



Ingénierie de données et
développement logiciel-
IDDLO

Développement Java EE sous l'IDE
NetBeans

Module : Architecture Java j2ee.



Encadré par : Mr le Professeur Anouar RIAD SOLH

Réalisé par : - Redouane RAMI

بِسْمِ اللَّهِ الرَّحْمَنِ الرَّحِيمِ

Table des matières

I-Environnement Java EE 7

1.1-Configuration de l'IDE NetBeans

II-Développement des applications Web avec JavaServer Faces

2.1- Création de notre JSF Application.

2.1.1-Création de nouveau projet

2.1.2-Modification sur la page.

2.1.3- Création du CDI bean(Beau Nommé)

2.1.4-Implementer la page de confirmation.

2.1.5-Execution & JSF Validation

2.2- Les Templates d'une Facelet.

2.3-Composants composites

2.4-Les FaceFlows.

2.5-Support HTML5.

III-Composants JSF

3.1- PrimeFaces

3.2-ICEFaces.

3.3-RichFaces

IV-Se Connecter à une Base de données via JAVA Persistence API

4.1- Création d'une entité JPA.

4.1.1-Ajout des champs.

4.1.2-Création d'un DAO(Data Access Object).

4.2-Auto-génération des entités JPA.

4.2.1- Requêtes nommées et JPQL.

4.2.2-Bean Validation

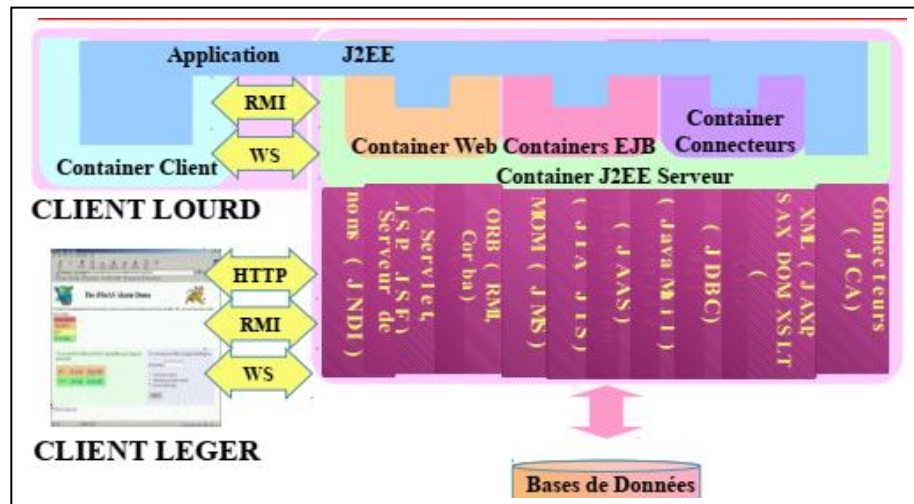
4.2.3-Relation des entités.

4.3-Générer les JSF via les entités JPA.

I-Environnement Java EE 7

Le terme « Java EE » signifie *Java Enterprise Edition*, et était anciennement raccourci en « J2EE »

➔ Architecture JAVA EE :



1.1-Configuration de l'IDE NetBeans

La plate forme NetBeans est un environnement de développement intégré (Integrated Development Environment IDE). A partir de la version 6, NetBeans supporte plusieurs langages de programmation : Java, JavaFX, C, C++ et PHP. Groovy, Scala, Ruby et d'autres sont pris en charge via des plug-ins supplémentaires.

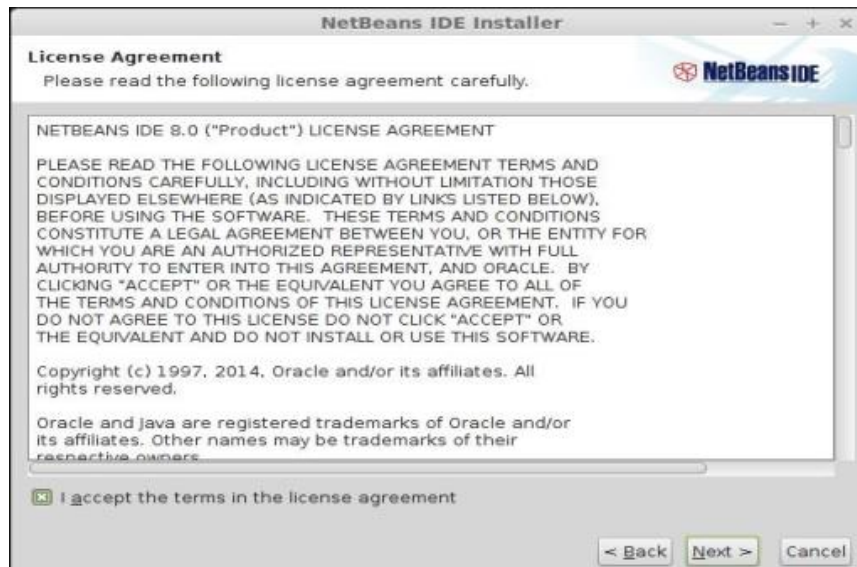
a)-Installation

NetBeans 8.0 nécessite l'installation d'un Java Development Kit (JDK) version 7.0 ou plus récente (la version choisie ici est 8.0u25). **Attention** l'installation d'un JRE n'est pas suffisante.

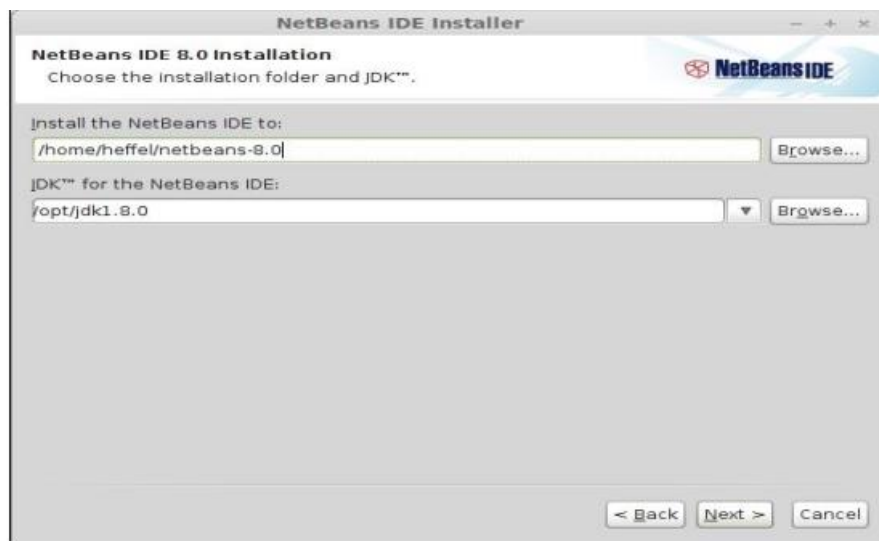
Après l'exécution du fichier d'installation de NetBeans 8.0, on doit voir une fenêtre semblable à celle illustrée ci-dessous :



Pour continuer l'installation de NetBeans, cochez les deux serveurs et cliquez sur le bouton Suivant. Il faut accepter les termes de la licence si on veut continuer l'installation



A cette étape l'installateur va nous inviter à choisir un répertoire d'installation de NetBeans, et celui du JDK à utiliser avec NetBeans. On peut soit choisir de nouvelles valeurs ou prendre celles fournies par défaut.



Une fois les répertoires d'installation appropriés sont choisis, on doit cliquer sur le bouton Suivant pour continuer l'installation. NetBeans utilise la variable d'environnement `JAVA_HOME` pour référencer l'emplacement du répertoire d'installation de JDK.

L'installateur va maintenant nous inviter à spécifier un répertoire d'installation pour le serveur GlassFish, on peut soit choisir un répertoire ou prendre la valeur par défaut.



Dans l'étape suivante de l'assistant, l'installateur va nous inviter à un répertoire d'installation pour Tomcat (si vous l'aviez coché auparavant), un conteneur de servlets très populaire, qui est livré avec NetBeans



L'installateur affiche alors un résumé des choix effectués.

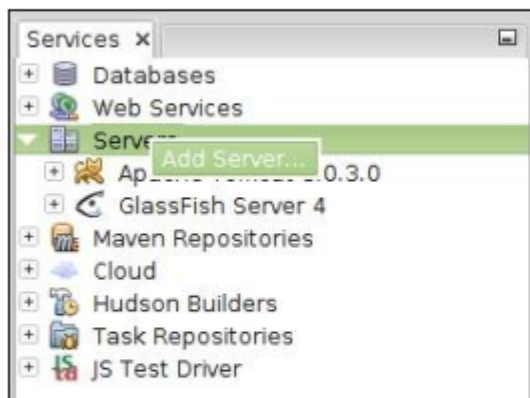
NetBeans est préconfiguré avec le serveur d'application GlassFish 4, et avec le SGBDR JavaDB. Si nous souhaitons utiliser GlassFish 4 et JavaDB, il n'y a rien à configurer. On peut choisir d'intégrer avec NetBeans d'autres serveurs tels que JBoss, Weblogic, WebSphere ou d'autres systèmes de bases de données relationnelles telles que MySQL, PostgreSQL, Oracle...

L'intégration de NetBeans avec un serveur d'application est très simple, elle se fait en suivant les étapes suivantes:

1. Premièrement, cliquez sur la fenêtre | Services.



2. Ensuite, on doit faire un clique droit sur le nœud Serveurs dans la Fenêtre Services, sélectionner Ajouter un serveur ...dans le menu contextuel.



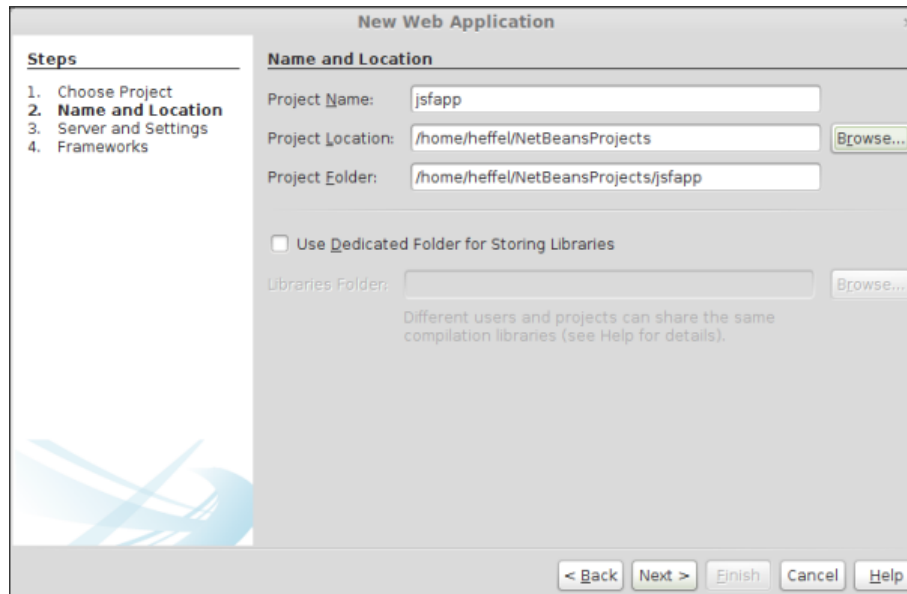
3. Ensuite, on doit sélectionner le serveur à installer dans la liste déroulante, puis cliquer sur le bouton Suivant.

4. On a besoin alors de saisir l'emplacement d'installation du serveur souhaité et cliquer sur Suivant.

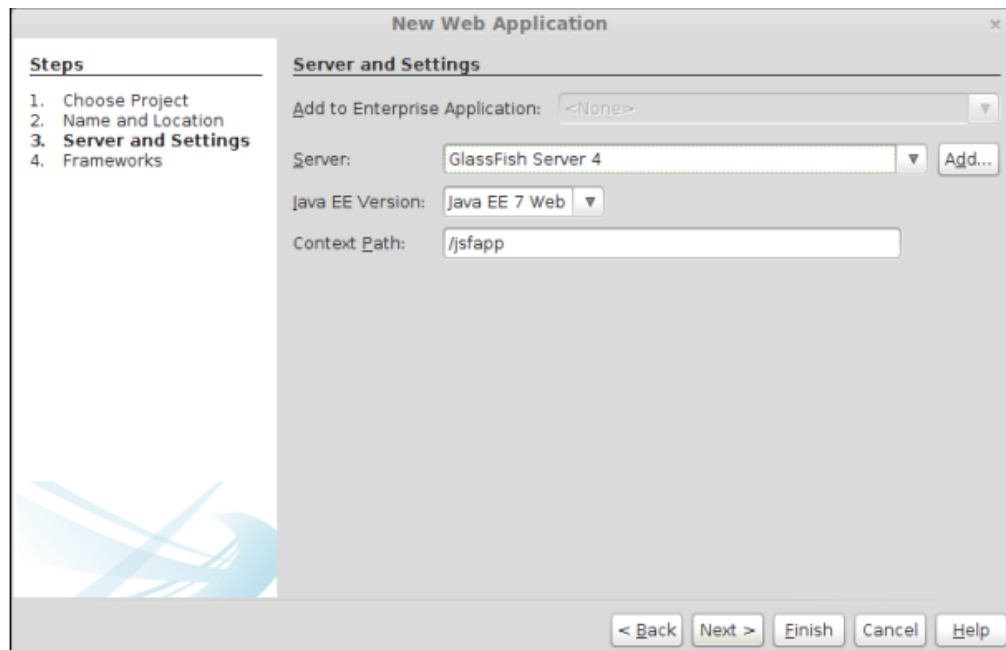
II-Développement des applications Web avec JavaServer Faces

Créer un nouveau projet JSF

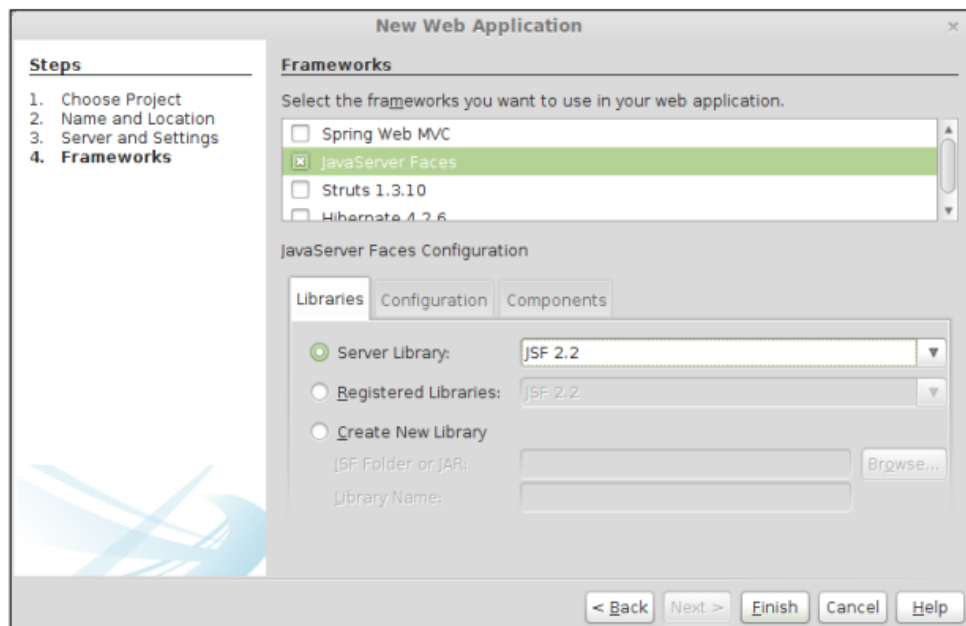
Pour créer un nouveau projet JSF, on doit aller dans **Fichier | Nouveau projet**, sélectionnez la catégorie **Java Web**, et **Web application** comme type de projet.



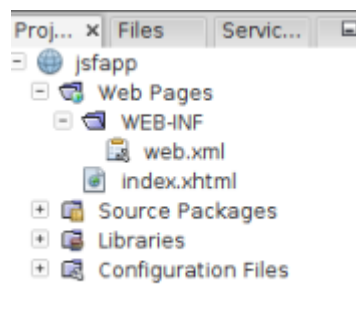
Après avoir cliqué sur **Suivant**, faites entrer le nom de projet, et vous pouvez aussi modifier les autres informations pour le projet. NetBeans fournit un choix par défaut, qui est recommandé.



Sur la **page suivante** de l'assistant, on peut choisir le framework qui sera utilisé par notre application web. Pour les applications JSF, on doit sélectionner le framework JSF.



En cliquant sur **Terminer**, l'Assistant nous génère un squelette de projet JSF, composé d'un unique fichier facelet **index.xhtml**, un fichier de configuration **web.xml**. Ce fichier est optionnel avec Java EE 7.



```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd">
  <context-param>
    <param-name>javax.faces.PROJECT_STAGE</param-name>
    <param-value>Development</param-value>
  </context-param>
  <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>
  <session-config>
    <session-timeout>
      30
    </session-timeout>
  </session-config>
  <welcome-file-list>
    <welcome-file>faces/index.xhtml</welcome-file>
  </welcome-file-list>
</web-app>
```

```
</welcome-file-list>
</web-app>
```

Les éléments suivants sont des valeurs valides pour le paramètre contexte **javax.faces.PROJECT_STAGE** de la Faces servlet :

- Développement
- Production
- SystemTest
- UnitTest

La classe **javax.faces.application.Application** a une méthode **getProjectStage()** qui nous permet d'obtenir l'état actuel du projet. En se basant sur cette valeur, on peut mettre en œuvre un code qui ne sera exécutée que dans l'étape appropriée.

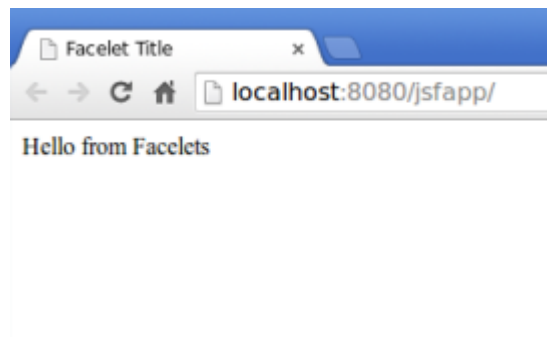
Lorsqu'on crée un projet Java Web en utilisant JSF, une facelet est générée automatiquement. Le fichier facelet généré ressemble à ceci:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://xmlns.jcp.org/jsf/html">
    <h:head>
        <title>Facelet Title</title>
    </h:head>
    <h:body>
        Hello from Facelets
    </h:body>
</html>
```

Une facelet n'est rien d'autre qu'un fichier XHTML utilisant certains espaces de noms XML. Dans cette page générée automatiquement, la définition de **namespace** nous permet d'utiliser "f" pour HTML, la bibliothèque de composants JSF:

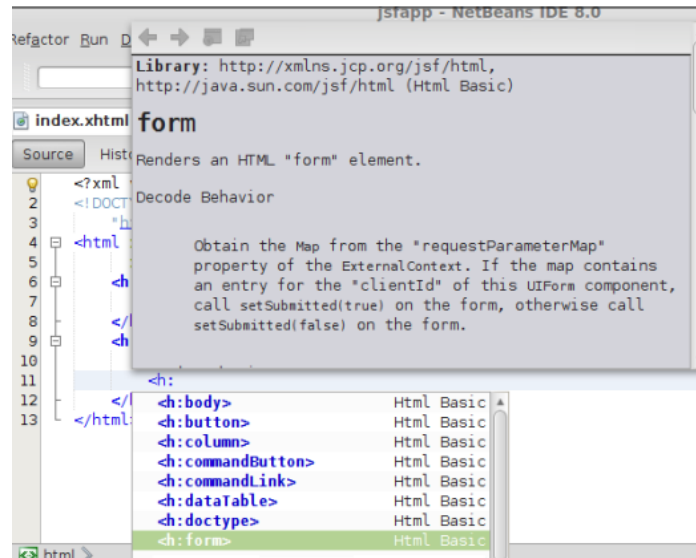
```
xmlns:f= "http://xmlns.jcp.org/jsf/core"
```

Nous pouvons le voir en cliquant droit sur notre projet dans la fenêtre projet et en sélectionnant Exécuter.



2.1.2-Modification sur la page.

On va modifier le fichier d'accueil généré **index.xhtml** pour collecter des données de l'utilisateur. Pour cela, on doit ajouter une balise **<h:form>** à notre page. Après avoir tapé les premiers caractères de la balise **<h:form>**, et en tapant **Ctrl + Espace**, on peut ainsi voir la liste des propositions offertes par NetBeans.



On ajoute la balise **<h:form>** et un certain nombre de balises JSF supplémentaires.

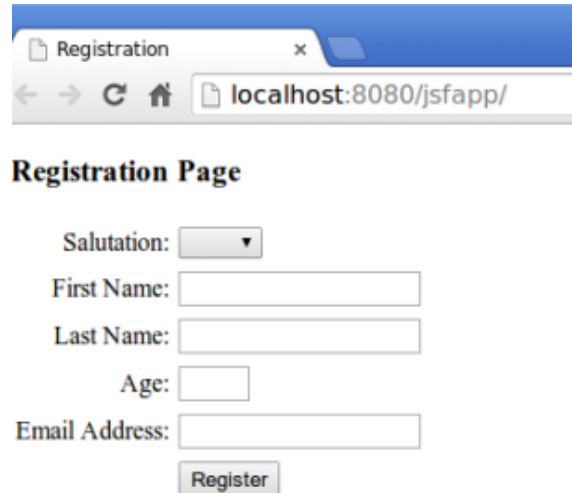
```
<h:form>
<h:panelGrid columns="3"
columnClasses="rightalign,leftalign,leftalign">
<h:outputLabel value="Salutation: " for="salutation"/>
<h:selectOneMenu id="salutation" label="Salutation"
value="#{registrationBean.salutation}" >
<f:selectItem itemLabel="" itemValue=""/>
<f:selectItem itemLabel="Mr." itemValue="MR"/>
<f:selectItem itemLabel="Mrs." itemValue="MRS"/>
7
<f:selectItem itemLabel="Miss" itemValue="MISS"/>
<f:selectItem itemLabel="Ms" itemValue="MS"/>
<f:selectItem itemLabel="Dr." itemValue="DR"/>
</h:selectOneMenu>
<h:message for="salutation"/>
<h:outputLabel value="First Name:" for="firstName"/>
<h:inputText id="firstName" label="First Name"
required="true"
value="#{registrationBean.firstName}" />
<h:message for="firstName" />
<h:outputLabel value="Last Name:" for="lastName"/>
<h:inputText id="lastName" label="Last Name"
required="true"
value="#{registrationBean.lastName}" />
<h:message for="lastName" />
<h:outputLabel for="age" value="Age:"/>
<h:inputText id="age" label="Age" size="2"
value="#{registrationBean.age}"/>
<h:message for="age"/>
<h:outputLabel value="Email Address:" for="email"/>
<h:inputText id="email" label="Email Address"
```

```

required="true"
value="#{registrationBean.email}" />
</h:inputText>
<h:message for="email" />
<h:panelGroup/>
<h:commandButton id="register" value="Register"
action="confirmation" />
</h:panelGrid>
</h:form>

```

L'exécution de notre application affiche:



Registration Page

Salutation:

First Name:

Last Name:

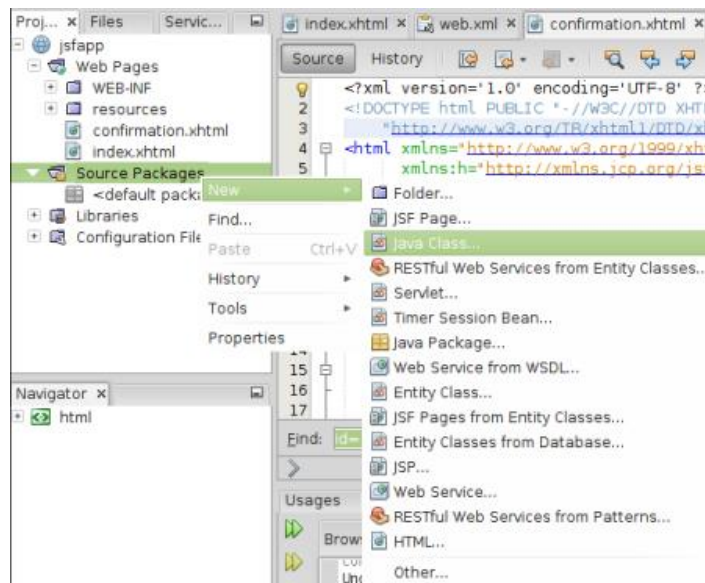
Age:

Email Address:

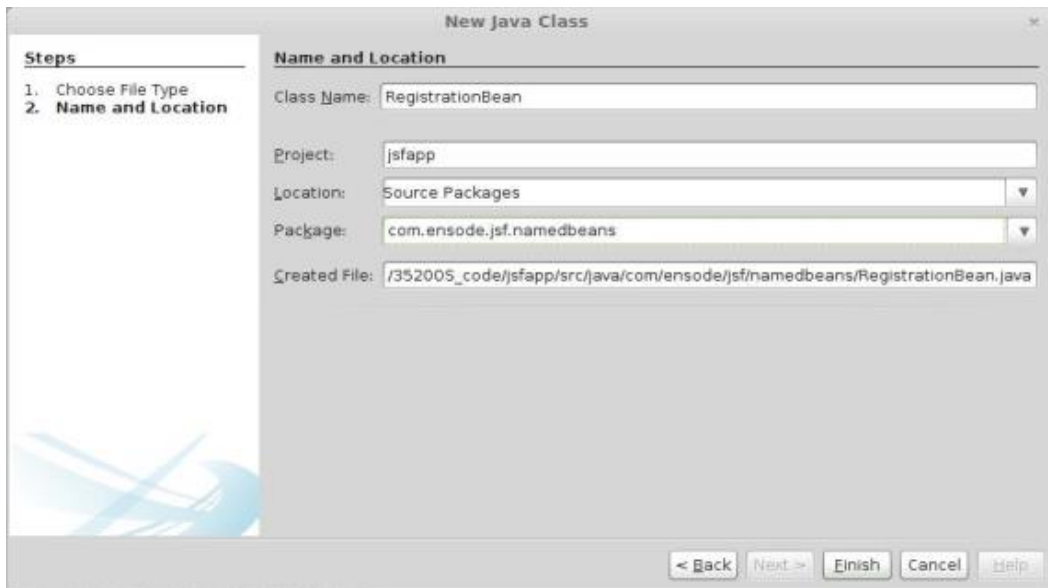
2.1.3--Création du CDI bean(Beau Nommé)

Les Beans només sont utilisés pour contenir les données entrées par les utilisateurs.

Pour créer un CDI bean nommé, considéré comme une classe standard de java , allez dans le répertoire **Source Packages** , **New | Java Class ...**



Dans l'assistant, on a besoin de spécifier le nom du bean nommé, le package comme suit :



Le code généré est :

```
package com.ensode.jsf.namedbeans;
```

```
public class RegistrationBean {}
```

On modifie notre backing bean pour ajouter les propriétés qui vont contenir les valeurs saisies par les utilisateurs.

```
import javax.enterprise.context.RequestScoped;
import javax.inject.Named;

@Named
@RequestScoped
public class RegistrationBean {

    private String salutation;
    private String firstName;
    private String lastName;
    private Integer age;
    private String email;

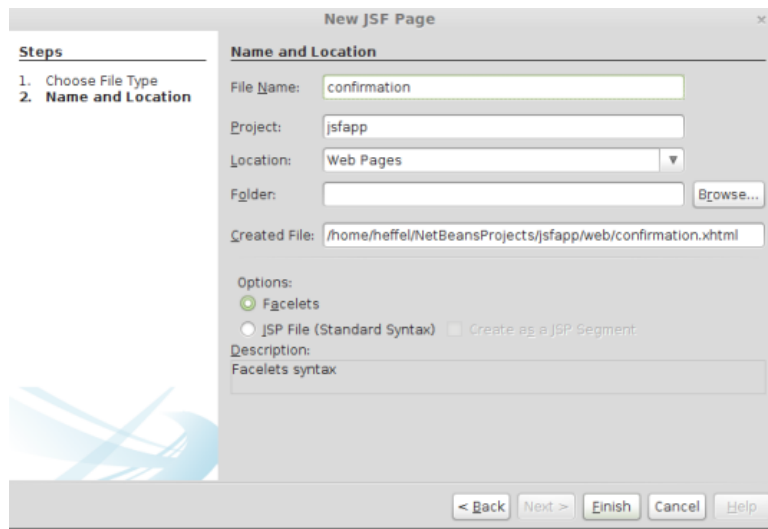
    //getters and setters omitted for brevity

}
```

2.1.4-Implementer la page de confirmation.

Une fois que l'utilisateur remplit les données dans la page d'entrée et soumet le formulaire, on montre une page de confirmation qui affiche les valeurs entrées par l'utilisateur. On utilise les expressions de liaison pour tous les champs d'entrée de la page, les champs correspondants au backing bean sont remplis avec les données saisies par l'utilisateur. La page de confirmation va afficher les données du backing bean par une série de balises JSF **<h:outputText>**.

On peut créer la page de confirmation via l'assistant **New JSF File** (File| New File en sélectionnant la catégorie JavaServer Faces et le type du fichier comme étant JSF Page)



Pour que la navigation statique fonctionne correctement, on doit s'assurer que le nom du nouveau fichier correspond à la valeur de l'attribut **action** du bouton de commande de la page d'entrée (**confirmation.xhtml**).

Après modification pour répondre aux exigences, la page générée devrait ressembler à:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:h="http://xmlns.jcp.org/jsf/html">
<h:head>
<title>Confirmation Page</title>
<h:outputStylesheet library="css" name="styles.css"/>
</h:head>
<h:body>
<h2>Confirmation Page</h2>
<h:panelGrid columns="2"
columnClasses="rightalign-bold,leftalign">
<h:outputText value="Salutation:"/>
${registrationBean.salutation}
<h:outputText value="First Name:"/>
${registrationBean.firstName}
<h:outputText value="Last Name:"/>
${registrationBean.lastName}
<h:outputText value="Age:"/>
${registrationBean.age}
<h:outputText value="Email Address:"/>
${registrationBean.email}
14
</h:panelGrid>
</h:body>
</html>
```

2.1.5-Execution & JSF Validation

Pour exécuter notre application JSF, faites un clic-droit sur le projet et cliquez sur exécuter.

Le serveur GlassFish démarre automatiquement, si ce n'est déjà fait, et le navigateur par défaut est ouvert et sera dirigé vers l'URL de notre page.

Registration Page

Salutation: Mr. ▼

First Name: Mohammed

Last Name: Bousba

Age: 31

Email Address: mbousba@gmail.com

Register

Quand on clique sur le bouton enregistrer, notre backing bean **RegistrationBean** est chargé avec les valeurs entrées dans la page.

Confirmation Page

Salutation: Mr.

First Name: Mohammed

Last Name: Bousba

Age: 31

Email Address: mbousba@gmail.com

→ Validation

On a déjà souligné comment l'attribut **required**, d'un champ de saisie, permet de rendre la saisie d'une valeur obligatoire.

Si un utilisateur tente de soumettre un formulaire avec un ou plusieurs champs **required** manquants, un message d'erreur est automatiquement généré.

Registration Page

Salutation: Mrs. ▼

First Name: First Name: Validation Error: Value is required.

Last Name:

Age:

Email Address:

Le champ âge est lié à une propriété de type Integer dans notre backing bean. Si un utilisateur entre une valeur qui n'est pas un entier, une erreur de validation est générée automatiquement.

Le champ de saisie adresse email est lié à une propriété de type String dans notre backingbean. Ceci ne permet pas de s'assurer que l'utilisateur a saisi une adresse email valide. C'est pour cela, on a besoin d'écrire notre propre validateur JSF. Les validateurs JSF personnalisés doivent implémenter l'interface **javax.faces.validator.Validator**. Cette interface contient une méthode unique nommée **validate()**, qui prend trois paramètres, une instance de **javax.faces.context.FacesContext**, une instance de **javax.faces.component.UIComponent** contenant le composant JSF à valider, et une instance de **java.lang.Object** contenant la valeur entrée par l'utilisateur pour le composant. L'exemple suivant illustre un validateur personnalisé:

Créez une classe **EmailValidator** dans le package **com.ensode.jsf.validators**

```
package com.ensode.jsf.validators;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import javax.faces.application.FacesMessage;
import javax.faces.component.UIComponent;
import javax.faces.component.html.HtmlInputText;
import javax.faces.context.FacesContext;

import javax.faces.validator.Validator;
import javax.faces.validator.ValidatorException;
import javax.faces.validator.FacesValidator;
@FacesValidator(value="emailValidator")
public class EmailValidator implements Validator {
    public void validate(FacesContext facesContext,
        UIComponent uIComponent, Object value) throws
        ValidatorException {
        Pattern pattern = Pattern.compile("\\w+@\\w+\\.\\w+");
        Matcher matcher = pattern.matcher((CharSequence) value);
        HtmlInputText htmlInputText = (HtmlInputText)uIComponent;
        String label;
        if (htmlInputText.getLabel() == null ||
            htmlInputText.getLabel().trim().equals("")) {
            label = htmlInputText.getId();
        } else {
            label = htmlInputText.getLabel();
        }
        if (!matcher.matches()) {
            FacesMessage facesMessage =
                new FacesMessage(label + ": not a valid email address");

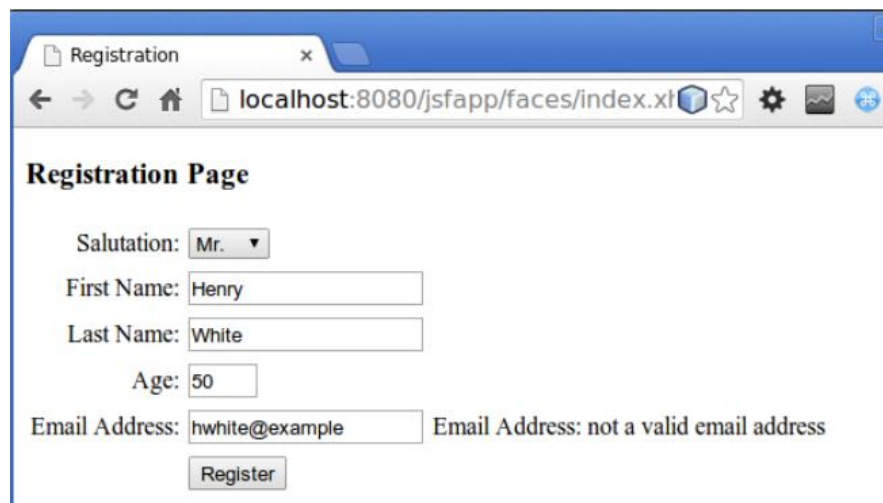
            throw new ValidatorException(facesMessage);
        }
    }
}
```

La méthode **validate()** effectue une correspondance entre l'expression régulière et la valeur du composant à valider. Si la valeur correspond à l'expression, la validation a réussi, sinon, la validation a échoué et une instance de l'exception **javax.faces.validator.ValidatorException** est levée. Notre validateur doit être annoté avec l'annotation **@FacesValidator**. La valeur de son attribut **value** est l'**id** qui sera utilisé pour faire référence à notre validateur dans nos pages JSF

Notre validateur est prêt à être utilisé. Modifiez, dans le fichier **index.xhtml**, la balise du champ email pour utiliser notre validateur personnalisé.

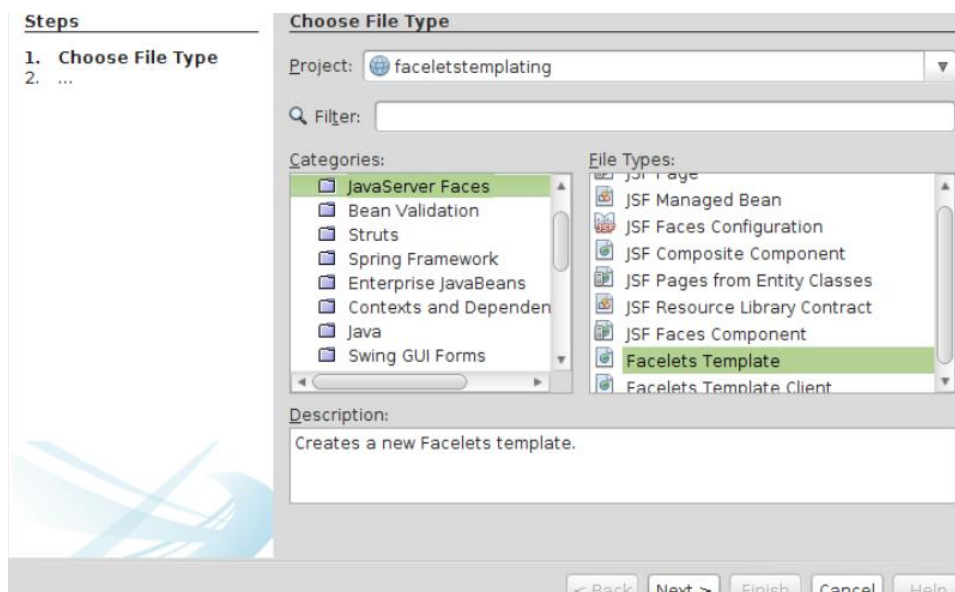
```
<h:inputText id="email" label="Email Address"
required="true" value="#{registrationBean.email}">
<f:validator validatorId="emailValidator"/>
</h:inputText>
```

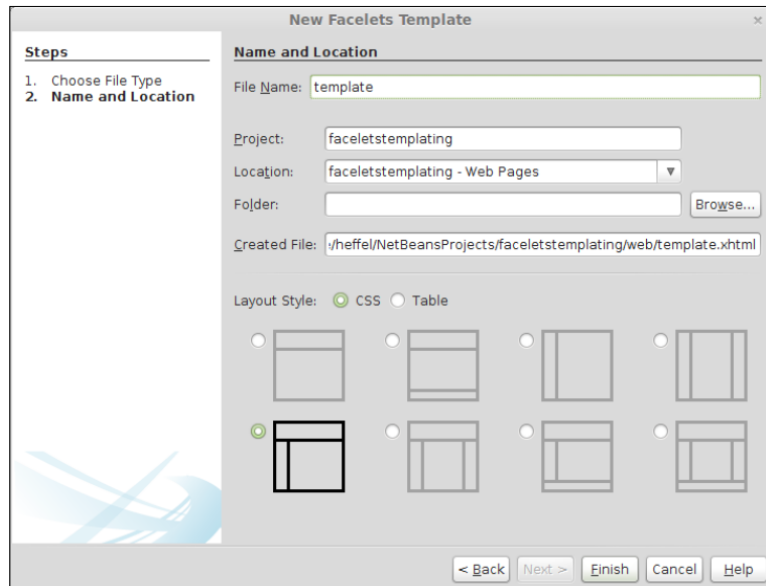
Mettre la balise **<f:validator>** pour le champ de saisie à valider avec le validateur personnalisé. La valeur de l'attribut **validatorId** de **<f:validator>** doit correspondre à la valeur de l'attribut **value** dans l'annotation **@FacesValidator** de notre validateur.



2. 2- Les Templates d'une Facelet.

NetBeans fournit un très bon support pour les templates des Facelets.





Après avoir cliqué sur Terminer, NetBeans génère automatiquement notre modèle, avec les fichiers CSS nécessaires.

Le modèle généré ressemble à ceci:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
xmlns:h="http://xmlns.jcp.org/jsf/html">
<h:head>
<meta http-equiv="Content-Type" content="text/html;
charset=UTF-8" />
<h:outputStylesheet name="css/default.css"/>
<h:outputStylesheet name="css/cssLayout.css"/>
<title>Facelets Template</title>
</h:head>
<h:body>
<div id="top" class="top">
<ui:insert name="top">Top</ui:insert>
</div>
<div>
<div id="left">
<ui:insert name="left">Left</ui:insert>
</div>
20
<div id="content" class="left_content">
<ui:insert name="content">Content</ui:insert>
</div>
</div>
</h:body>
</html>
```

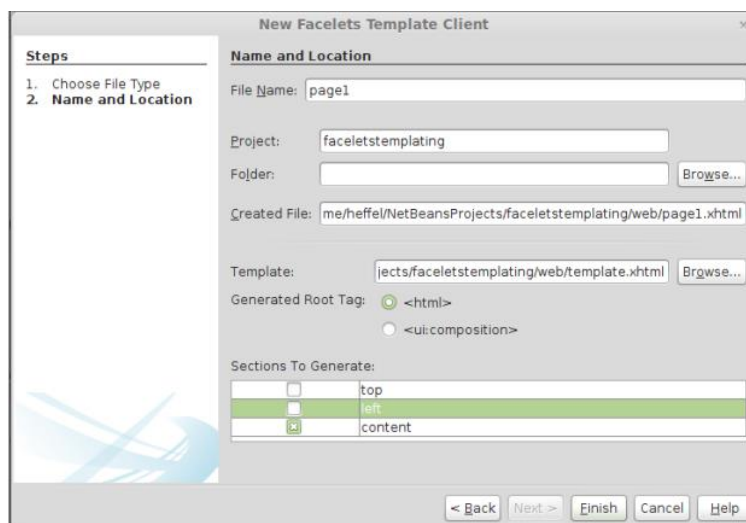
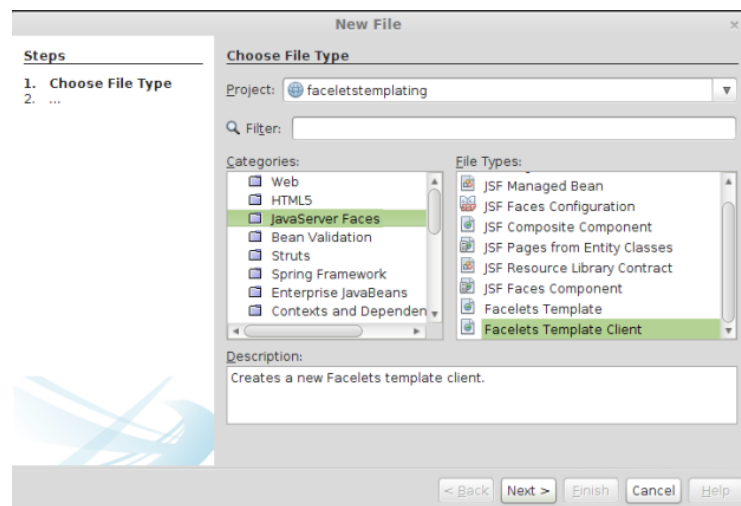
> Ajout d'un modèle Facelets à notre projet

On peut ajouter un modèle Facelets à notre projet en cliquant sur Fichier | Nouveau fichier, puis en sélectionnant la catégorie **JavaServer Faces** et le type de fichier **Facelets Template Client**.

On note que le template utilise l'espace de noms suivants: `xmlns: ui = "http://xmlns.jcp.org /JSF/Facelets"`. Cet espace de noms nous permet d'utiliser la balise `<ui:insert>`, le contenu de cette balise sera remplacée par le contenu de la balise correspondante `<ui:define>` de la page cliente du modèle

→ Utilisation du template

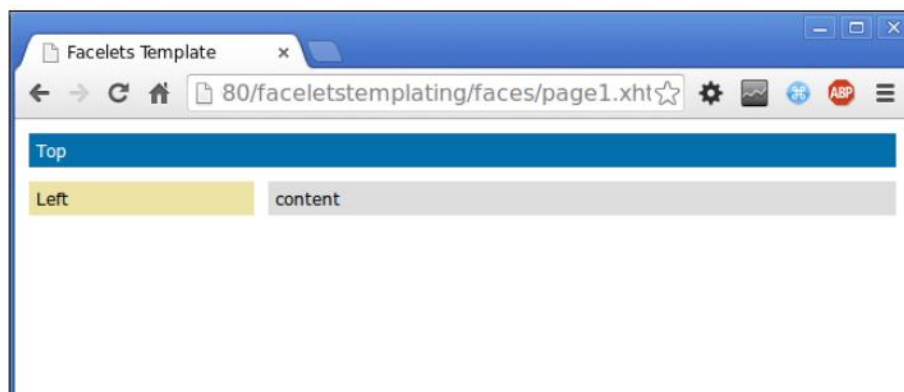
Pour utiliser notre template, on va créer une Facelet cliente du template, ainsi on clique sur **Fichier | Nouveau fichier**, on sélectionne la catégorie **JavaServer Faces** et **Facelets Template Client** comme type du fichier.



Après avoir cliqué sur Terminer, notre template client est créé.

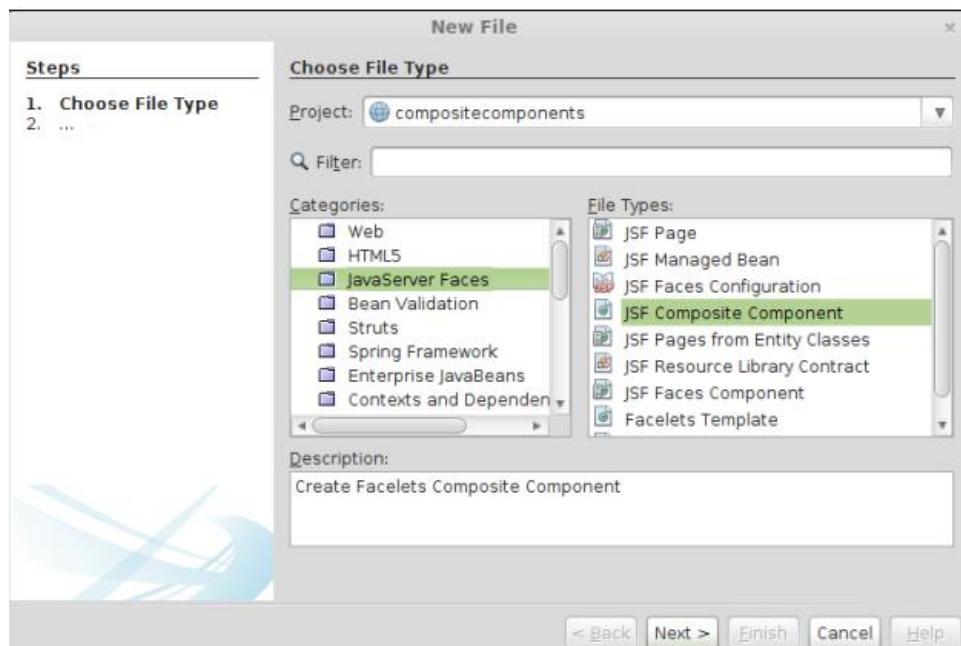
```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:ui="http://xmlns.jcp.org/jsf/facelets">
<body>
<ui:composition template="./template.xhtml">
<ui:define name="content">
content
</ui:define>
</ui:composition>
</body>
</html>
```

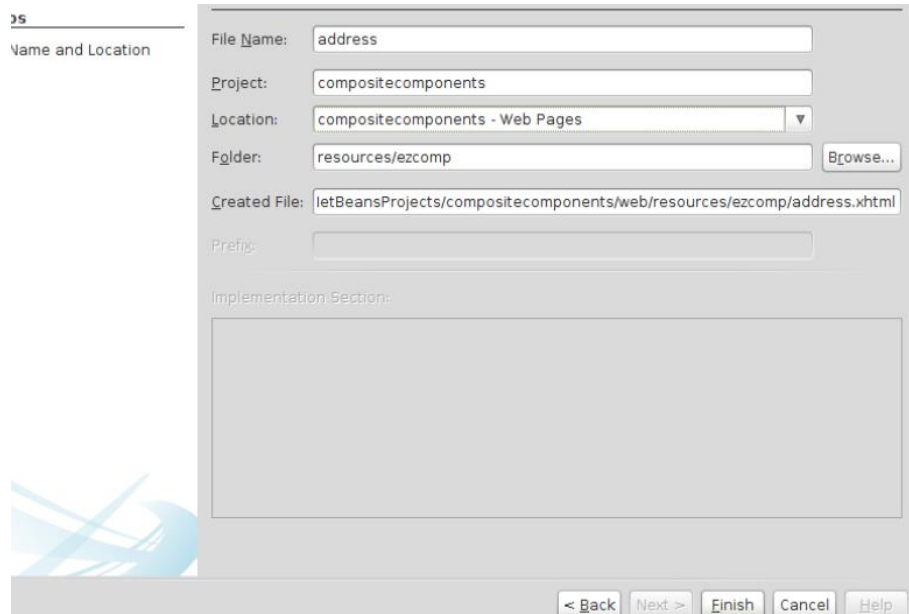
La valeur de l'attribut **name** dans **<ui:define>** doit correspondre à la balise **<ui:insert>** du template.
Après le déploiement de notre application, en dirigeant le navigateur sur l'URL du template client.



2.3-Compsants composites

Nous pouvons générer un composant composite en cliquant sur **Fichier | Nouveau**, dans catégorie **JavaServer Faces** et le type de fichier est **JSF Composite Component**.





Lorsqu'on clique sur Terminer, NetBeans génère un composant composite vide utilisé comme une base pour créer le notre.

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:cc="http://xmlns.jcp.org/jsf/composite">
<!-- INTERFACE -->
<cc:interface>
</cc:interface>
<!-- IMPLEMENTATION -->
<cc:implementation>
</cc:implementation>
</html>
```

Après avoir "rempli les blancs", notre composant composite ressemble maintenant à ceci:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:cc="http://xmlns.jcp.org/jsf/composite"
xmlns:h="http://xmlns.jcp.org/jsf/html"
xmlns:f="http://xmlns.jcp.org/jsf/core">
<!-- INTERFACE -->
<cc:interface>
<cc:attribute name="addrType"/>
<cc:attribute name="namedBean" required="true"/>
</cc:interface>
<!-- IMPLEMENTATION -->
<cc:implementation>
<h:panelGrid columns="2">
<f:facet name="header">
```

```

<h:outputText value="#{cc.attrs.addrType}
Address"/>
</f:facet>
<h:outputLabel for="line1" value="Line 1"/>
<h:inputText id="line1"
value="#{cc.attrs.namedBean.line1}"/>
<h:outputLabel for="line2" value="Line 2"/>
<h:inputText id="line2"
value="#{cc.attrs.namedBean.line2}"/>
<h:outputLabel for="city" value="City"/>
<h:inputText id="city"
value="#{cc.attrs.namedBean.city}"/>
<h:outputLabel for="state" value="state"/>
<h:inputText id="state"
value="#{cc.attrs.namedBean.state}" size="2" maxlength="2"/>
<h:outputLabel for="zip" value="Zip"/>
<h:inputText id="zip" value="#{cc.attrs.namedBean.zip}"
size="5" maxlength="5"/>
</h:panelGrid>
</cc:implementation>
</html/>

```

Modifiez le contenu de la page **index.xhtml** en celui de :

```

<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:h="http://xmlns.jcp.org/jsf/html"
xmlns:ezcomp="http://xmlns.jcp.org/jsf/composite/ezcomp">
<h:head>
<title>Address Entry</title>
</h:head>
<h:body>
<h:form>
<h:panelGrid columns="1">
<ezcomp:address namedBean="#{addressBean}"
addrType="Home"/>
<h:commandButton value="Submit" action="confirmation"
style="display: block; margin: 0 auto;"/>
</h:panelGrid>
</h:form>
</h:body>
</html>

```

Par convention, l'espace de noms pour nos composants personnalisés seront toujours **xmlns:ezcomp="http://xmlns.jcp.org/jsf/composite/ezcomp"**.

Dans notre application, on a un simple backing bean **AddressBean**. Il s'agit d'un simple backing bean avec quelques propriétés et des getters et des setters correspondants. Ce backing bean est utilisé dans l'attribut **namedBean** de notre composant.

Dans le projet **compositecomponents**, cliquez sur le bouton droit **Nouveau, JSF NamedBean**. Introduire le nom **AddressBean**, un nom **fsr** pour le package et copiez le code suivant :

```

import javax.enterprise.context.RequestScoped;
import javax.inject.Named;
@Named
@RequestScoped
public class AddressBean {

    /** Creates a new instance of AddressBean */
    public AddressBean() {
    }
    private String line1;
    private String line2;
    private String city;
    private String state;
    private String zip;
    public String getCity() {return city;
    }
    public void setCity(String city) {this.city = city;
    }
    public String getLine1() {return line1;
    }
    public void setLine1(String line1) {this.line1 = line1;
    }
    public String getLine2() {return line2;
    }
    public void setLine2(String line2) {this.line2 = line2;
    }
    public String getState() {return state;
    }
    public void setState(String state) {this.state = state;
    }
    public String getZip() { return zip;
    }
    public void setZip(String zip) { this.zip = zip;
    }
}

```

Page de **confirmation.xhtml**

```

<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:h="http://xmlns.jcp.org/jsf/html">
<h:head>
<title>Facelet Title</title>
</h:head>
<h:body>
<h2>Address</h2>
#{addressBean.line1}<br/>
#{addressBean.line2}<br/>
#{addressBean.city}<br/>
#{addressBean.state}<br/>
#{addressBean.zip}<br/>
</h:body>
</html>

```


Address Entry

localhost:8080/compositecomponents/

Home Address

Line 1

Line 2

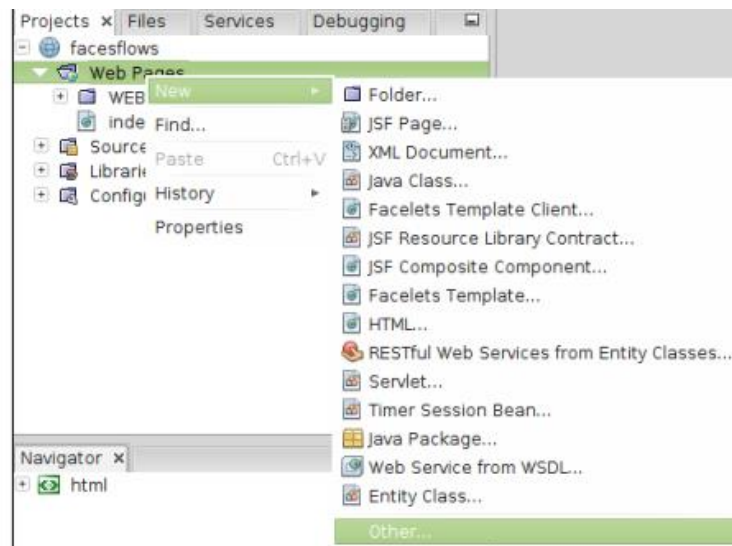
City

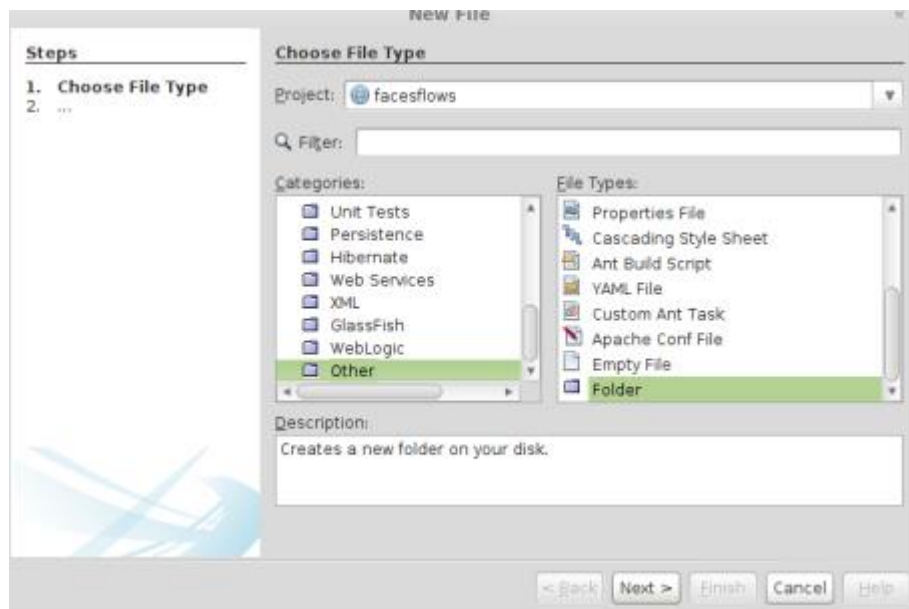
State

Zip

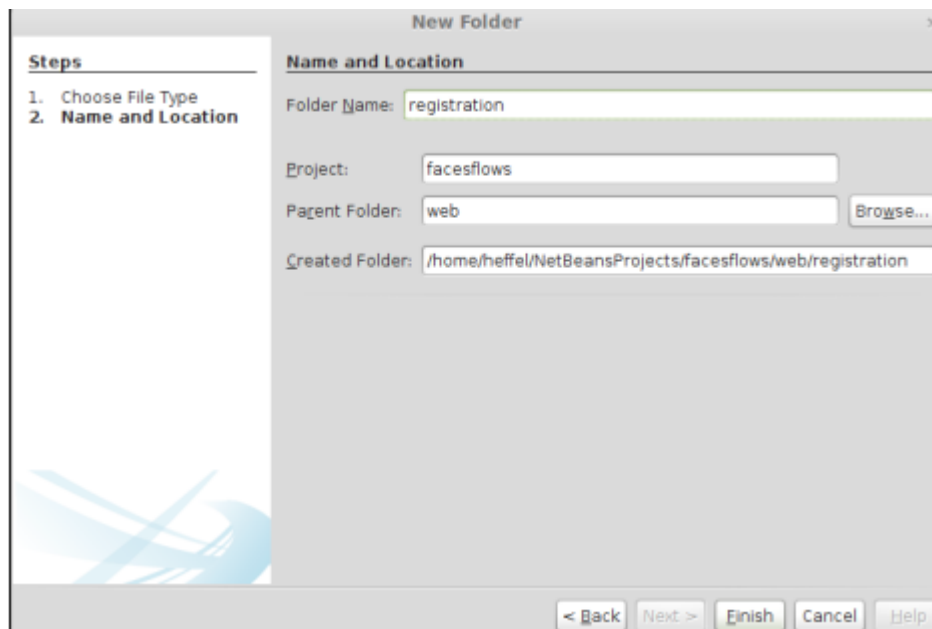
2.4-Les FaceFlows.

Dans la fenêtre project de NetBeans, on peut créer ce dossier en cliquant avec le bouton droit sur le dossier **Web pages** et aller dans **new|Other**.

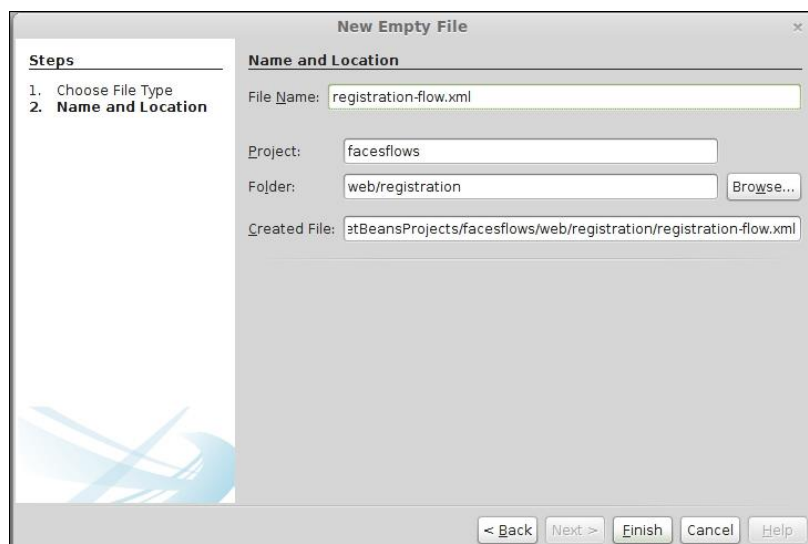




Le nom du dossier (flow) dans notre cas est : **registration**



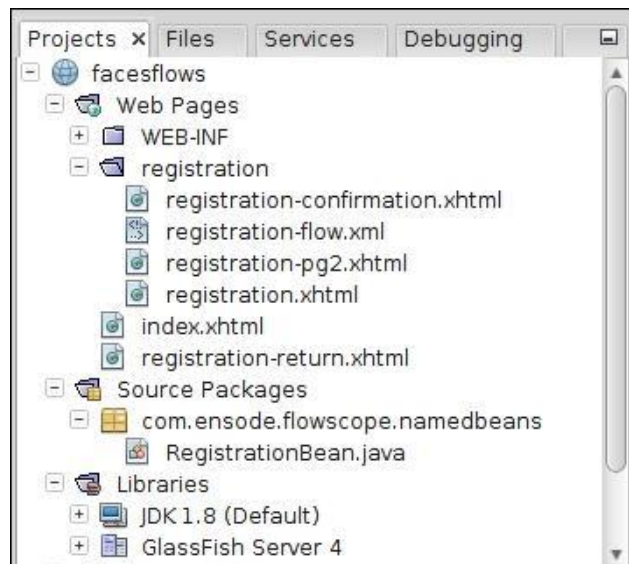
On introduit le nom **registration-flow.xml** et on vérifie qu'on est bien dans le dossier flow.



Les données du flow doivent être tenues dans un ou plusieurs backing bean de portée flow comme le montre l'exemple suivant :

```
package ma.fsr.flowscope.namedbeans;
import java.io.Serializable;
import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import javax.faces.flow.FlowScoped;
import javax.inject.Named;
@Named
@FlowScoped("registration")
public class RegistrationBean implements Serializable {
    private String salutation;
    private String firstName;
    private String lastName;
    private Integer age;
    private String email;
    private String line1;
    private String line2;
    private String city;
    private String state;
    private String zip;
    @PostConstruct
    public void init() {
        System.out.println(this.getClass().getCanonicalName() + "
        initialized.");
    }
    @PreDestroy
    public void destroy() {
        System.out.println(this.getClass().getCanonicalName() + "
        destroyed.");
    }
}
```

Les fichiers nécessaires au projet sont organisés comme suit



Lorsque l'utilisateur clique sur le lien, la première page du flow est affichée

Registration Page

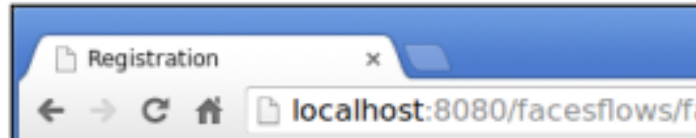
Salutation: ▼

First Name:

Last Name:

Age:

Email Address:



Dans la première page du flow, l'action de la balise `commandButton` nous dirige vers `registration-pg2` :

```
<h:commandButton id="continue" value="Continue" action="registrationpg2" />.
```

Dans la seconde page (`registration-pg2`) les `commandButton` nous permettent soit de nous diriger vers la première page (`registration`) soit de nous diriger vers `registrationconfirmation` :

```
<h:commandButton id="back" value="Go Back" action="registration" />
<h:commandButton id="continue" value="Continue"
action="registration-confirmation" />
```

En cliquant sur le bouton Continue, on arrive à la dernière page du flow :

```
<h:commandButton value="Continue" action="registration-return"/>
```

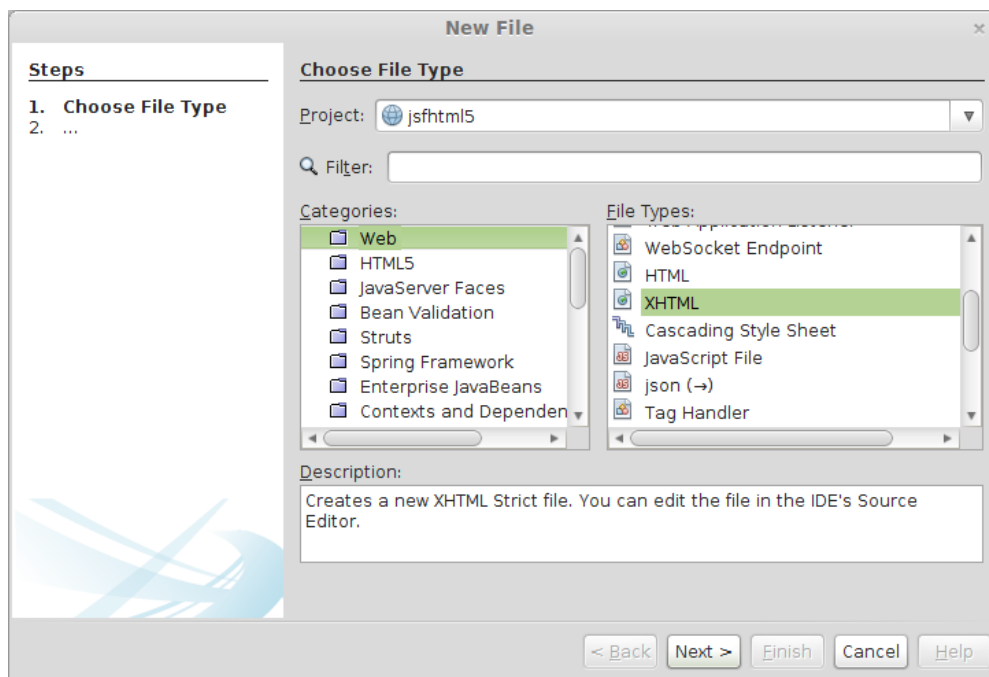
En cliquant sur le bouton Continue, on sort du flow et le backing bean est détruit comme c'est confirmé par le fichier log de GlassFish :

```
Info: ma.fsr.flowscope.namedbeans.RegistrationBean destroyed
```

2.5-Support HTML5.

JSF 2.2 ajoute de nouvelles améliorations pour supporter les fonctionnalités de HTML5. Les deux caractéristiques les plus importantes sont : le balisage amical de HTML5 et les attributs traversés

Pour utiliser le balisage amical de HTML5 avec NetBeans, on doit créer un projet d'application Web à partir de la section **Java Web** et on sélectionne **JavaServer Faces** comme framework habituel. Lors de l'ajout des pages à notre application, on doit sélectionner **XHTML** comme type du fichier dans la catégorie **Web**.



Après avoir ajouté des balises dans notre page **registration.xhtml**, elle ressemble à ceci :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
38
xmlns:jsf="http://xmlns.jcp.org/jsf"
xmlns:f="http://xmlns.jcp.org/jsf/core">
<head>
<title>Registration</title>
<meta name="viewport" content="width=device-width, initial-scale=1.0"/>
<link rel="stylesheet" type="text/css" href="css/styles.css"/>
</head>
<body>
<h3>Registration Page</h3>
<form jsf:id="mainForm" jsf:prependId="false">
<table border="0" cellspacing="0" cellpadding="0">
<tbody>
<tr>
<td class="rightalign">Salutation:</td>
<td class="leftalign">
<select name="salutation" jsf:id="salutation"
jsf:value="#{registrationBean.salutation}" size="1">
<f:selectItem itemValue="" itemLabel=""/>
<f:selectItem itemValue="MR" itemLabel="Mr."/>
<f:selectItem itemValue="MRS" itemLabel="Mrs."/>
<f:selectItem itemLabel="Miss" itemValue="MISS"/>
<f:selectItem itemLabel="Ms" itemValue="MS"/>
<f:selectItem itemLabel="Dr." itemValue="DR"/>
</select> </td> </tr>
<tr>
<td class="rightalign"> First Name: </td>
<td class="leftalign">
<input type="text" jsf:id="firstName" jsf:value =
"#{registrationBean.firstName}"/> </td> </tr>
<tr>
<td class="rightalign"> Last Name: </td>
<td class="leftalign">
<input type="text" jsf:id="lastName"
```

```

jsf:value="#{registrationBean.lastName}"/>
</td> </tr> <tr>
<td class="rightalign"> Age: </td>
<td class="leftalign">
<input type="number" jsf:id="age" jsf:value="#{registrationBean.age}"/>
</td> </tr>
<tr>
<td class="rightalign"> Email Address: </td>
<td class="leftalign">
<input type="text" jsf:id="email" jsf:value="#{registrationBean.email}"
placeholder="username@example.com"/> </td>
</tr>
<tr>
<td> </td>
<td><input type="submit" value="Submit" jsf:action="confirmation" />
</td> </tr>
</tbody>
</table>
</form>
</body>
</html>

```

Pour que JSF puisse interpréter les balises HTML, il faut ajouter au moins, à chaque balise un attribut spécifique JSF.

Ces balises sont définies dans l'espace de noms **xmlns:**

JSF="http://xmlns.jcp.org/jsf", qu'on doit ajouter à la page.

On ajoute la page de **confirmation.xhtml** suivante :

```





<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:jsf="http://xmlns.jcp.org/jsf">
<head>
<title>Confirmation</title>
<meta name="viewport" content="width=device-width, initialscale=1.0"/>
</head>
<body jsf:id="body">
<h3>Confirmation</h3>
Salutation: ${registrationBean.salutation}<br/>
First Name: ${registrationBean.firstName}<br/>
Last Name: ${registrationBean.lastName}<br/>
Age: ${registrationBean.age}<br/>
Email Address: ${registrationBean.email}
</body>
</html>

```

Et un backing bean **RegistrationBean**.

Lorsqu'on exécute notre code, on peut voir la page rendue dans le navigateur.

Registration x

    localhost:8080/jsfhtml5/faces/index.xhtml

Registration Page

Salutation:

Mr.

Mrs.

Miss

Ms

Dr.

First Name:

Last Name:

Age:

Email Address:

username@example.com

Submit