Une introduction à JSF

1 Introduction

Le framework JSF (Java Server faces) a pour ambition de faciliter l'écriture d'applications WEB en offrant d'une part, une architecture MVC et d'autre part, une approche composant qui rappelle les interfaces graphiques natives. L'objectif est de remplacer l'approche
je traite la requête HTTP et je renvoie une page HTML
par
j'envoie des composants (boutons, champs de texte, etc..) et je réagis aux actions de l'utilisateur
JSF est composée de trois parties :
_ une servlet générique qui va traiter les dialogues,
_ une librairie de balises destinée à mettre en place les composants graphiques,
_ une librairie de balises destinée à gérer ces composants et le dialogue avec le serveur.

Quelques liens :
Site officiel de JSF http://www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html
API JSF 2.0 https://javaserverfaces.java.net/nonav/docs/2.0/javadocs/
Tags JSF 2.0 http://javaserverfaces.java.net/nonav/docs/2.0/vdldocs/facelets/
myFaces : l'implantation libre proposée par Apache http://myfaces.apache.org/
Des cours sur JSF :
_ Un cours complet en français [1] (et pas seulement sur JSF) (1),
_ 176 trés bons transparents en français sur JSF (2),
_ Une liste de cours sur le sujet (3),
_ Un site associé à un livre (4).
_ Des cours en ligne (5) (pas seulement sur JSF).

1. http://www.jmdoudoux.fr/java/dej/chap-jsf.htm#jsf
2. http://mbaron.developpez.com/javaee/jsf/
3. http://www.jsftutorials.net/
4. http://corejsf.com/
5. http://www.coreservlets.com/JSF-Tutorial/
6. http://tomee.apache.org/apache-tomee.html
7. ress-jsf
8. http://myfaces.apache.org

_ Créez une application WEB « myapp » basée sur le serveur glassfish.
_ Ajoutez à votre projet le framework JavaServer faces
_ le fichier WEB-INF/web.xml sera créé:

_ Créez la page JSP WebContent/index.jsp listée ci-dessous

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<c:redirect url="/hello.xhtml"/>
```

_ Créez la page WebContent/hello.xhtml listée ci-dessous

```
<?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:f="http://java.sun.com/jsf/core"
xmlns:h="http://java.sun.com/jsf/html">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Hello page</title>
</head>
<body>
<f:view>
<h1>
<h:outputText value="Hello, world" />
</h1>
</f:view>
</body>
</html>
```

Remarque : Une page JSF (aussi appelée Facelet) est un document XML basé sur le langage XHTML. Les espaces de noms permettent d'identifier les balises qui seront traitées avant l'envoie du résultat à l'utilisateur (dans notre exemple <h:outputText .../>).
Remarque : Vous pouvez accéder à cette page en utilisant les adresses : /faces/hello.xhtml , /hello.xhtml et /hello.jsf
Remarque : Il est à noter que la servlet récupére la main à chaque requête et prépare le contexte pour que la vue puisse produire le résultat.

2.1 Feuille de style

Dans JSF les ressources statiques (fichier CSS et/ou JavaScript) sont localisées dans des répertoires particuliers.
Créez les répertoires

```
Racine de votre application
+ WebContent
| + resources
| | + css
```

et créez le fichier WebContent/resources/css/style.css

```
h1 {
border-bottom: solid 1px blue;
}
table {
border-collapse: collapse;
border: 1px solid black;
}
td, th {
border: 1px solid black;
padding: 3px;
}
```

```
#page {
width: 800px;
margin: 0 auto;
}
#header {
border: solid 1px blue;
padding: 5px;
size: large;
color: blue;
background-color: #c8bfbf;
margin-bottom: 20px;
}
#menu {
float: right;
width: 150px;
border: solid 1px gray;
padding: 5px;
color: blue;
background-color: white;
margin-left: 20px;
}
#content {
}
#footer {
border: solid 1px blue;
padding: 5px;
size: small;
color: blue;
background-color: #c8bfbf;
margin-top: 20px;
}
.alert {
color: red;
border: solid 2px red;
}
```

Nous pouvons maintenant préparez une deuxième version de hello.xhtml qui utilise cette feuille de style :

```
<?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:f="http://java.sun.com/jsf/core"
xmlns:h="http://java.sun.com/jsf/html">
<h:head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Hello page</title>
</h:head>
<h:body>
<h:outputStylesheet library="css" name="style.css" />
<f:view>
<h1>
<h:outputText value="Hello, world" />
</h1>
</f:view>
</h:body>
</html>
```

Travail à faire : Testez cette deuxième version et analysez le code source XHTML renvoyé par le serveur.
Travail à faire : En utilisant le tag <h:graphicImage/> introduisez une image dans votre page (n'oubliez pas de prévoir le répertoire WebContent/resources/theme1 et l'image).

```
<h:graphicImage library="theme1" name="minimage.jpg" />
```

Travail à faire : Faites de même avec <h:outputScript/> :

```
<h:outputScript library="js" name="hello.js" />
```

pour le script WebContent/resources/js/hello.js :

```
alert('Hello');
```

et observez le code source généré.

2.2 Les templates JSF

La technologie JSF permet de définir un (ou plusieurs) gabarit(s) de page et de le réutiliser quand cela est nécessaire.
_ Commencez par créez le çhier WebContent/template/mylayout.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:f="http://java.sun.com/jsf/core"
xmlns:h="http://java.sun.com/jsf/html"
xmlns:c="http://java.sun.com/jsp/jstl/core"
xmlns:ui="http://java.sun.com/jsf/facelets">
<h:head>
<title><ui:insert name="title">My application</ui:insert></title>
</h:head>
<h:body>
<h:outputStylesheet library="css" name="style.css" />
<div id="page">
<div id="header">
<ui:insert name="header">
<span>My application</span>
</ui:insert>
</div>
<div id="menu">
<ui:insert name="menu">
<ui:include src="/template/menu.xhtml" />
</ui:insert>
</div>
<ui:debug />
<div id="content">
<ui:insert name="content">
<p>CONTENT</p>
</ui:insert>
<h:messages />
</div>
<div id="footer">
<ui:insert name="footer">
<span>(c) 2015</span>
</ui:insert>
</div>
</div>
</h:body>
</html>
```

et WebContent/template/menu.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:f="http://java.sun.com/jsf/core"
xmlns:h="http://java.sun.com/jsf/html"
xmlns:ui="http://java.sun.com/jsf/facelets">
<ui:composition>
<div>Menu :</div>
<ul>
<li>lien 1</li>
<li>lien 2</li>
</ul>
</ui:composition>
</html>
```

Ces deux çhiers définissent un format par défaut pour les pages de notre application et prévoient une valeur par défaut pour chacune des parties.

_ Nous pouvons maintenant reformuler hello.xhtml en utilisant le template :

```
<?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:f="http://java.sun.com/jsf/core"
xmlns:h="http://java.sun.com/jsf/html"
xmlns:ui="http://java.sun.com/jsf/facelets">
<h:body>
<ui:composition template="/template/mylayout.xhtml">
<ui:define name="content">
<h1>
<h:outputText value="Hello, world" />
</h1>
</ui:define>
<ui:define name="title">Hello v2</ui:define>
</ui:composition>
</h:body>
</html>
```

Travail à faire : Prévoir un titre particulier pour votre page hello.xhtml et testez le résultat.

Travail à faire : Introduisez des paragraphes avant et aprés le tag <ui:composition/> et vérifiez qu'ils sont bien ignorés.

Travail à faire : Remplacez <ui:composition/> par <ui:decorate/> et testez le résultat (avec des paragraphes avant et aprés). Revenez ensuite à <ui:composition/> .

Travail à faire : Testez le Ctrl + Shift + D pour avoir accés la page de debug (activée par <ui:debug/>).

2.3 Les fichiers de messages

Nous pouvons facilement (comme dans la plupart des frameworks) délocaliser les messages afin de simplifier le code et de faciliter l'internationalisation du logiciel.

_ Pour ce faire, nous allons créer le package monapp.resources et placer à l'intérieur les fichiers

messages.properties

```
helloWorld=Hello, world!
bye=Bye
```

et le fichier

messages_fr.properties :

helloWorld=Bonjour à tous et toutes !
bye=Au revoir

Pour utiliser ces ressources dans vos pages JSF, il faut déclarer ces messages sous la forme d'une variable et les utiliser :

7

```
<?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:f="http://java.sun.com/jsf/core"
xmlns:h="http://java.sun.com/jsf/html"
xmlns:ui="http://java.sun.com/jsf/facelets">
<h:body>
<ui:composition template="/template/mylayout.xhtml">
<ui:define name="content">
<f:loadBundle basename="monapp.resources.messages" var="msg" />
<h1>
<h:outputText value="#{msg.helloWorld}" />
</h1>
</ui:define>
<ui:define name="title">Hello v3</ui:define>
</ui:composition>
</h:body>
</html>
```

Remarque : nous introduisons à cette étape le langage d'expressions utilisé dans JSF (#f...g). Il n'a pas la même syntaxe que celui des pages JSP et nous donnerons des éléments supplémentaires au fur et à messure du TP.

_ Aprés avoir testé cette version, creer le fichier WEB-INF/faces-config.xml :

```
<?xml version="1.0" encoding="UTF-8"?>
<faces-config 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-facesconfig_2_2.xsd"
version="2.2">
<!-- Configuration de l'application -->
<application>
<resource-bundle>
<base-name>monapp.resources.messages</base-name>
<var>messages</var>
</resource-bundle>
<message-bundle>
monapp.resources.messages
</message-bundle>
<locale-config>
<default-locale>en</default-locale>
<supported-locale>fr</supported-locale>
</locale-config>
</application>
</faces-config>
```

Cela a pour effet de déclarer une variable globale que vous pouvez directement utiliser dans vos pages JSF :

```
<?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:f="http://java.sun.com/jsf/core"
```

```
xmlns:h="http://java.sun.com/jsf/html"
xmlns:ui="http://java.sun.com/jsf/facelets">
<h:body>
<ui:composition template="/template/mylayout.xhtml">
<ui:define name="content">
<h1>
<h:outputText value="#{messages.helloWorld}" />
</h1>
</ui:define>
<ui:define name="title">Hello v4</ui:define>
</ui:composition>
</h:body>
</html>
```

Vous trouverez plus d'information sur l'internationalisation dans ce guide http://www.laliluna.de/javaserver-faces-message-resource-bundle-tutorial.html

3 Navigation et contrôleur

3.1 La navigation dans JSF

_ Créez la nouvelle page /bye.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:f="http://java.sun.com/jsf/core"
xmlns:h="http://java.sun.com/jsf/html"
xmlns:ui="http://java.sun.com/jsf/facelets">
<ui:composition template="/template/mylayout.xhtml">
<ui:define name="content">
<f:view>
<h1>
<h:outputText value="#{messages.bye}" />
</h1>
<h:form>
<p>
<h:commandLink action="hello">Hello</h:commandLink>
<h:outputText> | </h:outputText>
<h:commandLink action="bye">Bye</h:commandLink>
</p>
</h:form>
<p>
<h:link outcome="hello">Hello</h:link>
</p>
</f:view>
</ui:define>
</ui:composition>
</html>
```

Remarque : Dans les pages JSF les liens sont des éléments d'un formulaire. Ils impliquent donc une soumission. Ces opérations sont réalisées automatiquement par un code JavaScript. Analysez le code source XHTML généré.

Remarque : A ce stade, vous devez être capable de naviguer entre /hello.xhtml et /bye.html . Vous pouvez d'ailleurs mettre à jour le fichier qui code le menu (WebContent/template/menu.xhtml).

Remarque : Ces liens entre pages JSF sont appelés Navigation implicite dans la terminologie JSF. Comme nous sommes dans le cadre d'un formulaire (en méthode POST),

l'adresse de la page ne change pas. Vous pouvez forcer la redirection en utilisant la forme suivante :

```
<p>
<h:commandLink action="hello?faces-redirect=true"
value="Redirection to Hello" />
</p>
```

_ Nous allons introduire un nouveau lien dans la page /bye.html :

```
<p>
<h:commandLink action="home" value="New path to Hello" />
</p>
```

L'utilisation de ce lien entraîne l'apparition d'un message (grâce au tag <h:messages/> prévu dans le gabarit). Il n'existe pas en effet de page /home.xhtml . Pour régler ce cas, nous allons introduire, dans le fichier faces-config.xml une régle de navigation

```
<navigation-rule>
<display-name>Home</display-name>
<from-view-id>/bye.xhtml</from-view-id>
<navigation-case>
<from-outcome>home</from-outcome>
<to-view-id>/hello.xhtml</to-view-id>
</navigation-case>
</navigation-rule>
```

Le lien est de nouveau fonctionnel.

3.2 Un contrôleur pour la navigation

_ Nous pouvons confier la navigation à un objet Java côté serveur. Pour ce faire nous allons créer un managedbean : c'est une objet dont l'instanciation est gérée par JSF :

```
package monapp;
import javax.faces.bean.ManagedBean;
@ManagedBean
public class Navigation {
public String hello() {
return "hello";
}
}
```

L'action hello va nous renvoyer le nom de page JSF (ou null si aucun déplacement n'est souhaité). Le nouveau lien a la forme suivante :

```
<p>
<h:commandLink action="#{navigation.hello}"
value="Controller to Hello" />
</p>
```

Nous pouvons aussi remplacer le lien par un bouton :

```
<p>
<h:commandButton action="#{navigation.hello}"
value="Controller to Hello" />
</p>
```

_ Bien entendu, le contrôleur peut agir sur des données présentées dans la page JSF. Imaginons un compteur de portée session :

```
package monapp;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;
@ManagedBean
@SessionScoped
public class Counter {
Integer value = 1000;
public String inc() {
```

```
value++;
return null; // ne pas se déplacer
}
public Integer getValue() {
return value;
}
}
```

Nous pouvons visualiser et incrémenter ce compteur avec :

```
<p>
<h:commandButton action="#{counter.inc}" value="#{counter.value}" />
</p>
```

Travail à faire : vérifiez dans plusieurs navigateurs qu'il existe bien un compteur par utilisateur.

3.3 Portée des Managed Beans

Nous déclarions les beans par l'annotation @ManagedBean mais nous pouvons l'enrichir en fixant le nom :

```
@ManagedBean(name="myBean")
public class BeanImplementation { ... }
```

ou demander une création à la demande :

```
@ManagedBean(name="myBean", eager=false)
public class BeanImplementation { ... }
```

Il existe quatre portées pour les managed beans :

_ @ApplicationScoped : l'instance est de portée application
_ @SessionScoped : l'instance est associée à la session
_ @ViewScoped : l'instance est associée à la vue
_ @RequestScoped : l'instance est associée à la requ^ete

Travail à faire : Créez le bean ci-dessous :

```
package monapp;
import java.io.Serializable;
import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import javax.faces.bean.ApplicationScoped;
import javax.faces.bean.ManagedBean;
@ManagedBean
@ApplicationScoped
public class ApplicationCounter implements Serializable {
private static final long serialVersionUID = 79831409760756496 22L;
int value = 0;
public Integer getCounter() {
return ++value;
}
@PostConstruct
void init() {
System.err.println("Create " + this);
}
@PreDestroy
void close() {
System.err.println("Close " + this);
}
}
```

et vérifiez sa création et son évolution avec la page /counters.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:f="http://java.sun.com/jsf/core"
xmlns:h="http://java.sun.com/jsf/html"
xmlns:ui="http://java.sun.com/jsf/facelets">
<h:body>
<ui:composition template="/template/mylayout.xhtml">
<ui:define name="content">
<h1>Counters</h1>
<p>
App counter =
<h:outputText value="#{applicationCounter.counter}" />
</p>
</ui:define>
<ui:define name="title">Counters</ui:define>
</ui:composition>
</h:body>
</html>
```

Travail à faire : Créez un autre bean similaire mais de portée vue et testez son fonctionnement en enrichissant la page /counters.xhtml .

```
package monapp;
import java.io.Serializable;
import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.ViewScoped;
@ManagedBean()
@ViewScoped
public class ViewCounter implements Serializable {
... même code que le compteur de portée application
}
```

Vous remarquez que la portée est plus subtile car une nouvelle instance est créée à chaque fois. Ce type d'instance permet de faire le lien entre une premiére requête GET qui va créer le bean et une série de requêtes POST qui vont l'utiliser et le modifier (durée de vie d'un formulaire).

Pour illustrer ce fonctionnement, vous devez ajouter à la page /counters.xhtml un formulaire simple :

```
<h:form>
<h:commandButton value="Submit" />
</h:form>
```

3.4 Injection des Managed Beans

Pour faire le lien entre toutes ces instances, nous pouvons injecter dans une bean une autre instance (si elle a une portée supérieure ou égale).

Travail à faire : Ajoutez à ViewCounter la déclaration

```
@ManagedProperty("#{applicationCounter}")
ApplicationCounter appCounter;
```

et vérifiez dans la méthode annotée @PostConstruct que l'injection est réalisée. Nous pouvons donc, facilement, faire un lien entre les données globales (paramétres de l'application), les données session (identiçation de l'utilisateur) et les données vue (formulaire en cours de traitement).

4 Une application compléte

4.1 La couche métier

Notre application va gérer des enseignements. Commençons par la classe entité Course .

Nous n’utilisons les annotations JPA pour la persistance et les annotations de validation pour la vérification des données, mais nous créerons des classes dites POJO

```
package monapp;
import java.io.Serializable;
public class Course implements Serializable {
Integer id;
String name;
Integer hours;
String level;
String description;
public String getLevel() {
return level;
}
public void setLevel(String level) {
this.level = level;
}
public Long getId() {
return id;
}
public void setId(Long id) {
this.id = id;
}
public String getName() {
return name;
}
public void setName(String name) {
this.name = name;
}
public Integer getHours() {
return hours;
}
public void setHours(Integer hours) {
this.hours = hours;
}
public String getDescription() {
return description;
}
public void setDescription(String description) {
this.description = description;
}
}
```

Enchainons par un EJB de manipulation des enseignements :

```
package monapp;
import javax.ejb.Stateless;
import java.util.List;
@Stateless
public class CourseManager {
private List<Course> listeCourses;
public CourseManager(){
//creer la liste de course
}
public List<Course> findCourses() {
/
return listeCourses;
}
public Course findCourse(Integer n) {
// a coder
```

```
}
public Course saveCourse(Course c) {
if (c.getId() == null) {
        //ajout dans la liste
} else {
        //le rechercher et le modifier
}
return c;
}
public void deleteCourse(Course c) {
        //le supprimer
}
}
```

4.2 Le contrroleur

Nous allons définir un managed bean qui va se charger de toutes les opérations sur les enseignements. Nous lui injectons une instance de l'EJB affin qu'il puisse manipuler les données :

```
package monapp;
import java.util.List;
import javax.annotation.PostConstruct;
import javax.ejb.EJB;
import javax.faces.application.FacesMessage;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;
import javax.faces.context.FacesContext;
@ManagedBean(name = "course")
@SessionScoped
public class CourseControler {
@EJB
CourseManager cm;
Course theCourse = new Course();
@PostConstruct
public void init() {
System.out.println("Create " + this);
if (cm.findCourses().size() == 0) {
Course c1 = new Course();
c1.setName("Architecture JEE");
c1.setHours(60);
c1.setDescription("Introduction à JEE.");
cm.saveCourse(c1);
}
}
public List<Course> getCourses() {
return cm.findCourses();
}
public Course getTheCourse() {
return theCourse;
}
public String show(Integer n) {
theCourse = cm.findCourse(n);
return "showCourse";
}
public String save() {
cm.saveCourse(theCourse);
return "showCourse";
}
```

```
public String newCourse() {
theCourse = new Course();
return "editCourse";
}
}
```

4.3 Lister les enseignements

Nous allons commencer par définir une page pour les lister les enseignements courses.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:f="http://java.sun.com/jsf/core"
xmlns:h="http://java.sun.com/jsf/html"
xmlns:c="http://java.sun.com/jsp/jstl/core"
xmlns:ui="http://java.sun.com/jsf/facelets">
<h:body>
<ui:composition template="/template/mylayout.xhtml">
<ui:define name="content">
<f:view>
<h1>List of courses </h1>
<h:form>
<table>
<tr>
<th>Id</th>
<th>Name</th>
<th>Actions</th>
</tr>
<c:forEach var="c" items="#{course.courses}">
<tr>
<td>#{c.id}</td>
<td>#{c.name}</td>
<td><h:commandLink
action="#{course.show(c.id)}"
value="Show" />
</td>
</tr>
</c:forEach>
<tr>
</table>
</h:form>
</f:view>
</ui:define>
</ui:composition>
</h:body>
</html>
```

A ce stade le lien de visualisation ne fonctionne pas.

_ Nous pouvons remplacer l'itération JSTL par une tag JSF (<ui:repeat/>) qui nous permet d'obtenir un statut affin de suivre l'itération :

```
<ui:repeat var="c" varStatus="s" value="#{course.courses}">
<tr>
<td>#{s.index}</td>
<td>#{c.id}</td>
<td>#{c.name}</td>
<td><h:commandLink action="#{course.show(c.id)}"
value="Show" /></td>
```

```
</tr>
</ui:repeat>
```
_ Nous pouvons également utiliser un tag JSF pour construire un tableau de données (<h:dataTable/>) :
```
<h:dataTable value="#{course.courses}" var="o"
styleClass="course-table" headerClass="course-table-header"
footerClass="course-table-header"
rowClasses="course-table-odd-row,course-table-even-row">
<h:column>
<f:facet name="header">Number</f:facet>
<h:outputText value="#{o.id}" />
</h:column>
<h:column>
<f:facet name="header">Name</f:facet>
<h:outputText value="#{o.name}" />
</h:column>
<h:column>
<f:facet name="header">Hours</f:facet>
<h:outputText value="#{o.hours}" />
</h:column>
<h:column>
<f:facet name="header">Action</f:facet>
<h:commandLink value="Show" action="#{course.show(o.id)}" />
</h:column>
</h:dataTable>
```
N'oubliez-pas d'enrichir votre feuille CSS :
```
.course-table {
border-collapse: collapse;
border-left: 0px;
border-right: 0px;
}
.course-table-header {
text-align: center;
background: none repeat scroll 0 0 #E5E5E5;
border-top: 1px solid #BBBBBB;
border-bottom: 1px solid #BBBBBB;
padding: 5px;
}
.course-table-odd-row {
text-align: center;
background: none repeat scroll 0 0 #FFFFFFF;
border-top: 1px solid #BBBBBB;
}
.course-table-even-row {
text-align: center;
background: none repeat scroll 0 0 #F9F9F9;
border-top: 1px solid #BBBBBB;
}
```
_ Il nous reste maintenant à construire la page JSF qui va présenter un enseignement (/showCourse.xhtml).
Cette page utilise l'enseignement par défaut préparé par le contrôleur (de portée session) et accessible via la propriété theCourse du contrôleur :
```
<?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:f="http://java.sun.com/jsf/core"
```

```
xmlns:h="http://java.sun.com/jsf/html"
xmlns:ui="http://java.sun.com/jsf/facelets"
xmlns:c="http://java.sun.com/jsp/jstl/core">
<ui:composition template="/template/mylayout.xhtml">
<ui:define name="content">
<f:view>
<c:set var="c" value="#{course.theCourse}" />
<h1>Course #{c.name}</h1>
<p>Id: #{c.id}</p>
<p>Name: #{c.name}</p>
<p>Hours: #{c.hours}</p>
<p>Description: #{c.description}</p>
<p>
<h:link outcome="courses">List of courses</h:link>
</p>
</f:view>
</ui:define>
</ui:composition>
</html>
```

4.4 Modiér un enseignement

_ Pour offrir cette fonction, nous allons ajouter à la page showCourse.xhtml un lien de modification :

```
<h:form>
<p>
<h:commandLink action="editCourse">Modifier</h:commandLink>
</p>
</h:form>
```

La page d'édition (editCourse.xhtml) utilise elle aussi l'enseignement par défaut du contr^oleur :

20

```
<?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:f="http://java.sun.com/jsf/core"
xmlns:h="http://java.sun.com/jsf/html"
xmlns:c="http://java.sun.com/jsp/jstl/core"
xmlns:ui="http://java.sun.com/jsf/facelets">
<ui:composition template="/template/mylayout.xhtml">
<ui:define name="content">
<f:view>
<h1>Course edition</h1>
<h:form id="test">
<c:set var="c" value="#{course.theCourse}" />
<table border="1">
<tr>
<td><h:outputText value="Name :" /></td>
<td><h:inputText id="name" value="#{c.name}" />
<h:message style="color:red" for="name" /></td>
</tr>
<tr>
<td><h:outputText value="Hours :" /></td>
<td><h:inputText id="hours" value="#{c.hours}" />
<h:message style="color:red" for="hours" /></td>
</tr>
<tr>
```

```
<td><h:outputText value="Description :" /></td>
<td>
<h:inputTextarea id="description"
value="#{c.description}" cols="50" rows="10" />
<h:message style="color:red" for="description" />
</td>
</tr>
<tr>
<td colspan="2">
<h:commandButton
action="#{course.save()}" value="Save" /> |
<h:link outcome="courses" value="List of courses" />
</td>
</tr>
</table>
</h:form>
</f:view>
</ui:define>
</ui:composition>
</html>
```

Travail à faire : vérifier que la phase de modification fonctionne correctement et que les contraintes sont bien vérifiées.

Travail à faire : avec l'aide de ce tutoriel http://www.mkyong.com/jsf2/jsf-2-panelgrid-example/, utilisez le tag <h:panelGrid/> pour simplifier la mise en place du formulaire.

4.5 Ajouter des étapes de validation

Nous pouvons également ajouter de nouvelles contraintes en utilisant les tags de validation disponibles avec JSF.

En voici un exemple pour le nom :

```
<h:inputText id="name" value="#{c.name}" required="true"
requiredMessage="Le nom est obligatoire">
<f:validateLength minimum="3" maximum="10" />
</h:inputText>
```

Travail à faire : utilisez le tag <f:validateRegex/> pour valider un champ du formulaire (aide en ligne . http://www.mkyong.com/jsf2/jsf-2-validateregex-example/) :

```
<f:validateRegex pattern="expression" />
```

Travail à faire : avec l'aide de ce tutoriel http://www.mkyong.com/jsf2/customize-validation-error-message-in-jsf-2-0/, améliorez les messages d'erreur issus de la phase de validation.

Remarque : Vous trouverez sur ce site WEB http://www.mkyong.com/tutorials/jsf-2-0-tutorials/ d'autres exemples de validation.

4.6 Validation par programmation

Nous pouvons aussi programmer des étapes de validation au sein du contrôleur. Par exemple, vérifions que les heures d'enseignement sont multiples de trois :

```
public String save() {
if (theCourse.getHours() % 3 != 0) {
FacesContext ct = FacesContext.getCurrentInstance();
FacesMessage msg = new FacesMessage("Hours is not multiple of 3");
ct.addMessage("test:hours", msg);
ct.validationFailed();
return "editCourse";
}
cm.saveCourse(theCourse);
return "showCourse";
```

```
}
```
Travail à faire : Prévoir un traduction en français de ce message d'erreur.

4.7 Injecter des données dans un formulaire

_ Le niveau de l'enseignement (level) n'est actuellement pas proposé dans le formulaire d'édition. Pour l'ajouter, nous allons créer la liste des niveaux possibles à l'aide d'un contrôleur de portée application :

```
package monapp;
import java.util.LinkedHashMap;
import java.util.Map;
import javax.annotation.PostConstruct;
import javax.faces.bean.ApplicationScoped;
import javax.faces.bean.ManagedBean;
@ManagedBean(name = "params", eager = false)
@ApplicationScoped
public class ApplicationParameters {
Map<String, String> levels = new LinkedHashMap<String, String>();
@PostConstruct
void init() {
levels.put("--", "");
levels.put("Débutant", "D");
levels.put("Introduction", "I");
levels.put("Avancé", "A");
levels.put("Expert", "E");
System.out.println("Init " + this);
}
public Map<String, String> getLevels() {
return levels;
}
}
```

Remarque : le paramétre eager=false permet d'instancier le bean sur demande uniquement.

Modifiez le formulaire pour introduire le menu ci-dessous :

```
<h:outputText value="Level : " />
<h:selectOneMenu id="level" value="#{c.level}" required="false">
<f:selectItems value="#{params.levels}" />
</h:selectOneMenu>
```

4.8 Création d'un enseignement

Travail à faire : Ajoutez à votre application la possibilité de créer un nouvel enseignement (à partir de la page courses.xhtml).

5 Définir ses propres tags

Imaginons que nous voulons dans notre application produire des messages d'alertes tous identiques. Pour ce faire, nous allons définir une nouvelle balise (alert) et l'utiliser dans nos pages.

_ Préparez le fichier WebContent/resources/mytags/alert.xhtml ci-dessous :

```
<!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:composite="http://java.sun.com/jsf/composite"
xmlns:h="http://java.sun.com/jsf/html">
<h:body>
<composite:interface>
<composite:attribute name="title" required="false" default="ALERT !" />
```

```
</composite:interface>
<composite:implementation>
<div
style="margin: 30px; width: 500px; border: solid 2px red;
background: #FFAAAA; color: red;">
<div style="padding: 3px; border-bottom: solid 2px red;">
<b><h:outputLabel value="#{cc.attrs.title}" /></b>
</div>
<div style="padding: 10px;">
<composite:insertChildren />
</div>
</div>
</composite:implementation>
</h:body>
</html>
```

Nous commençons par définir l'interface (les attributs utilisables dans le tag) puis le corps de la balise personnalisée. Nous pouvons ainsi définir plusieurs balises dans le même répertoire, voir faire plusieurs sous-répertoires.

Note : Vous remarquerez que l'accès aux attributs passe par la notation #fcc.attrs.nomAttributg .

_ Nous pouvons maintenant utiliser cette balise alert dans une de nos pages :

```
<?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:f="http://java.sun.com/jsf/core"
xmlns:h="http://java.sun.com/jsf/html"
xmlns:ui="http://java.sun.com/jsf/facelets"
xmlns:mt="http://java.sun.com/jsf/composite/mytags">
<ui:composition template="/template/mylayout.xhtml">
<ui:define name="content">
<h1>Alert test</h1>
<mt:alert title="Une alerte">
<span>C'est grave ?</span>
</mt:alert>
<mt:alert>
<span>C'est vraiment grave ?</span>
</mt:alert>
</ui:define>
</ui:composition>
</html>
```

Note : Vous remarquerez que le lien est réalisé par un nouvel espace de nom basé sur composite auquel nous avons ajouté le nom du répertoire.

6 Approfondissements

6.1 Un formulaire de test

_ Commencez par créer un nouveau formulaire de test formtest.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:f="http://java.sun.com/jsf/core"
xmlns:h="http://java.sun.com/jsf/html"
xmlns:ui="http://java.sun.com/jsf/facelets">
```

```
<ui:composition template="/template/mylayout.xhtml">
<ui:define name="content">
<h1>Test Form</h1>
<h:form id="formtest">
<h:panelGrid columns="2">
<h:outputText value="Input text&amp;#160;:" />
<h:panelGroup>
<h:inputText id="text" value="#{formTest.text}"
required="true"
requiredMessage="Le texte est obligatoire">
<f:validateLength minimum="3" maximum="15" />
</h:inputText>
<h:message errorClass="error" for="text" />
</h:panelGroup>
<h:commandButton value="Submit" action="#{formTest.submit}" />
<span></span>
</h:panelGrid>
</h:form>
</ui:define>
</ui:composition>
</html>
```

Et le bean managé qui va avec :

```
package monapp;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;
@ManagedBean(name = "formTest")
@SessionScoped
public class FormTestControler {
private String text = "X";
public String submit() {
System.out.println("LOG: Submit");
return null;
}
public String getText() {
return text;
}
public void setText(String text) {
this.text = text;
System.out.println("LOG: Set text with " + text);
}
}
```

Travail à faire : vérifiez le bon fonctionnement de ce formulaire.

6.2 Surveiller le Workow

_ Pour mieux comprendre le Workow suivi par JSF dans le traitement d'une requête nous allons installer un espion. Préparez la classe MyPhaseListener :

```
package monapp;
import javax.faces.event.PhaseEvent;
import javax.faces.event.PhaseId;
import javax.faces.event.PhaseListener;
public class MyPhaseListener implements PhaseListener {
private static final long serialVersionUID = 1L;
@Override
public void beforePhase(PhaseEvent pe) {
System.out.println("BEFORE Phase " + pe.getPhaseId());
}
@Override
```

```
public void afterPhase(PhaseEvent pe) {
System.out.println("AFTER Phase " + pe.getPhaseId());
}
@Override
public PhaseId getPhaseId() {
return PhaseId.ANY_PHASE;
}
}
```

et installez cet espion dans le fichier faces-config.xml :

```
<?xml version="1.0" encoding="UTF-8"?>
<faces-config ... >
<lifecycle>
<phase-listener>monapp.MyPhaseListener</phase-listener>
</lifecycle>
...
</faces-config>
```

Travail à faire : Testez le bon fonctionnement de cet espion. Quelles sont les étapes quand nous appelons directement la page ? Quelles sont les étapes quand nous soumettons avec erreurs puis sans erreurs ?

Il existe six étapes :

_ Restore View : JSF reconstruit l'arborescence des composants qui composent la page.

_ Apply Requests : les valeurs des données sont extraites de la requête

_ Process Validations : procède à la validation des données

_ Update model values : mise à jour du modèle selon les valeurs reçues si validation ou conversion réussie

_ Invoke Application : les événements émis de la page sont traités. Elle permet de déterminer la prochaine page

_ Render Response : création du rendue de la page

Vous devriez retrouver ces étapes grâce aux traces émises par le contrôleur.

6.3 Contrôler la validation

_ Nous pouvons améliorer la validation en ajoutant des contraintes :

```
<h:outputText value="Input text :" />
<h:panelGroup>
<h:inputText id="text" value="#{formTest.text}" required="true"
requiredMessage="Le nom est obligatoire">
<f:validateLength minimum="3" maximum="15" />
<f:validator for="text" validatorId="myconstraints.Hello" />
</h:inputText>
<h:message errorClass="error" for="text" />
</h:panelGroup>
```

avec la classe :

```
package monapp;
import javax.faces.application.FacesMessage;
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.validator.FacesValidator;
import javax.faces.validator.Validator;
import javax.faces.validator.ValidatorException;
@FacesValidator("myconstraints.Hello")
public class HelloValidator implements Validator {
@Override
public void validate(FacesContext ct, UIComponent comp, Object obj)
throws ValidatorException {
System.out.println("LOG: HelloValidator on " + comp);
```

```
String value = obj.toString();
if (!value.contains("hello")) {
FacesMessage msg = new FacesMessage(FacesMessage.SEVERITYéRROR,
"hello absent", "Il manque le mot 'hello'");
throw new ValidatorException(msg);
}
}
}
```

Travail à faire : Vérifiez le bon fonctionnement de cette validation programmée. Suivez dans les traces le moment de la validation.

6.4 Convertir les dates

_ Ajouter à votre formulaire un nouveau champ :

```
<h:outputText value="Input date :" />
<h:panelGroup>
<h:inputText id="birthday" value="#{formTest.birthday}"
label="Birthday">
<f:convertDateTime pattern="dd-MM-yyyy" />
</h:inputText>
<h:message errorClass="error" for="birthday" />
</h:panelGroup>
```

et modiéz le contr^oleur :

```
...
public class FormTestControler {
...
@Past(message = "Trop récent !")
private Date birthday = new Date();
public Date getBirthday() {
return birthday;
}
public void setBirthday(Date birthday) {
this.birthday = birthday;
System.out.println("LOG: Set birthday with " + birthday);
}
}
```

Travail à faire : vérifiez la conversion de la date et sa bonne traduction et validation.

Note : La conversion peut également être utilisée lors d'un affichage :

```
<h:outputText value="Output date :" />
<h:outputText value="#{formTest.birthday}">
<f:convertDateTime pattern="dd-MM-yyyy" />
</h:outputText>
```

6.5 Convertir les nombres

_ Continuons en ajoutant un champ de type nombre :

```
<h:outputText value="Input number :" />
<h:panelGroup>
<h:inputText id="number" value="#{formTest.number}" required="true">
<f:validateLongRange minimum="50" />
<f:convertNumber type="currency" currencySymbol="$"
minFractionDigits="2" maxFractionDigits="3" />
</h:inputText>
<h:message errorClass="error" for="number" />
</h:panelGroup>
```

et modifions à nouveau le contrôleur :

```
...
public class FormTestControler {
```

```
...
private Double number = 100.0;
public Double getNumber() {
return number;
}
public void setNumber(Double number) {
this.number = number;
System.out.println("LOG: Set number with " + number);
}
}
```

Travail à faire : vérifiez la conversion du nombre ainsi que sa bonne traduction et validation.

6.6 Définir sa conversion

_ Suivez cet exemple http://www.mkyong.com/jsf2/custom-converter-in-jsf-2-0/ pour construire votre propre outil de conversion.

6.7 Les événements JSF

_ Durant le traitement de la requête, le moteur JSF génère des événements auxquels nous pouvons nous abonner.

Ajoutez un attribut valueChangeListener à la balise de saisie du nombre :

```
<h:outputText value="Input number :" />
<h:panelGroup>
<h:inputText id="number" value="#{formTest.number}" required="true"
valueChangeListener="#{formTest.numberChanged}">
<f:validateLongRange minimum="50" />
<f:convertNumber type="currency" currencySymbol="$"
minFractionDigits="2" maxFractionDigits="3" />
</h:inputText>
<h:message errorClass="error" for="number" />
</h:panelGroup>
```

et préparez, dans le contrôleur, la nouvelle méthode :

```
public void numberChanged(ValueChangeEvent e) {
System.out.println("LOG: old number = " + e.getOldValue());
System.out.println("LOG: new number = " + e.getNewValue());
}
```

Travail à faire : Vérifiez dans les traces le moment d'apparition de ces événements.

_ Nous pouvons également capter les événements liés à la soumission du formulaire. Modifiez le bouton de soumission de la manière suivante :

```
<h:commandButton value="Submit" action="#{formTest.submit}">
<f:attribute name="forActionlistener" value="Hello" />
<f:actionListener type="monapp.MyActionListener" />
</h:commandButton>
```

et préparez la classe :

```
package monapp;
import javax.faces.component.UIComponent;
import javax.faces.event.AbortProcessingException;
import javax.faces.event.ActionEvent;
import javax.faces.event.ActionListener;
public class MyActionListener implements ActionListener {
@Override
public void processAction(ActionEvent evt) throws AbortProcessingException {
UIComponent c = evt.getComponent();
System.out.println("LOG: actionEvent sur " + c);
System.out.println("LOG: actionEvent attribute = "
+ c.getAttributes().get("forActionlistener"));
```

```
}
}
```
Note : Nous pouvons donc ajouter, avant la phase de soumission, des traitements communs éventuellement paramétrés (avec f:attribute).
_ Finalement, nous pouvons aussi indiquer une méthode pour traiter les événements (attribut actionListener) :
```
<h:commandButton value="Submit" action="#{formTest.submit}"
actionListener="#{formTest.myListener}">
<f:setPropertyActionListener target="#{formTest.parameter}"
value="Fin" />
<f:attribute name="forActionlistener" value="Hello" />
<f:actionListener type="monapp.MyActionListener" />
</h:commandButton>
<span></span>
```
et nous ajoutons dans le contrôleur :
```
public void setParameter(String value) {
System.out.println("LOG: Fix parameter with " + value);
}
public void myListener(ActionEvent evt) {
UIComponent c = evt.getComponent();
System.out.println("LOG: method actionEvent sur " + c);
}
```

6.8 Les widgets JSF

_ Avec l'aide des exemples ci-dessous et de cette documentation http://corejsf.com/refcard.html, essayez d'utiliser d'autres widgets de JSF :
_ <h:inputSecret/> /> http://www.mkyong.com/jsf2/jsf-2-password-example/
_ <h:inputHidden/> http://www.mkyong.com/jsf2/jsf-2-hidden-value-example/
_ <h:selectBooleanCheckbox/> http://www.mkyong.com/jsf2/jsf-2-checkboxes-example/
_ <h:selectOneRadio/> http://www.mkyong.com/jsf2/jsf-2-radio-buttons-example/

7 Traitement AJAX

Pour l'instant, nous utilisons un mode de communication client/serveur classique dans lequel chaque interaction nécessite
_ client : l'envoie d'une requête HTTP
_ serveur : le décodage et le traitement de cette requête
_ serveur : la construction d'une réponse (page XHTML complète)
_ client : l'analyse et l'affichage de cette page
Ce mode de fonctionnement pose plusieurs problèmes :
_ les pages complètes transitent par le réseau (trop de données)
_ le rafraîchissement côté client n'est pas très agréable (effet de flip/flap)
_ si une petite partie de la page change, la page complète doit être reconstruite (pas facile si la page est sophistiquée)
Pour corriger ces problèmes, les applications Web modernes utilisent une architecture AJAX : côté client, un code JavaScript va lancer des requêtes asynchrones au serveur affinn d'obtenir des données (au format XML et/ou JSON) et de modifier dynamiquement la page en cours de visualisation (à l'aide de l'API DOM).
Dans JSF 2 ce mode de fonctionnement est pris en compte et géré par la balise <h:ajax/> .

7.1 Un exemple simple

Considérons la page /ajaxSample.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:f="http://java.sun.com/jsf/core"
xmlns:h="http://java.sun.com/jsf/html"
xmlns:ui="http://java.sun.com/jsf/facelets">
<ui:composition template="/template/mylayout.xhtml">
<ui:define name="content">
<h1>Ajax Test</h1>
<p>Now = <h:outputText id="now" value="#{ajaxBean.now}" /></p>
<h2>No AJAX</h2>
<h:form>
<h:inputText id="text" value="#{ajaxBean.text}" />
<h:commandButton value="Show" />
<br />
<h:outputText id="affichage" value="#{ajaxBean.text}" />
</h:form>
</ui:define>
</ui:composition>
</html>
```

et le bean :

```
package monapp;
import java.io.Serializable;
import java.util.Date;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.ViewScoped;
@ManagedBean
@ViewScoped
public class AjaxBean implements Serializable {
private static final long serialVersionUID = 5443351151396868724L;
private String text = "";
public AjaxBean() {
}
public String getText() {
return text;
}
public void setText(String text) {
this.text = text;
System.err.println("setText1 with " + text);
}
public String getNow() {
return new Date().toString();
}
}
```

Pour l'instant, le fonctionnement AJAX n'est pas utilisé et nous avons un formulaire classique. L'affichage de la date permet de suivre les rafraîchissements complets de la page.

7.2 Utiliser AJAX

Ajoutons à la page ce deuxième code :

```
<h2>AJAX 1</h2>
<h:form>
<h:inputText id="text" value="#{ajaxBean.text}" />
<h:commandButton value="Show">
<f:ajax execute="text" render="affichage" />
</h:commandButton>
<br />
<h:outputText id="affichage" value="#{ajaxBean.text}" />
```

```
</h:form>
```
L'attribut exécute indique les éléments envoyés au serveur (éventuellement séparés par des espaces) et l'attribut render indique les éléments à rafraîchir après le traitement AJAX. Dans ces attributs, vous pouvez utiliser
_ idElement pour l'élément identifié (notre exemple),
_ @this pour l'élément concerné,
_ @form pour tous les éléments du formulaire,
_ @all pour tous les éléments,
_ @none pour aucun.
_ #fexpressiong pour calculer les éléments (une Collection de String)
Travail à faire :
_ Tester cet exemple.
_ Si execute="@this" quel est le résultat ?
_ Si execute="text" et render='affichage now' quel est le résultat ?
_ Pour finir, essayez render='@all' .

7.3 Traiter les événements avec AJAX

Ajoutons à la page ce troisième code :
```
<h2>AJAX Events</h2>
<h:form>
<h:inputText id="text" value="#{ajaxBean.text}">
<f:ajax event="keyup" render="affichage" />
</h:inputText>
<br />
<h:outputText id="affichage" value="#{ajaxBean.text}" />
</h:form>
```
Dans cet exemple, nous captons les événements keyup pour déclencher le traitement AJAX qui va rafraichir le champ affichage .
Les événements possibles sont blur , change , click , dblclick , focus , keydown , keypress , keyup , mousedown , mousemove , mouseout , mouseover , mouseup , select , valueChange , action .
Travail à faire :
_ Tester cet exemple.
_ Tester un autre événement (comme focus).
_ Ajoutez un nouveau traitement AJAX basé sur le double-click :
```
<f:ajax event="dblclick" render="affichage"
listener="#{ajaxBean.toUpper}" />
```
et ajoutez la méthode toUpper à votre bean :
```
public void toUpper(AjaxBehaviorEvent event) {
text = text.toUpperCase();
}
```
_ Pour finir, essayez l'événement valueChange et faites en sorte de conserver la valeur passée en majuscules.

7.4 Validation avec AJAX

Ajoutons à la page ce quatrième code :
```
<h2>AJAX Validation on submit</h2>
<h:form>
<h:inputText id="text4" value="#{ajaxBean.text}">
<f:validateLength minimum="3" maximum="8" />
</h:inputText>
<h:message id="messageText4" for="text4" />
<br />
```

```
<h:commandButton value="Ok">
<f:ajax execute="text4" render="@form" />
</h:commandButton>
</h:form>
```
Nous déclenchons la validation du formulaire en mode AJAX sans générer de requête en mode POST. Testez cet exemple.
Nous pouvons aussi demander la validation d'un champ à chaque changement :
```
<h2>AJAX Validation on event</h2>
<h:form>
<h:inputText id="text" value="#{ajaxBean.text}">
<f:validateLength minimum="3" maximum="8" />
<f:ajax event="keyup" render="messageText" />
</h:inputText>
<h:message id="messageText" for="text" />
</h:form>
```
Avec ce mécanisme, nous pouvons implanter l'exemple classique de la validation d'un identifiant avec vérification en base de son unicité.

7.5 Un traitement complet en AJAX

Nous allons gérer une liste de villes en mode AJAX. Commencez par ajouter ces méthodes à votre bean :
```
List<String> cities = new LinkedList<String>();
public List<String> getCities() {
return cities;
}
public void setCities(List<String> cities) {
this.cities = cities;
}
public void addCity() {
if (text.trim().length() > 0) {
cities.add(text);
System.err.println("add " + text);
text = "";
}
}
public void removeCity(int index) {
cities.remove(index);
}
```
Nous pouvons maintenant prévoir un affichage des villes (notez l'attribut rendered) et une soumission en AJAX
pour ajouter une nouvelle ville et mettre à jour la liste (toujours en AJAX) :
```
<h2>CRUD in AJAX</h2>
<h:form>
<h:panelGroup rendered="#{ajaxBean.cities.size() > 0}">
<p>Cities :</p>
<ul>
<ui:repeat var="city" varStatus="s" value="#{ajaxBean.cities}">
<li><h:outputText value="#{city}" /></li>
</ui:repeat>
</ul>
</h:panelGroup>
<h:inputText id="city" value="#{ajaxBean.text}">
<f:validateLength minimum="3" maximum="8" />
</h:inputText>
<h:message id="messageCity" for="city" />
<br />
```

```
<h:commandButton value="Add" action="#{ajaxBean.addCity()}">
<f:ajax execute="@this city" render="@form" />
</h:commandButton>
</h:form>
```

Travail à faire : ajoutez pour chaque ville un bouton de suppression qui va appeler la méthode removeCity(s.index) et mettre à jour le formulaire.

Travail à faire : ajoutez pour chaque ville un bouton de modification qui permet de changer le nom de la ville. Vous devez

_ prévoir une nouvelle méthode updateCity(index)

_ modifier la fonction addCity() pour quelle gère également la mise à jour

_ prévoir la suppression d'une ville qui est en-cours de modification

8 PrimeFaces

L'un des avantages de JSF est la possibilité d'étendre la plateforme en utilisant des widgets définis dans des bibliothéques annexes. Nous pouvons en citer trois : PrimeFaces . http://primefaces.org/, Richefaces http://richfaces.jboss.org/ ou IceFaces http://www.icesoft.org/java/projects/ICEfaces/overview.jsf.

_ Vous pouvez à présent tester cette première page apres avoir ajouter les composants dans votre framework du projet :

```
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:h="http://java.sun.com/jsf/html"
xmlns:p="http://primefaces.org/ui">
<h:head>
</h:head>
<h:body>
<p:spinner />
</h:body>
</html>
```

Observez le code source généré. La bibliothèque primeFaces utilise JQuery pour les interactions JavaScript avec le client.

_ Dans vos pages, remplacez la datatable de JSF par celle de PrimeFaces http://www.primefaces.org/showcase/ui.

_ Modiéz dans vos formulaires la saisie des dates en utilisant le composant Primefaces.

_ Utilisez les outils de validation de Primefaces.

_ Testez l'exemple des boites de dialogue.