

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 composé 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/javasee/javaserverfaces-139869.html>

API JSF 2.0 <https://jaserverfaces.java.net/nonav/docs/2.0/javadocs/>

Tags JSF 2.0 [http://jaserverfaces.java.net/nonav/docs/2.0/vldocs/facelets/](http://jaserverfaces.java.net/nonav/docs/2.0/vlddocs/facelets/)

myFaces : l'implantation libre proposée par Apache <http://myfaces.apache.org/>

Des **cours** sur JSF :

- Un cours complet en français¹ (et pas seulement sur JSF),
- 176 très bons transparents en français sur JSF²,
- Une liste de cours sur le sujet³,
- Un site associé à un livre⁴.
- Des cours en ligne⁵ (pas seulement sur JSF).

2 Mise en place

Suivez les étapes dans l'ordre pour obtenir une petite application JSF :

- Créez une application WEB dynamique avec **Eclipse JEE** et associez cette application à un serveur **TomEE**. Nous allons utiliser l'implantation MyFaces⁶ de JSF intégrée à **TomEE**.
- Ajoutez à votre projet la facette **JavaServer faces** en version 2.2 (menu *Properties/Project Facets*). N'utilisez aucune librairie additionnelle (*Further configuration available*).
- Créez le fichier **WEB-INF/web.xml** listé ci-dessous :

1. <http://www.jmdoudoux.fr/java/dej/chap-jsf.htm#jsf>
2. <http://mbaron.developpez.com/javasee/jsf/>
3. <http://www.jsftutorials.net/>
4. <http://corejsf.com/>
5. <http://www.coreservlets.com/JSF-Tutorial/>
6. <http://myfaces.apache.org>

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
  id="WebApp_ID" version="3.0">
  <display-name>JSF Application</display-name>
  <description>Exemple JSF</description>

  <!-- Déclaration de la servlet générique JSF -->
  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <!-- URL traitées par cette servlet -->
  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>/faces/*</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>*.xhtml</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>*.jsf</url-pattern>
  </servlet-mapping>

  <!-- Paramétrage du projet -->
  <context-param>
    <param-name>javax.faces.PROJECT_STAGE</param-name>
    <param-value>Development</param-value>
    <!-- <param-value>Production</param-value> -->
  </context-param>
  <context-param>
    <param-name>org.apache.myfaces.annotation.SCAN_PACKAGES</param-name>
    <param-value>monapp</param-value>
  </context-param>
</web-app>

```

- Créez (ou vérifiez) le fichier `WEB-INF/faces-config.xml` listé ci-dessous :

```

<?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">

</faces-config>

```

- 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 identifier les balises que seront traitées avant l'envoi 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 récupère 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 préparer 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" />
      <h:graphicImage library="theme1" name="calanques.jpg" />
      <h:outputScript library="js" name="hello.js" />
    </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.

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

Travail à faire : Faites de même avec `<h:outputScript/>` et le script ci-dessous. Observez le code source généré.

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

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 créer le fichier `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 fichiers 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 **après** 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 après). Revenez ensuite à `<ui:composition/>`.

Travail à faire : Testez le `Ctrl + Shift + D` pour avoir accès 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 :

```

<?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 (`{...}`). Il n'a pas la même syntaxe que celui des pages JSP et nous donnerons des éléments supplémentaires au fur et à mesure du TP.

- Après avoir testé cette version, modifiez 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 guide⁷.

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>

```

7. <http://www.laliluna.de/javaxserver-faces-message-resource-bundle-tutorial.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 règle 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 *managed bean* : c'est un 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.

4 Une application complète

4.1 La couche métier

Notre application va gérer des enseignements. Commençons par la classe entité `Course`. Nous utilisons les annotations JPA pour la persistance et les annotations de validation pour la vérification des données.

```

package monapp;

import java.io.Serializable;
import javax.persistence.*;
import javax.validation.constraints.*;

@Entity
public class Course implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id @GeneratedValue
    Long id;

    @Column
    @NotNull @Size(min = 5, max = 200)
    String name;

    @Column
    @NotNull @Min(5) @Max(600)
    Integer hours;

    @Column
    String level;

    @Lob
    @NotNull @Size(min = 300, max = 4000)
    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;
    }
}

```

Continuons avec le fichier de configuration JPA (`WebContent/META-INF/persistence.xml`) basé sur la `DataSource` par défaut de `TomEE` :

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
  version="1.0">

  <persistence-unit name="myData" transaction-type="JTA">
    <properties>
      <property name="openjpa.jdbc.SynchronizeMappings"
        value="buildSchema(ForeignKeys=true)" />
    </properties>
  </persistence-unit>

</persistence>
```

Enchainons par un EJB de manipulation des enseignements :

```
package monapp;

import java.util.List;

import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

@Stateless
public class CourseManager {

    @PersistenceContext(unitName = "myData")
    EntityManager em;

    public List<Course> findCourses() {
        return em.createQuery("Select c From Course c", Course.class)
            .getResultList();
    }

    public Course findCourse(Integer n) {
        return em.find(Course.class, n);
    }

    public Course saveCourse(Course c) {
        if (c.getId() == null) {
            em.persist(c);
        } else {
            c = em.merge(c);
        }
        return c;
    }

    public void deleteCourse(Course c) {
        c = em.merge(c);
        em.remove(c);
    }
}
```

4.2 Le contrôleur

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 afin 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 CourseController {

    @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>
            </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 afin 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 #FFFFFF;
  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 Modifier 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 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: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 tutoriel⁸, 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 :

8. <http://www.mkyong.com/jsf2/jsf-2-panelgrid-example/>

```
<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 ligne⁹) :

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

Travail à faire : avec l'aide de ce tutoriel¹⁰, améliorez les messages d'erreur issus de la phase de validation.

Remarque : Vous trouverez sur ce site WEB¹¹ 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 :

9. <http://www.mkyong.com/jsf2/jsf-2-validateregex-example/>
10. <http://www.mkyong.com/jsf2/customize-validation-error-message-in-jsf-2-0/>
11. <http://www.mkyong.com/tutorials/jsf-2-0-tutorials/>

```

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 paramètre `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 `#{cc.attrs.nomAttribut}`.

- 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&#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 Workflow

- Pour mieux comprendre le **Workflow** 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 rendu 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.SEVERITY_ERROR,
                "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&#160;:" />
<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 modifiez le contrôleur :

```

...

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&#160;:" />
<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&#160;:" />
<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 exemple¹² pour construire votre propre outil de conversion.

6.7 Les évènements JSF

- Durant le traitement de la requête, le moteur JSF génère des évènements auxquels nous pouvons nous abonner. Ajoutez un attribut `valueChangeListener` à la balise de saisie du nombre :

```

<h:outputText value="Input number&#160;:" />
<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 évènements.

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

12. <http://www.mkyong.com/jsf2/custom-converter-in-jsf-2-0/>

```

<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 documentation¹³, essayez d'utiliser d'autres widgets de JSF :
 - `<h:inputSecret/>`¹⁴

13. <http://corejsf.com/refcard.html>

14. <http://www.mkyong.com/jsf2/jsf-2-password-example/>

- `<h:inputHidden/>` ¹⁵
- `<h:selectBooleanCheckbox/>` ¹⁶
- `<h:selectOneRadio/>` ¹⁷

7 PrimeFaces

L'un des avantages de JSF est la possibilité d'étendre la plateforme en utilisant des widgets définis dans des bibliothèques annexes. Nous pouvons en citer trois : PrimeFaces ¹⁸, Richfaces ¹⁹ ou IceFaces ²⁰.

- Commencez par récupérer le fichier `primefaces-5.1.jar` ²¹ et placez le dans le répertoire `WEB-INF/lib`. Vous pouvez à présent tester cette première page :

```
<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 ²².
- Modifiez dans vos formulaires la saisie des dates en utilisant le composant Primefaces ²³.
- Utilisez les outils de validation de Primefaces ²⁴.

15. <http://www.mkyong.com/jsf2/jsf-2-hidden-value-example/>

16. <http://www.mkyong.com/jsf2/jsf-2-checkboxes-example/>

17. <http://www.mkyong.com/jsf2/jsf-2-radio-buttons-example/>

18. <http://primefaces.org/>

19. <http://richfaces.jboss.org/>

20. <http://www.icesoft.org/java/projects/ICEfaces/overview.jsf>

21. [ress-jsf](http://www.mkyong.com/jsf2/jsf-2-hidden-value-example/)

22. <http://www.primefaces.org/showcase/ui/data/datatable/basic.xhtml>

23. <http://www.primefaces.org/showcase/ui/input/calendar.xhtml>

24. <http://www.primefaces.org/showcase/ui/csv/bean.xhtml>