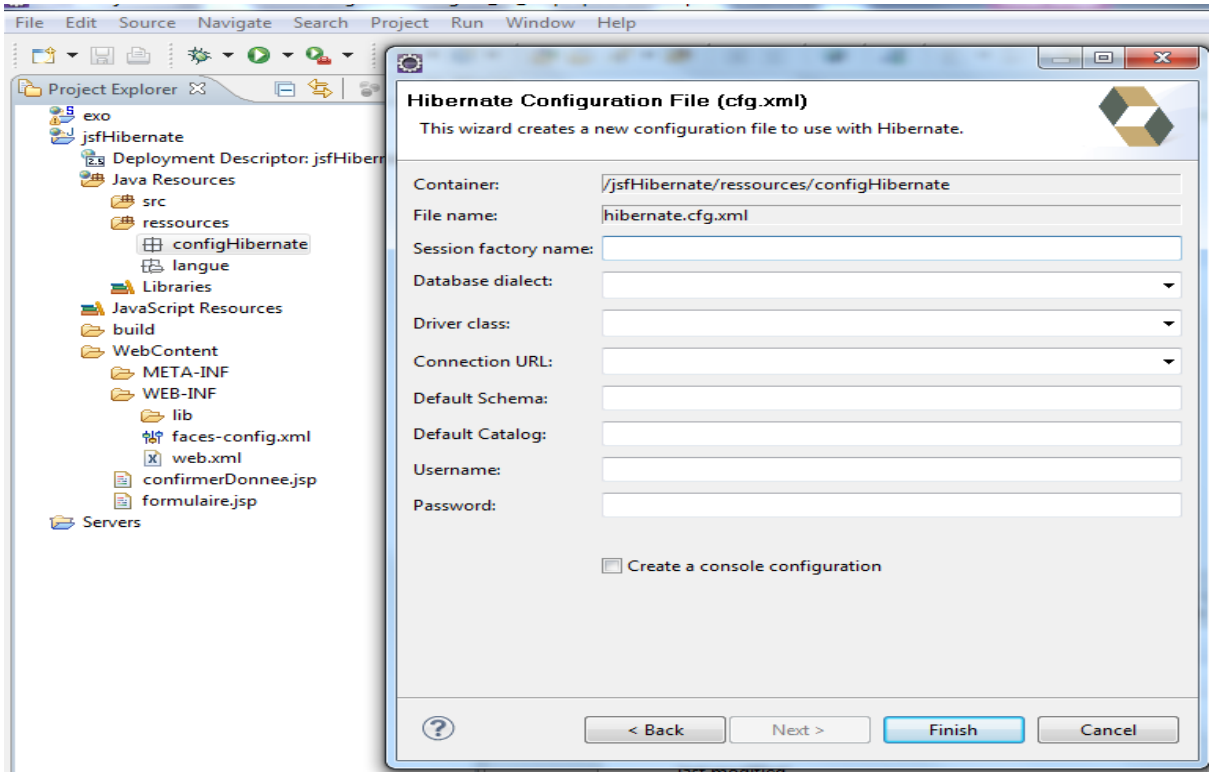
III-1.1.2 Configuration d'Hibernate

Nous considérons qu'un serveur de base de données a déjà été installée, qu'une base de données y a été créée (**BasePersonne)**, qu'une table de y a été insérée (**Personne**), et qu'un compte(**ComptePersonne**) de mot de passe(**ComptePersonne**) a aussi été créé avec tous les droits sur la base de données(**BasePersonne)**.

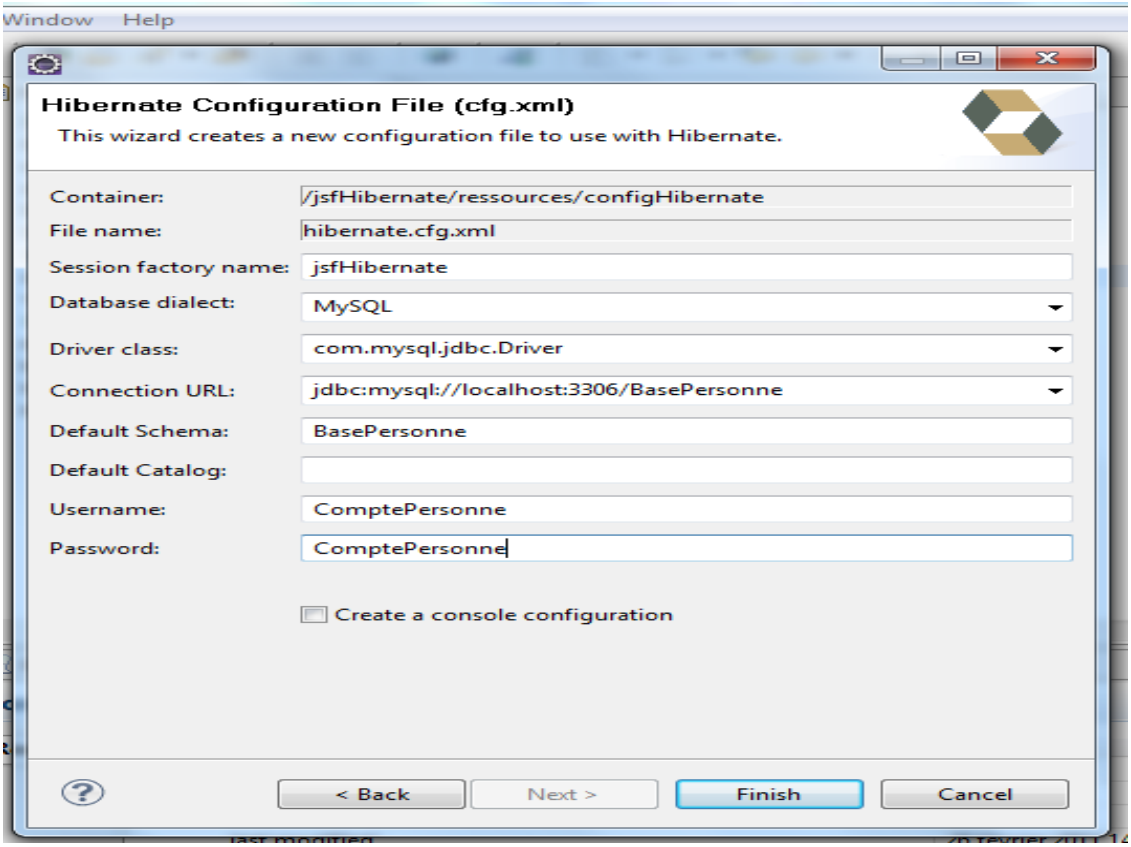
Nous avons fait usage pour notre tutoriel de **MYSQL Server 5.0.67.** Par ailleurs la connexion entre Hibernate et MySQL nécessitera un connecteur **(MySQL Connector/J),** aussi l'archive suivante **(mysql-connector-java-5.1.13-bin.jar)** a été complétée dans répertoire **/WEB-INF/lib** de notre projet. Les archives de téléchargement sont disponibles à l'adresse http://dev.mysql.com/downloads.

- Créez un package dans le dossier source **resources**, nommé le **configHibernate**

- Faites un clic droit de souris sur le package créé et allez dans **File→New→Other→Hibernate→**Choisissez **Hibernate Configuration File(cfg.xml)→**puis cliquer sur **Next→**laisser le nom par défaut et cliquez sur **Next.**
Vous aurez l'image suivante



Remplissez-le comme suit



Puis cliquez sur **finish**

III-1.1.3 Utilisation d'Hibernate

Pour utiliser Hibernate nous devons mapper notre classe **PersonneDTO** avec la table **Personne** de la base de données **BasePersonne** via les annotations de la **JPA (Java Persistence API)** et le fichier de configuration d'Hibernate ci-dessus créé**.**

La matérialisation dans le fichier de configuration se fait par l'ajout de l'élément **<mapping/>**, comme suit **<mapping class= ''model.personne.PersonneDTO''/>**

Le code de la classe est téléchargeable sur mon site.

IV-Structure logique, diagramme de séquence et cas d'utilisation de l'application

Structure logique

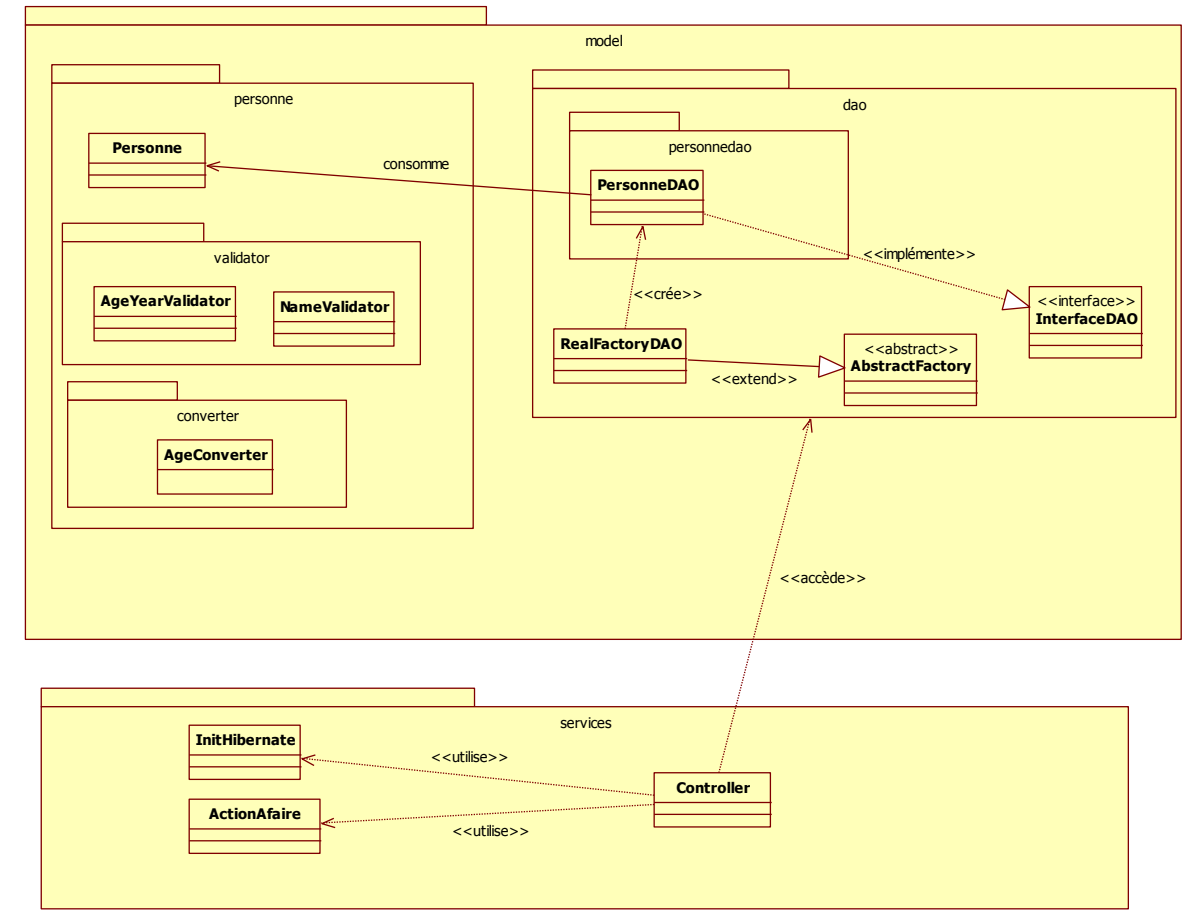
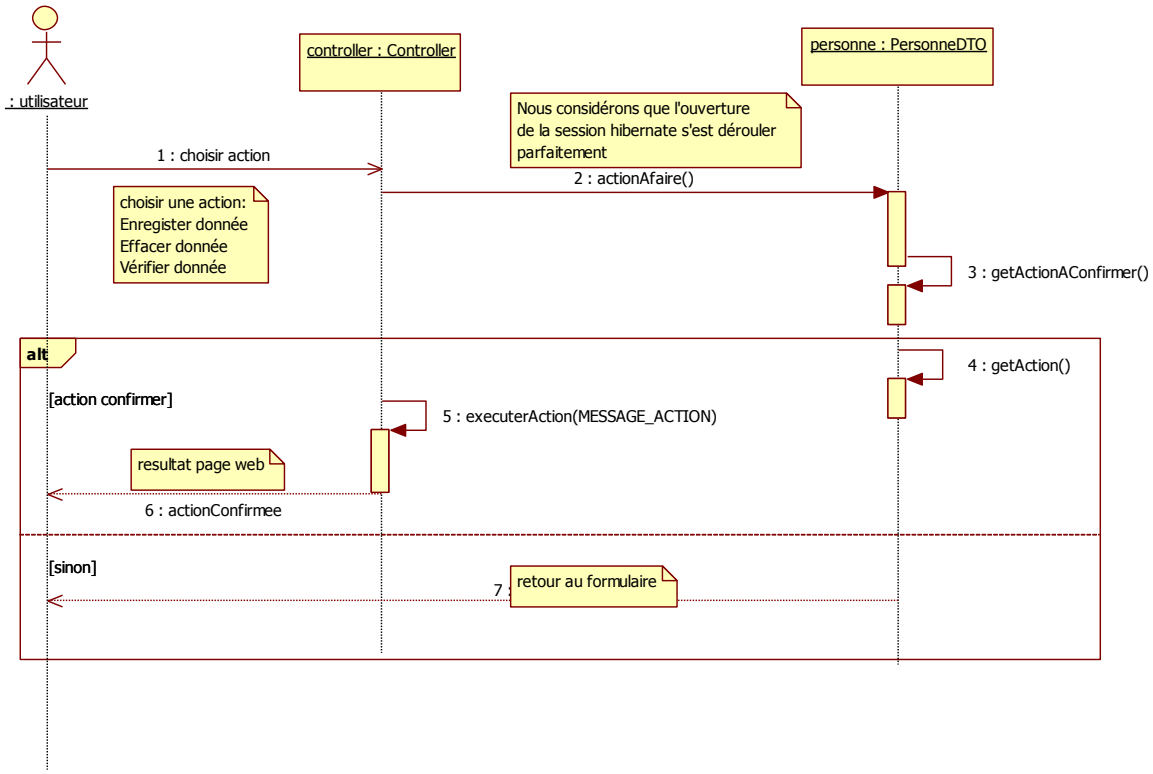


Diagramme de séquence



Cas d'utilisation