

# **JSF (Java Server Faces)**

Guillaume Defrain  
Octobre 2016



# Introduction : qu'est-ce-que c'est ...

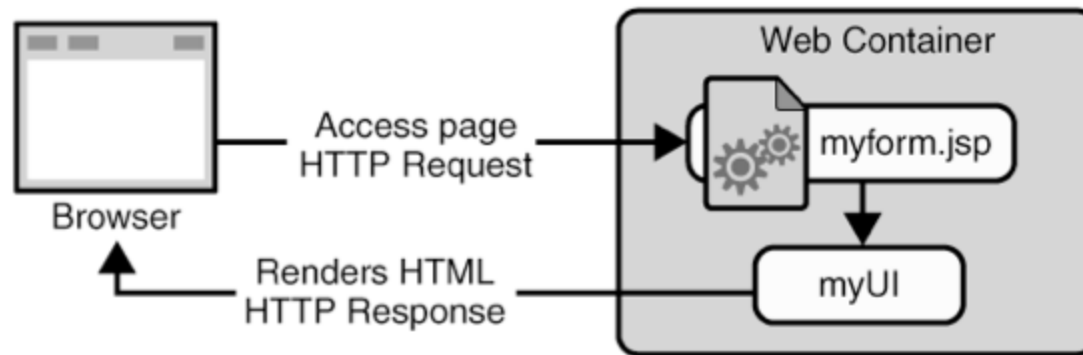
- Java Server Faces est un framework de développement d'applications Web en Java permettant de respecter le modèle d'architecture MVC et basé sur des composants côté présentation
- Java Server Faces permet
  - une séparation de la couche présentation des autres couches (MVC)
  - un mapping entre l'HTML et l'objet
  - un ensemble de composants riches et réutilisables
  - une liaison simple entre les actions côté client de l'utilisateur (event listener) et le code Java côté serveur
  - Création de nouveaux composants graphiques
  - JSF peut être utilisé pour générer autre chose que du HTML (XUL, XML, WML, ...)

# Introduction : une spec et plusieurs implémentations

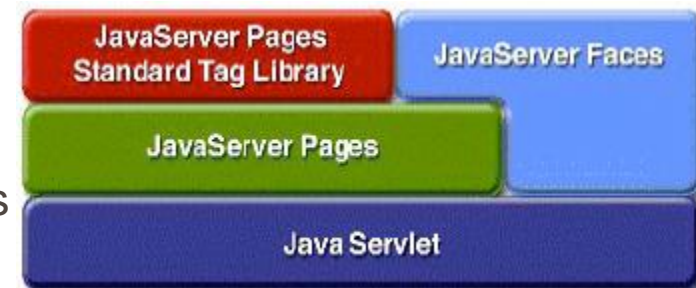
- JSF comme la plupart des technologies proposées par Sun est définie dans une spécification JSR-127 (2004 - version 1.1 → J2EE 1.4) puis JSR-252 (2006 - 1.2 → Java EE 5) puis JSR-314 (2009 - 2.0 → Java EE 6)
  - Il existe donc plusieurs implémentations de JSF
    - Oracle Mojarra : <http://javaserverfaces.java.net/>
    - Apache MyFaces : <http://myfaces.apache.org>
  - Apache fournit des fonctionnalités additionnels via le sous projet : **Tomahawk**
    - Composants graphiques
    - Validators plus fournis
- Il existe aussi Trinidad, Tobago, ICEFaces, RichFaces, PrimeFaces...

# Introduction : comment ça marche ...

- L'interface utilisateur construite dans la page JSP est générée à l'aide de la technologie JSF (résultat myUI)
- Elle fonctionne sur le serveur et le rendu est retourné au client



- JSF s'appuie sur les technologies précédentes
  - Génération en Servlet
  - Utilisation des composants JSF dans les pages JSP
  - Les composants JSF sont exposés aux JSPs grâce aux balises personnalisés



# Introduction : principe pour traiter un formulaire

1. Construire le formulaire dans une page JSP en utilisant les balises JSF
2. Développer un Bean qui effectue un « Mapping » avec les valeurs du formulaire
3. Modifier le formulaire pour spécifier l'action et l'associer au Bean
4. Fournir des Converters et des Validators pour traiter les données du formulaire
5. Paramétrer le fichier faces-config.xml pour déclarer le Bean et les règles de navigation
6. Créer les pages XHTML correspondant à chaque condition de retour

# Configuration : JSF dans le web.xml

## 1/3

- Nécessite la configuration du fichier **web.xml** de façon à ce que JSF soit pris en compte
  - Paramétrer le fonctionnement général de l'application : le contrôleur
  - Identifier la servlet principale : `javax.faces.webapp.FacesServlet`
- Spécifier le nom et le chemin du fichier de configuration
  - Nom du paramètre : `javax.faces.application.CONFIG_FILES`
  - Exemple : `/WEB-INF/faces-config.xml`
- Spécifie où l'état de l'application doit être sauvé
  - Nom du paramètre : `javax.faces.STATE_SAVING_METHOD`
  - Valeurs possibles : `client` ou `server`
- Valider ou pas les fichiers XML
  - Nom du paramètre : `com.sun.faces.validateXml`
  - Valeurs possibles : `true` ou `false` (défaut : `false`)

# Configuration : JSF dans le web.xml

## 2/3

- Indique si les objets développés tels que les Beans, les composants, les validators et les converters doivent être créés au démarrage de l'application
  - Nom du paramètre : `com.sun.faces.verifyObjects`
  - Valeurs possibles : `true` ou `false` (défaut : `false`)
- La servlet principale est le point d'entrée d'une application JSF
  - On trouve plusieurs manières de déclencher des ressources JSF
    - Préfixe `/faces/`
    - Suffixes `*.jsf` ou `*.faces`
  - Exemples (le contexte de l'application est `myAppli`)
    - `http://localhost/myAppli/faces/index.jsp`
    - `http://localhost/myAppli/index.jsf`

# Configuration : JSF dans le web.xml

## 3/3

- Exemple : paramétrer une application Web de type

Utilisation de *context-param* pour paramétrer le fonctionnement des JSF

```
...  
<context-param>  
  <param-name>com.sun.faces.validateXml</param-name>  
  <param-value>true</param-value>  
</context-param>  
<servlet>  
  <servlet-name>Faces Servlet</servlet-name>  
  <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>  
  <load-on-startup>1</load-on-startup>  
</servlet>  
<servlet-mapping>  
  <servlet-name>Faces Servlet</servlet-name>  
  <url-pattern>/faces/*</url-pattern>  
</servlet-mapping>  
<servlet-mapping>  
  <servlet-name>Faces Servlet</servlet-name>  
  <url-pattern>*.faces</url-pattern>  
</servlet-mapping>  
...
```

La Servlet qui gère les entrées au contexte JSF

Comment accéder à la Servlet « Faces Servlet »

*web.xml*



# Configuration : le fichier contrôleur « faces-config.xml »

- Le fichier gérant la logique de l'application web s'appelle par défaut faces-config.xml
- Il est placé dans le répertoire WEB-INF au même niveau que web.xml
- Il décrit essentiellement six principaux éléments :
  - les Beans managés <managed-bean>
  - les règles de navigation <navigation-rule>
  - les ressources éventuelles suite à des messages <message-bundle>
  - la configuration de la localisation <resource-bundle>
  - la configuration des Validators et des Converters <validator> <converter>
  - d'autres éléments liés à des nouveaux composants JSF <render-kit>
- Le fichier de configuration est un fichier XML décrit par une DTD. La balise de départ est <faces-config> (version 1.1)
- La version 1.2 de JSF utilise un schéma au lieu d'une DTD
- A partir de la version 2 la plupart de ces éléments peuvent être positionné directement dans le code à l'aide d'annotations

# Configuration : le fichier contrôleur « faces-config.xml »

```
<?xml version="1.0" encoding="UTF-8"?>
<faces-config
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-facesconfig_2_0.xsd"
  version="2.0">
  <navigation-rule>
  ...
</navigation-rule>
<managed-bean>
  ...
</managed-bean>
</faces-config>
```

# Configuration : balises personnalisées dans les JSP

- Les composants JSF sont utilisés dans les pages JSP au moyen de balises personnalisées dédiées aux JSF
  - CORE : noyau qui gère les vues
  - HTML : composants JSF
- Description des balises personnalisées Core et HTML
  - [http://java.sun.com/javaee/javaxserverfaces/1.2\\_MR1/docs/tlddocs](http://java.sun.com/javaee/javaxserverfaces/1.2_MR1/docs/tlddocs)
  - <http://www.horstmann.com/corejsf/jsf-tags.html>
- Possibilité d'utiliser d'autres balises (Tomahawk...)
- Les composants JSF doivent être encadrés par une vue JSF déclarée par la balise <core:view>

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

Utilisation des composants JSF

# Configuration : JSF et Tomcat

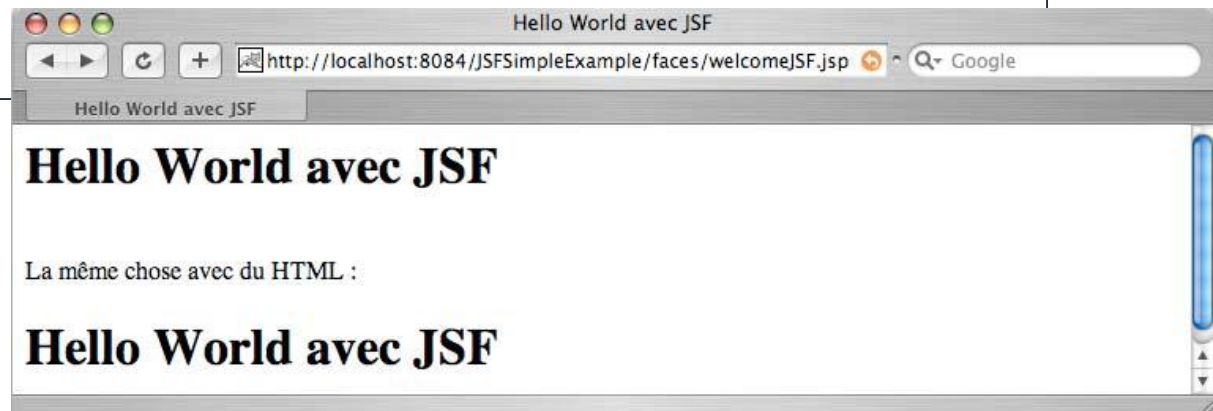
- Selon la version et le serveur d'application, le fichier web.xml doit être complété
- Pour la version 1.2 de JSF (Sun)

```
<web-app ...>
  ...
  <listener>
    <listener-class>
      com.sun.faces.config.ConfigureListener
    </listener-class>
  </listener>
  <listener>
    <listener-class>
      com.sun.faces.application.WebappLifecycleListener
    </listener-class>
  </listener>
</web-app>
```

# Le premier exemple « Hello World avec JSF »

- Exemple : afficher le message « Hello World » avec JSF
  - welcomeJSF.xhtml du projet **HelloWorld**

```
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:core="http://java.sun.com/jsf/core"
xmlns:html="http://java.sun.com/jsf/html"
xmlns:ui="http://java.sun.com/jsf/facelets">
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Hello World avec JSF</title>
  </head>
  <body>
    <core:view>
      <h1><html:outputText value="Hello World avec JSF" /></h1><br>
      La même chose avec du HTML : <h1>Hello World avec JSF</h1>
    </core:view>
  </body>
</html>
```



# Le premier exemple « Hello World avec JSF »

- web.xml du projet HelloWorld

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/webapp_2_4.xsd">
  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>/faces/*</url-pattern>
  </servlet-mapping>
  <security-constraint>
    <display-name>Constraint1</display-name>
    <web-resource-collection>
      <web-resource-name>No-JSP-JSF-Page</web-resource-name>
      <url-pattern>/welcomeJSF.jsp</url-pattern>
    </web-resource-collection>
    <auth-constraint>
      <description>Pas de rôles, donc pas d'accès direct</description>
    </auth-constraint>
  </security-constraint>
</web-app>
```

# Le premier exemple « Hello World avec JSF »

- Considérons que la page welcomeJSF.jsp n'est pas protégée et que la ressource est accédée directement (sans passer par la Servlet principale)



- Retour d'un message d'erreur car les éléments JSF ne sont pas traités par la Servlet principale

# Le premier exemple « Hello World avec JSF »

- faces-config.xml du projet HelloWorld

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE faces-config PUBLIC
"-//Sun Microsystems, Inc.//DTD JavaServer Faces Config
1.1//EN"
"http://java.sun.com/dtd/web-facesconfig_1_1.dtd">
<faces-config>
</faces-config>
```

- Dans cet exemple aucune configuration est nécessaire. Nous verrons dans la suite les règles de navigation et la déclaration de Bean managés



# Bean Managé : principe

- Rappelons qu'un Bean est une classe Java respectant un ensemble de directives
    - Un constructeur public sans argument
    - Les propriétés d'un Bean sont accessibles au travers de méthodes getXXX et setXXX portant le nom de la propriété
  - L'utilisation des Beans dans JSF permet
    - l'affichage des données provenant de la couche métier
    - le stockage des valeurs d'un formulaire
    - la validation des valeurs
    - l'émission de messages pour la navigation (reçus par faces-config.xml)
- Dans la suite nous verrons que les Beans peuvent retourner des messages exploitables par le contrôleur JSF : principe de la Navigation

# Bean Managé : principe

- Un Bean managé est un Bean dont la vie est gérée par JSF, il est soit déclaré dans le facesconfig soit annoté pour définir :
  - Définir la durée de vie (scope) d'un Bean
  - Initialiser les propriétés d'un Bean

# Bean managé : configuration dans faces-config.xml

- Pour créer un Bean managé on peut le déclarer dans le fichier de configuration de JSF à l'aide de la balise <managed-bean>
- Trois éléments essentiels sont à préciser
  - <managed-bean-name> définit un nom qui servira d'étiquette quand le Bean sera exploité dans les pages JSP
  - <managed-bean-class> déclare le nom de la classe de type package.class
  - <managed-bean-scope> précise le type de scope utilisé pour le Bean
    - none, application, session, request

```
...  
<navigation-rule>...</navigation-rule>  
<managed-bean>  
  <managed-bean-name>MyBean</managed-bean-name>  
  <managed-bean-class>mypackage.MyFirstBean</managed-bean-class>  
  <managed-bean-scope>session</managed-bean-scope>  
</managed-bean>  
...
```

# Bean managé : configuration par Annotations

- Pour créer un Bean managé on peut utiliser l'annotation `@ManagedBean` en déclarant dans l'attribut `name` son nom.
- On utilise une autre annotation pour déclarer le scope dans lequel il sera utilisé :
  - `@RequestScoped`
  - `@SessionScoped`
  - ....

```
@ManagedBean(name = "helloWorld")
@RequestScoped
public class HelloWorld {
```


# Bean managé : JSP, Bean et Expression Language (EL)

- Un formulaire JSF doit être construit dans un groupe défini par la balise `<html:form> ... </html:form>`
    - ACTION est automatiquement à SELF (URL courante)
    - METHOD est obligatoirement POST
  - Utilisation de composants JSF pour saisir des informations
    - `<html:inputText>` pour la balise HTML `<INPUT TYPE="Text">`
    - `<html:inputSecret>` pour la balise `<INPUT TYPE="PASSWORD">`
    - `<html:commandButton>` pour la balise `<INPUT TYPE="SUBMIT">`
  - La balise `<html:commandButton>` contient un attribut action qui permet d'indiquer un message traité par les règles de navigation définies dans `faces-config.xml`
- Nous verrons dans la partie Navigation le traitement de la valeur indiquée dans l'attribut action

# Bean managé : JSP, Bean et Expression Language (EL)

- Exemple : formulaire JSP utilisant des composants JSF  
beanform1.xhtml du projet **ManagedBean**

```
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:core="http://java.sun.com/jsf/core"
xmlns:html="http://java.sun.com/jsf/html"
xmlns:ui="http://java.sun.com/jsf/facelets">
<head>
<title>Premier formulaire avec JSF</title>
</head>
<body>
  <core:view>
    <html:form>
      <html:outputText value="Adresse Email " /><html:inputText/><br>
      <html:outputText value="Mot de Passe " /><html:inputSecret/><br>
      <html:commandButton value="Connecter" />
    </html:form>
  </core:view>
</body>
</html>
```



The screenshot shows a web browser window with the title "Premier formulaire avec JSF". The address bar shows the URL "http://localhost:8084/JSFCoursesExa". The page content displays a form with two input fields: "Adresse Email" and "Mot de Passe". Below the fields is a button labeled "Connecter".

# Bean managé : JSP, Bean et Expression Language (EL)

- Les Expressions Languages (EL) sont utilisées pour accéder aux éléments du Bean dans les pages JSP
- Un EL permet d'accéder simplement aux Beans des différents scopes de l'application (page, request, session et application)
- Forme d'un Expression Language JSF : `#{expression}`
- Les EL JSF sont différentes des EL JSP qui utilisent la notation : `${expression}`
- Une EL est une expression dont le résultat est évaluée au moment où JSF effectue le rendu de la page

# Bean managé : JSP, Bean et Expression Language (EL)

- L'écriture `{MyBean.value}` indique à JSF
  - de chercher un objet qui porte le nom de MyBean dans son contexte
  - puis invoque la méthode `getValue()` (chercher la propriété value)
- Le nom de l'objet est celui d'un Bean managé
- Possibilité d'accéder à un objet contenu dans le Bean
  - `{MyBean.myobject.value}` : propriété value de myobject contenu dans MyBean



# Bean managé : JSP, Bean et Expression Language (EL)

- JSF définit un ensemble d'objets implicites utilisables dans les Expressions Languages :
  - param : éléments définis dans les paramètres de la requête HTTP
  - param-values : les valeurs des éléments param
  - cookies : éléments définis dans les cookies
  - initParam : paramètres d'initialisation de l'application
  - requestScope : éléments défini dans la requête
  - facesContext : instance de la classe FacesContext
  - View : instance de UIViewRoot

Expliqué au niveau des composants graphiques

Etudiée au niveau de la partie cycle de vie

# Bean managé : JSP, Bean et Expression Language (EL)

- Exemple : objets implicites JSF dans une JSP

```
<html>
  <...>
  <body>
    <core:view>
      <p><html:outputText
        value="#{param.test}" /></p>
      <p><html:outputText
        value="#{cookie.JSESSIONID.value}" /></p>
      <p><html:outputText
        value="#{facesContext.externalContext.requestPathInfo}" /></p>
      <p><html:outputText
        value="#{facesContext.externalContext.requestServletPath}" /></p>
    </core:view>
  </body>
</html>
```

Utilisation des objets impli...

CCD5180840025B229F22EC3B0C235CAC

/converters1.jsp

/faces

# Bean managé : modifier/afficher la propriété d'un Bean

- Un formulaire regroupe un ensemble de composants contenant chacun une valeur (attribut value)
- Chaque valeur d'un composant peut être stockée dans une propriété du Bean
- Pour définir une propriété dans un Bean
  - Créer l'attribut correspondant à la propriété
  - Définir des méthodes set et get pour accéder en lecture et en écriture aux propriétés
- Pour stocker la valeur d'un composant vers le Bean ou afficher la valeur d'une propriété dans un composant : utilisation des EL dans l'attribut value du composant JSF
- JSF associe automatiquement la propriété d'un Bean à la valeur d'un composant

# Bean managé : modifier/afficher la propriété d'un Bean

- Exemple : gestion d'un formulaire « email et mot de passe »

```
package beanPackage;
public class RegistrationBean {

    private String email = "user@host";

    private String password = "";

    public String getEmail() {
        return email;
    }
    public void setEmail(String t)
        this.email = t;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String t) {
        this.password = t;
    }
}
```

RegistrationBean.java du projet **ManagedBean** possède deux propriétés et des modifieurs et accesseurs

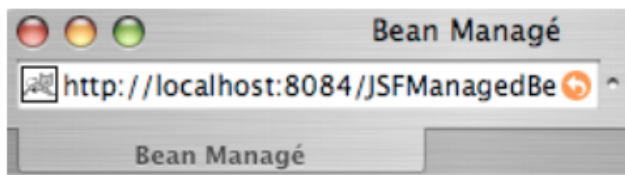
# Bean managé : modifier/afficher la propriété d'un Bean

- Exemple (suite) : gestion d'un formulaire ...

```
...  
<core:view>  
  <html:form>  
    <html:outputText value="Adresse Email "/>  
    <html:inputText value="#{registrationbean.email}"/><br>  
  
    <html:outputText value="Mot de Passe "/>  
    <html:inputSecret value="#{registrationbean.password}"/><br>  
  
    <html:commandButton value="Connecter" />  
  </html:form>  
</core:view>  
</body>  
</html>
```

EL pour accéder à la propriété *email*

EL pour accéder à la propriété *password*



*beanform2.jsp* du projet  
**ManagedBean**

Adresse Email :

Mot de Passe :

Affichage de la valeur  
d'initialisation du Bean

# Bean managé : modifier/afficher la propriété d'un Bean

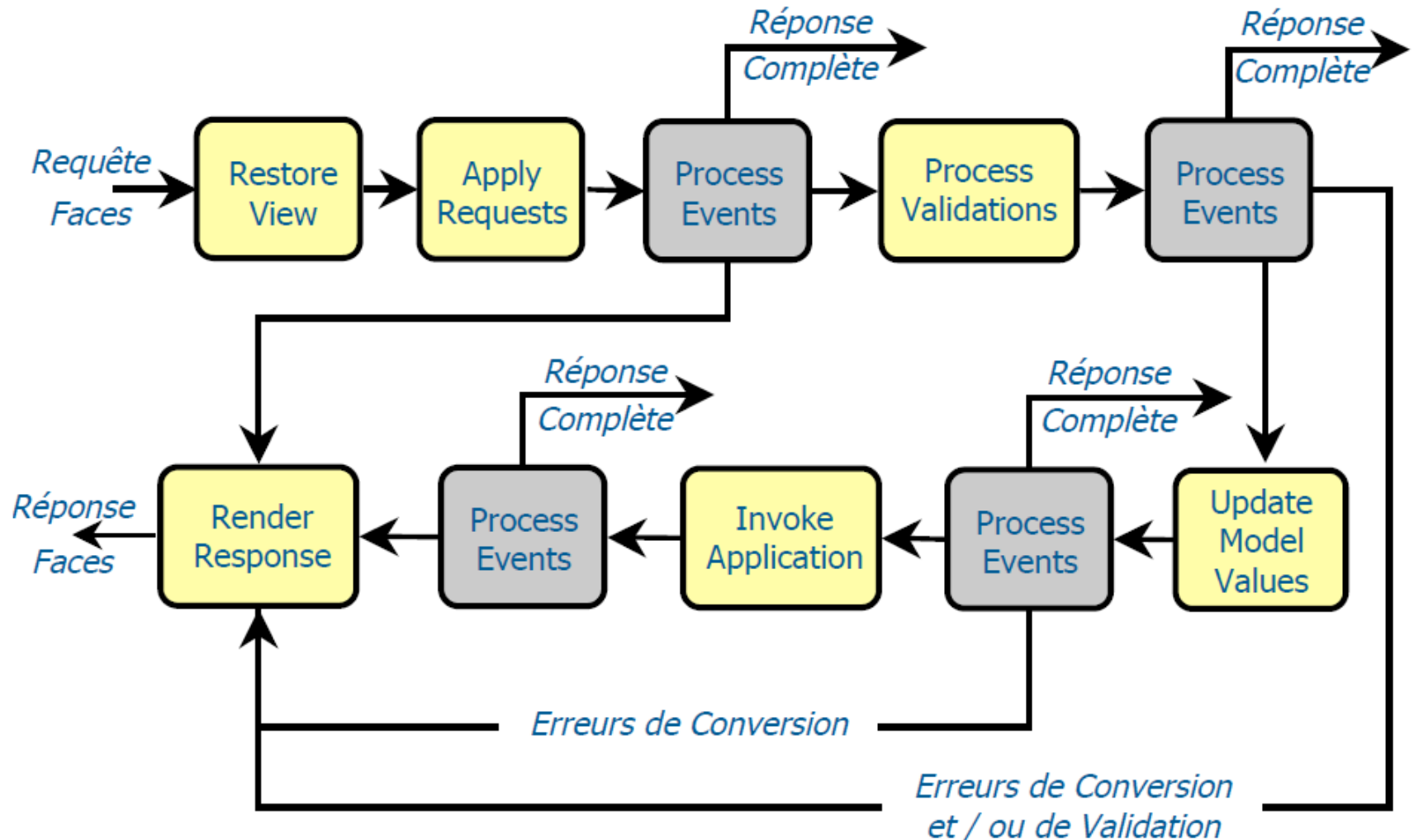
- Exemple (suite) : gestion d'un formulaire ...

```
<faces-config>
  <managed-bean>
    <managed-bean-name>
      registrationbean
    </managed-bean-name>
    <managed-bean-class>
      beanpackage.RegistrationBean
    </managed-bean-class>
    <managed-bean-scope>
      request
    </managed-bean-scope>
  </managed-bean>
</faces-config>
```

Identifiant utilisé pour accéder  
au Bean défini par la classe  
*beanpackage.RegistrationBean*

*faces-config.xml* du projet  
**ManagedBean**

# Cycle de vie d'une JSF : généralités



# Cycle de vie d'une JSF : généralités

- 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



# Navigation : configuration de faces-config.xml

- Le fichier de faces-config.xml joue le rôle de contrôleur, il décide de la ressource qui doit être appelée suite à la réception d'un message
- Les messages sont des simples chaînes de caractères
- Utilisation de la balise <navigation-rule> pour paramétrer les règles de navigation
- La balise <from-view-id> indique la vue source où est effectuée la demande de redirection. La vue peut être un :
  - Formulaire (action de soumission)
  - Lien hypertext
- Pour chaque valeur de message une page vue de direction est indiquée dans la balise <navigation-case>
  - <from-outcome> : la valeur du message
  - <to-view-id> : la vue de direction

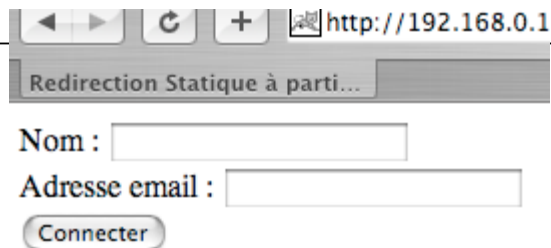
# Navigation : statique ou dynamique

- JSF implémente une machine à états pour gérer la navigation entre des pages JSF
- Pour schématiser nous distinguons deux sortes de navigation
  - Navigation statique
  - Navigation dynamique
- **Navigation statique**
  - La valeur de l'outcome est connue au moment de l'écriture de la JSP
- **Navigation dynamique**
  - La valeur de l'outcome est inconnue au moment de l'écriture de la JSP
  - Elle peut être calculée par un Bean Managé ou autre chose ...
- Remarques
  - Si une valeur d'outcome est inconnue la même page JSP est rechargée
  - Si la valeur vaut null la page JSP est rechargée

# Navigation statique : exemples

- Exemple : redirection statique à partir d'un formulaire

```
...  
<html>  
  <head>  
    <title>Redirection Statique à partir d'un formulaire</title>  
  </head>  
  <body>  
    <core:view>  
      <html:form>  
        <html:outputText value="Nom : " />  
        <html:inputText value="#{beancontroller1.name}" /><br>  
        <html:outputText value="Adresse email : " />  
        <html:inputText value="#{beancontroller1.email}" /><br>  
        <html:commandButton value="Connecter" action="register"/>  
      </html:form>  
    </core:view>  
  </body>  
</html>
```



Redirection Statique à parti...

Nom :

Adresse email :

*form1.jsp* du projet  
**Navigation**

# Navigation statique : exemples

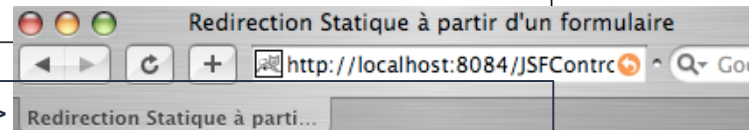
- Exemple (suite) : redirection statique ...

```
...  
<faces-config>  
  <navigation-rule>  
    <from-view-id>/form1.jsp</from-view-id>  
    <navigation-case>  
      <from-outcome>register</from-outcome>  
      <to-view-id>/accepted.jsp</to-view-id>  
    </navigation-case>  
  </navigation-rule>  
</faces-config>
```

*faces-config.xml*

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

```
  ...  
  <body>  
    <core:view>  
      <h1><html:outputText value="Redirection Réussie" /></h1>  
    </core:view>  
  </body>  
</html>
```



**Redirection Réussie**

# Navigation dynamique : exemples

- Exemple : redirection dynamique à partir d'un Bean

```
package beanPackage;

public class BeanController2 {
    private String email = "user@host";
    private String name = "";

    public String loginConnect() {
        if (this.email.isEmpty()) {
            return "Rejected";
        }
        if (this.name.isEmpty()) {
            return "Rejected";
        }
        return "Accepted";
    }
}
```

# Navigation dynamique : exemples

- Exemple (suite) : redirection dynamique à partir d'un Bean

```
<faces-config>
<navigation-rule>
  <from-view-id>form2.xhtml</from-view-id>
  <navigation-case>
    <from-outcome>Accepted</from-outcome>
    <to-view-id>/accepted.xhtml</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>Rejected</from-outcome>
    <to-view-id>/rejected.xhtml</to-view-id>
  </navigation-case>
</navigation-rule>
<managed-bean>
  <managed-bean-name>beancontroller2</managed-bean-name>
  <managed-bean-class>
    beanPackage.BeanController2
  </managed-bean-class>
  ...
</faces-config>
```

# Navigation dynamique : exemples

- Exemple (suite) : redirection dynamique à partir d'un Bean

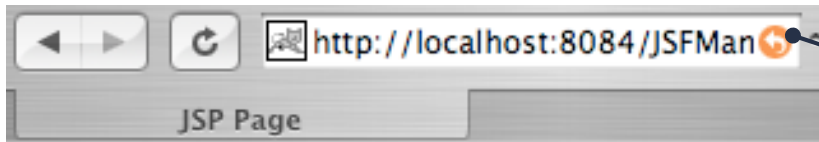
```
...  
<html>  
  <head>  
    <title>Redirection Dynamique à partir d'un formulaire</title>  
  </head>  
  <body>  
    <core:view>  
      <html:form>  
        <html:outputText value="Nom : " />  
        <html:inputText value="#{beancontroller2.name}" />  
        <br>  
        <html:outputText value="Adresse email : " />  
        <html:inputText value="#{beancontroller2.email}" />  
        <br>  
        <html:commandButton value="Connecter"  
          action="#{beancontroller2.LoginConnect}" />  
      </html:form>  
    </core:view>  
  </body>  
</html>
```

Accès au Bean identifié  
par  
beancontroller2 et à la  
méthode loginConnect

# Navigation dynamique : exemples

– Exemple (suite) : redirection dynamique à partir d'un Bean

```
<%@taglib uri="http://java.sun.com/jsf/core" prefix="core"%>
<%@taglib uri="http://java.sun.com/jsf/html" prefix="html"%>
<html>
    <head>
        <title>Résultat du traitement du formulaire</title>
    </head>
    <body>
        <core:view>
            <h1>
                <html:outputText value="Connexion de : " />
                <html:outputText value="#{beancontroller2.name}" />
            </h1>
        </core:view>
    </body>
</html>
```



Page affichée à la suite de la soumission. Lecture de la propriété email du Bean identifié par beancontroller2



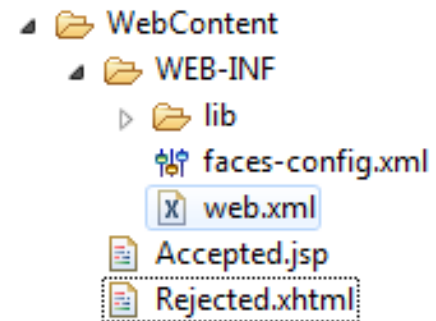
# Navigation en JSF 2.\*

- A partir de la version 2.0 JSF simplifie la gestion de la navigation en permettant de se passer du faces-config.
- Si le texte reçu dans l'attribut action (statiquement ou dynamiquement) n'est pas trouvé dans le faces-config JSF recherchera dans WebContent un fichier d'affichage (jsp, xhtml,...) nommé de cette façon.

```
package beanPackage;

public class BeanController2 {
    private String email = "user@host";
    private String name = "";

    public String loginConnect() {
        if (this.email.isEmpty()) {
            return "Rejected";
        }
        if (this.name.isEmpty()) {
            return "Rejected";
        }
        return "Accepted";
    }
}
```



# Composants graphiques : présentation

- JSF fournit un ensemble de composants graphiques pour la conception de l'IHM
- Un composant JSF est développé à partir de :
  - classes qui codent le comportement et l'état
  - classes de « rendu » qui traduisent la représentation graphique (HTML, FLASH, XUL, ...)
  - classes qui définissent les balises personnalisées relation entre JSP et classes Java
  - classes qui modélisent la gestion des événements
  - classes qui prennent en compte les Converters et les Validators
- Pour exploiter les composants JSF dans les pages JSP, des balises personnalisées sont utilisées ...

# Composants graphiques : balises CORE

- Les balises personnalisées décrites dans CORE s'occupent d'ajouter des fonctionnalités aux composants JSF
- Pour rappel pour avoir un descriptif de l'ensemble des balises CORE :
  - <http://java.sun.com/javaee/javaxserverfaces/1.2/docs/tlddocs>
- La bibliothèque contient un ensemble de balises
  - facet : déclare un élément spécifique pour un composant
  - view et subview : ajoute une vue ou une sous vue
  - attribute et param : ajoute un attribut ou paramètre à un composant
  - selectionitem et selectionitems : définit un ou plusieurs éléments
  - convertDateTime et convertNumber : conversion de données
  - validator, validateDoubleRange, ... : ajoute des validators
  - actionListener, valueChangeListener : différents écouteurs
  - loadBundle : chargement de bundle (fichier de ressources ...)

# Composants graphiques : balises HTML

## ■ Famille des regroupements

- form
- panelGroup
- panelGrid

## ■ Famille des saisis

- inputHidden
- inputSecret
- inputText
- inputTextArea
- selectBooleanCheckbox
- selectManyCheckbox
- selectManyListbox
- selectManyMenu

- selectOneListbox
- selectOneMenu
- selectOneRadio

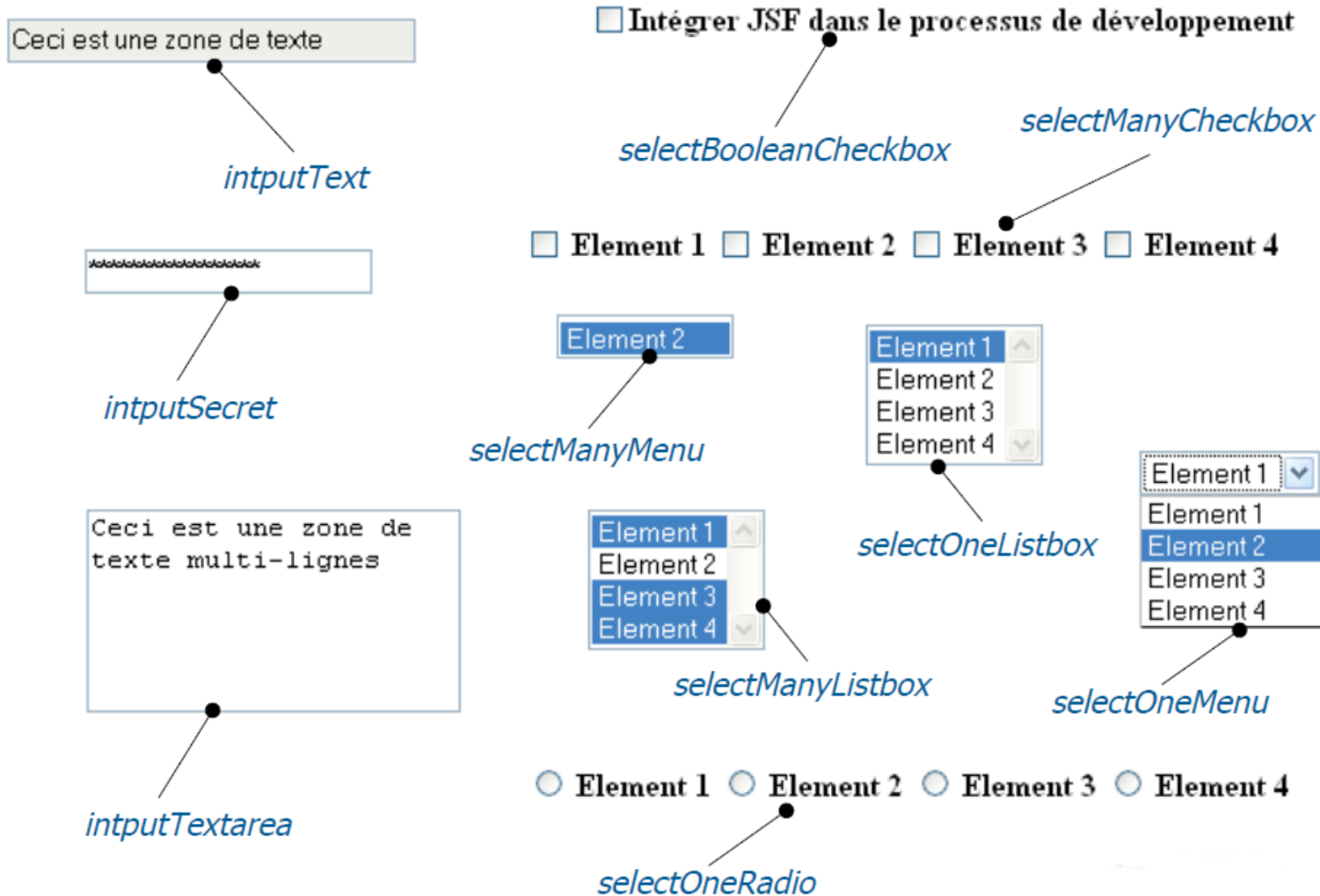
## • Famille des sorties

- column
- message et messages
- dataTable
- outputText
- outputFormat
- outputLink
- graphicImage

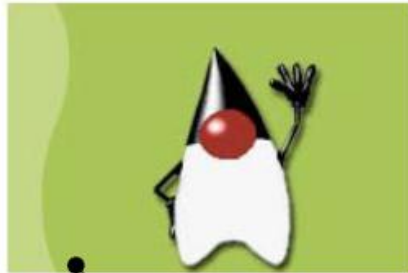
## • Famille des commandes

- commandButton
- commandLink

# Composants graphiques : balises HTML saisies



# Composants graphiques : balises HTML sorties et commandes



*graphicImage*

Commande Bouton

*commandButton*

Commande Lien

*commandLink*

*outputText*

Du texte affiché avec la balise `outputText`

Google

*outputLink*

Nom	Prenom	Date de naissance	Poids
Baron	Mickael	17081976	Ingénieur d'étude et de développement
Dupont	Marcel	21041956	Boucher

*dataTable*

Nom :	<input type="text" value="Baron"/>
Mot de passe :	<input type="password" value="kkkkkkkk"/>

*panelGrid*

# Composants graphiques : détail balises HTML

- Une balise contient généralement trois types d'attributs
  - Des attributs basiques (description)
  - Des attributs HTML (description de présentation)
  - Des attributs DHTML (description pour la partie dynamique)
- Selon la nature du composant graphique (saisie, sortie, commande et autre) le nombre d'attributs varie ...

# Composants graphiques : détail balises HTML

- Les attributs basiques sont utilisés pour ajouter une description spécifique à JSF
- Attributs basiques
  - id : identifiant du composant
  - binding : association avec un Bean « Backing »
  - rendered : true composant affiché, false composant caché
  - styleClass : précise le nom d'une CSS à appliquer au composant
  - value : valeur du composant
  - valueChangeListener : associer à une méthode pour le changement de valeur
  - converter : nom de la classe pour une conversion de données
  - validator : nom de la classe pour une validation de données
  - required : true valeur obligatoire, false valeur optionnelle



# Composants graphiques : détail balises HTML

- Les attributs HTML sont utilisés pour modifier l'apparence graphique d'un composant JSF
- Attributs HTML
  - alt : texte alternatif au cas où le composant ne s'affiche pas
  - border : bordure du composant
  - disabled : désactive un composant de saisie ou un bouton
  - maxlength : maximum de caractères pour un composant de texte
  - readonly : mode lecture uniquement
  - size : taille d'un champ de texte
  - style : information du style
  - target : nom de la frame dans lequel le document est ouvert
  - ...

# Composants graphiques : détail balises HTML

- Les attributs DHTML sont utilisés pour ajouter un aspect dynamique pour les composants JSF (appel à du JavaScript)
- Attributs DHTML
  - onblur : élément perd le focus
  - onclick : clique souris sur un élément
  - ondblClick : double clique souris sur un élément
  - onfocus : élément gagne le focus
  - onkeydown, onkeyup : touche clavier enfoncée et relâchée
  - onkeypress : touche clavier pressée puis relâchée
  - onmousedown, onmouseup : bouton souris enfoncé et relâché
  - onmouseout, onmouseover : curseur souris sort et entre
  - onreset : formulaire est initialisé
  - onselect : texte est sélectionné dans un champ de texte
  - onsubmit : formulaire soumis

# Composants graphiques : <html:panelGrid>

- La balise `html:panelGrid` est utilisée pour définir l'agencement des composants visuels
- Il ne s'agit donc pas d'un composant visuel mais d'un conteneur dont l'organisation de ces composants enfants est réalisée suivant une grille
- Les composants enfants sont ajoutés dans le corps
- Principaux attributs :
  - `columns` : nombre de colonnes de la grille
  - `bgcolor` : couleur du fond
  - `cellpadding`, `cellspacing` : espacement entre les cellules
  - `border` : border autour de la grille
- Le composant `html:panelGroup` permet de regrouper des composants dans une cellule (une sorte de fusion)

# Navigation dynamique : exemples

```
...  
<h4>  
  <html:panelGrid cellspacing="25" columns="2">  
    <html:outputText value="Nom : " />  
    <html:panelGroup>  
      <html:inputText size="15" required="true" />  
      <html:inputText />  
    </html:panelGroup>  
    <html:outputText value="Mot de passe : " />  
    <html:inputSecret />  
  </html:panelGrid>  
</h4>
```

Création d'une  
grille avec  
deux  
colonnes

Regroupement  
de  
deux composants  
dans une cellule

Nom :

Mot de passe :

# Composants graphiques : <html:dataTable>

- Le composant <html:dataTable> permet de visualiser des données sur plusieurs colonnes et plusieurs lignes
- La table peut être utilisée quand le nombre d'éléments à afficher est inconnu
- Les données (la partie modèle) peuvent être gérées par des Beans
- Attributs de la balise :
  - value : une collection de données (Array, List, ResultSet, ...)
  - var : nom donné à la variable à manipuler pour chaque ligne
  - border, bgcolor, width : attributs pour l'affichage
  - rowClasses, headerClass : attributs pour la gestion des styles (CSS)
- Pour chaque colonne de la table, la valeur à afficher est obtenue par la balise <html:column>

# Composants graphiques : <html:dataTable>

- Exemple : la représentation d'une table JSF

```
public class Personne {  
    private String name;  
    private String firstName;  
    private String birthName;  
    private String job;  
    public Personne(String pN, String pF, String pB, String pJ) {  
        name = pN; firstName = pF; birthName = pB; job = pJ;  
    }  
    public String getName() { return name; }  
    public void setName(String pName) { name = pName; }  
    public String getFirstName() { return firstName; }  
    public void setFirstName(String pFirstName) { firstName = pFirstName; }  
    ...  
}
```

```
public class DataTableBean {  
    private List<Personne> refPersonne;  
    public List getPersonne() {  
        if (refPersonne == null) {  
            refPersonne = new ArrayList<Personne>();  
            refPersonne.add(new Personne("Baron", "Mickael", "17081976",  
"Développeur"));  
            refPersonne.add(new Personne("Dupont", "Marcel", "21041956", "Boucher"));  
        }  
        return refPersonne;  
    }  
}
```

# Composants graphiques : <html:dataTable>

- Exemple (suite) : la représentation d'une table JSF

```
<core:view>
  <html:dataTable value="#{outputbean.personne}"
var="personne" border="1" cellpadding="4" width="60%" >
    <html:column>
        <html:outputText value="#{personne.name}" />
    </html:column> <html:column>
        <html:outputText value="#{personne.firstname}" />
    </html:column> <html:column>
        <html:outputText value="#{personne.birthdata}" />
    </html:column> <html:column>
        <html:outputText value="#{personne.job}" />
    </html:column>
  </html:dataTable>
</core:view>
```

Baron	Mickael	17081976	Ingénieur d'étude et de développement
Dupont	Marcel	21041956	Boucher
Martin	Alexandre	28011946	Magicien
Fox	Tracy	18021969	Sexologue

# Composants graphiques : <html:dataTable>

- La modification des en-tête et pied de page d'une table est obtenue en utilisant la balise <core:facet>
- <core:facet> s'occupe d'effectuer une relation de filiation entre un composant et un autre
- La filiation consiste à associer le composant <core:facet> est un composant défini dans le corps de <core:facet>
- Attribut du composant <core:facet>
  - name : nom de la filiation
- Pour le composant table deux filiations possibles
  - header : une filiation entre une colonne et le nom de la colonne
  - footer : une filiation entre la table et un nom
  - caption : une filiation entre le titre de la table et un nom



# Composants graphiques : <html:dataTable>

- Exemple (suite) : la représentation d'une table JSF

```
<core:view><html:dataTable value="#{outputbean.personne}" var="personne"
border="1" cellspacing="4" width="60%" >
  <html:column>
    <core:facet name="header" >
      <html:outputText value="Nom" />
    </core:facet>
    <html:outputText value="#{personne.name}" />
  </html:column>
  <html:column>
    <core:facet name="header" >
      <html:verbatim>Prénom</verbatim>
    </core:facet>
    <html:outputText value="#{personne.firstname}" />
  </html:column>
  <html:column>
    <core:facet name="header" >
      Date de naissance
    </core:facet>
    <html:outputText value="#{personne.birthdata}" />
  </html:column>
  <html:facet name="footer">
    <html:outputText value="#{outputbean.caption}" />
  </html:facet>
  ...
</html:dataTable></core:view>
```

Nom	Prénom		Emploi
Baron	Mickael	17081976	Ingénieur d'étude et de développement
Dupont	Marcel	21041956	Boucher
Martin	Alexandre	28011946	Magicien
Fox	Tracy	18021969	Sexologue
Une Simple Table			

Ne sera jamais  
affiché car  
<core:facet> n'est  
associé à aucun  
autre composant

# Composants graphiques : <html:dataTable>

- Pour l'instant, seul l'aspect peuplement des informations a été traité, <html:dataTable> offre la possibilité de modifier l'apparence d'une table
- Les attributs :
  - headerClass : style CSS pour la partie en-tête
  - footerClass : style CSS pour le bas de page
  - rowClasses : liste de styles CSS pour les lignes
- L'attribut rowClasses permet de définir une liste de style CSS qui sera appliquée pour chaque ligne et répétée autant de fois qu'il y a de lignes (définition d'un motif)
- Exemple :
  - Soit la définition suivante : rowClasses="row1, row2, row2"
  - Répétition du motif row1, row2, row2, row1, row2, ...

# Composants graphiques : <html:dataTable>

- Exemple : table JSF « maquillé »

```
<head>
<...>
<link href="output.css" rel="stylesheet" type="text/css" >
</head>
<body>
    <core:view>
    <html:dataTable value="#{outputbean.personne}" var="personne" border="1"
        cellspacing="4" width="60%" rowClasses="row1,row2"
        headerClass="heading" footerClass="footer" >
    ...
    </html:dataTable>
</core:view>
```

```
.heading {
font-family: Arial, Helvetica,
sans-serif;
font-weight: bold;
font-size: 20px;
color: black;
background-color:silver;
text-align:center;
}
.row1 {
background-color:#GFGFGF;
}
.row2 {
Background-color:#CECECE;
.footer {
background-color:#000009;
Color:white;
}
```

Nom	Prenom	Date de naissance	Emploi
Baron	Mickael	17081976	Ingénieur d'étude et de développement
Dupont	Marcel	21041956	Boucher
Martin	Alexandre	28011946	Magicien
Fox	Tracy	18021969	Sexologue
Une Simple Table			

# Composants graphiques : <html:dataTable>

- Exemple : table JSF avec des composants JSF de saisie

```
<core:view>
  <html:dataTable value="#{outputbean.personne}" var="personne"
border="1" cellspacing="4" width="60%" rowClasses="..." >
  <html:column>
    <core:facet name="header" >
      <html:ouputText value="Nom" />
    </core:facet>
    <html:inputText value="#{personne.name}" />
  </html:column>
  <html:column>
    <core:facet name="header" >
      <html:verbatim>Prénom</verbatim>
    </core:facet>
    <html:outputText value="#{personne.firstname}" />
  </html:column>
</html:dataTable>
</core:view>
```

Nom	Prenom	Date de naissance	Emploi
<input type="text" value="Baron"/>	Mickael	17081976	Ingénieur d'étude et de développement
<input type="text" value="Dupont"/>	Marcel	21041956	Boucher
<input type="text" value="Martin"/>	Alexandre	28011946	Magicien
<input type="text" value="Fox"/>	Tracy	18021969	Sexologue

Une Simple Table

# Fin cours 1

# Composants graphiques : items

- Les balises HTML de type sélection (selectOneRadio, selectOneMenu) permettent de gérer un ensemble d'éléments
- Les éléments peuvent être « peuplés » par :
  - core:selectItem : affiche un seul élément
  - core:selectItems : affiche plusieurs éléments
- Les balises HTML de sélection s'occupent de gérer la sélection courante (une ou plusieurs) au travers de leur attribut value
- Le type de la sélection courante (retourné au serveur) est donné par les éléments de sélection (int, String, Object, ...)

# Composants graphiques : <core:selectItem>

- La balise core:selectItem est utilisée pour spécifier un seul élément
- Principaux attributs de la balise core:selectItem :
  - itemDescription : description (utilisable dans les outils uniquement)
  - itemDisabled : active ou pas l'item
  - value : valeur qui pointe vers un objet SelectItem
  - itemLabel : le texte à afficher
  - itemValue : la valeur retournée au serveur
- Un objet SelectItem est exploitable directement dans un Bean
- Différents constructeurs :
  - SelectItem(Object value) : constructeur avec une valeur à retourner et à afficher
  - SelectItem(Object value, String label) : constructeur avec une valeur à retourner au serveur et une valeur à afficher

# Composants graphiques : <core:selectItem>

- Exemple : choix d'un élément unique

```
...  
<html:outputText value="Fruit préféré avec SelectItem : " />  
<html:selectOneRadio layout="pageDirection"  
value="#{inputbean.oneRadioValue}" />  
  <core:selectItem value="#{inputbean.bananeItem}" />  
  <core:selectItem value="#{inputbean.orangeItem}" />  
  <core:selectItem itemValue="Clémentine" />  
  <core:selectItem itemValue="Pomme" />  
  <core:selectItem itemValue="Framboise" />  
</html:selectOneRadio>  
...
```

Utilisation d'un  
binding  
d'un composant  
SelectItem

Fruit préféré avec SelectItem :

- ☐ Banane
- ☒ Orange
- ☐ Clémentine
- ☐ Pomme
- ☐ Framboise

Retourne un objet  
SelectItem

```
public class InputBean {  
    private String oneRadioValue;  
    public String getOneRadioValue() {  
        return oneRadioValue;  
    }  
    public void setOneRadioValue(String p) {  
        oneRadioValue = p;  
    }  
    public SelectItem getBananeItem() {  
        return new SelectItem("Banane");  
    }  
    ...  
}
```



# Composants graphiques :

## <core:selectItems>

- L'utilisation de la balise core:selectItem peut alourdir l'écriture des différents éléments des composants de sélection
- Cette balise est plutôt adaptée pour un nombre réduits d'éléments
- La balise core:selectItems allège donc l'écriture puisqu'une seule occurrence de core:selectItems remplace toutes les occurrences de core:selectItem
- L'attribut value est une valeur binding qui pointe vers une structure de type SelectItem
- La structure de type SelectItem peut être de type
  - instance unique
  - map : les entrées sont les itemLabels et itemValues de SelectItem
  - collection
  - tableau

# Composants graphiques : <core:selectItems>

- Exemple : choisir de nombreux éléments

Fruit préféré avec SelectItems :

- ☐ Banane
- ☐ Orange
- ☒ Clémentine *html:selectOneRadio*
- ☐ Pomme
- ☐ Franboise

Ajouter des périphériques :

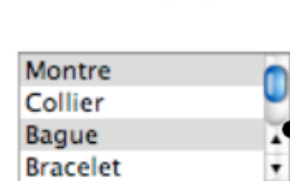
- ☒ Clavier *html:selectManyCheckbox*
- ☐ Souris
- ☒ Ecran
- ☒ Unité centrale
- ☐ Haut parleur

Accessoire préféré :



*html:selectOneListbox*

Choisir équipements préférés :



*html:selectManyListbox*

Choisir une marque de voiture :



*html:selectOneMenu*

Choisir plusieurs marques de voiture :



*html:selectManyMenu*

# Composants graphiques : <core:selectItem>

- Exemple (suite) : choisir de nombreux éléments

```
<h4>
  <html:outputText value="Fruit préféré avec SelectItems : " />
  <html:selectOneRadio value="#{inputbean.oneRadioValue}">
    <code:selectItems value="#{inputbean.oneRadioItems}" />
  </html:selectOneRadio>
</h4>

<h4>
  <html:outputText value="Ajouter des périphériques : " />
  <html:selectManyCheckBox value="#{inputbean.manyCheckBoxValues}">
    <code:selectItems value="#{inputbean.manyCheckBoxItems}" />
  </html:selectManyCheckBox >
</h4>

<h4>
  <html:outputText value="Accessoire préféré : " />
  <html:selectOneListBox value="#{inputbean.oneListBoxValues}" >
    <code:selectItems value="#{inputbean.manyAndOneListBoxItems}" />
  </html:selectOneListBox>
</h4>
```

Simplification  
d'écriture avec  
la balise  
selectItems

Principe  
identique  
quelque soit le  
composant de  
sélection

# Composants graphiques : <core:selectItem>

- Exemple (suite) : choisir de nombreux éléments

```
<h4>
  <html:outputText value="Choisir équipements préférés : " />
  <html:selectManyListBox value="#{inputbean.manyListBoxValues}" />
    <code:selectItems value="#{inputbean.manyAndOneListBoxItems}" />
  </html:selectManyListBox>
</h4>

<h4>
  <html:outputText value="Choisir une marque de voiture : " />
  <html:selectOneMenu value="#{inputbean.oneMenuValue}" />
    <code:selectItems value="#{inputbean.manyAndOneMenuItems}" />
  </html:selectOneMenu>
</h4>

<h4>
  <html:outputText value="Choisir plusieurs marques de voiture : " />
  <html:selectManyMenu value="#{inputbean.manyMenuValue}" />
    <code:selectItems value="#{inputbean.manyAndOneMenuItems}" />
  </html:selectManyMenu>
</h4>
```

Une seule  
valeur  
sélectionnable

Une seule  
valeur  
sélectionnable

# Composants graphiques : <core:selectItem>

- Exemple (suite) : choisir de nombreux éléments

```
public class InputBean {  
    private SelectItem[] manyCheckBoxItems = new SelectItem[] {  
        new SelectItem("Clavier"),  
        new SelectItem("Souris"),  
        new SelectItem("Ecran"),  
        new SelectItem("Unité Centrale"),  
        new SelectItem("Haut Parleur")  
    };  
  
    private String[] manyCheckBoxValues;  
  
    public SelectItem[] getManyCheckBoxItems() {  
        return manyCheckBoxItems;  
    }  
  
    public String[] getManyCheckBoxValues() {  
        return this.manyCheckBoxValues;  
    }  
  
    public void setManyCheckBoxValues(String[] p) {  
        manyCheckBoxValues = p;  
    }  
}
```

Utilisées pour peupler les éléments du composant selectManyCheckBox

Utilisée pour afficher la valeur sélectionnée courante

Utilisée pour modifier la valeur sélectionnée courante

# <core:ajax>

- Permet d'ajouter des fonctionnalités « Ajax » au composant JSF dans lequel il est inclus
- Par exemple, un bouton de soumission de formulaire peut envoyer une requête et la réponse du serveur mettra à jour une partie de la page sur laquelle l'utilisateur a continué à travailler

# <core:ajax> : les attributs

- event : événement déclencheur de la requête
- execute : composants pris en compte pour l'envoi de la requête (espace séparateur)
- render : composants mis à jour au moment de la réponse du serveur (espace séparateur)

# <core:ajax> : Exemples

## Exemple 1

```
<h:form>
    <h:input value= />
        <h:commandButton>
            <f:ajax event="click" execute="@form"
                render="out1 out2" />
        </h:commandButton>
    <h:input id="out1" .../>
    <h:input id="out2" .../>
</h:form>
```

## Exemple 2

```
<h:form>
    <h:input value="#{bean.***}" />
    <h:selectOneMenu value="#{bean.selected}">
        <f:selectItems value="#{bean.items}" />
        <f:ajax event="change" execute="@form"
            render="@form" />
    </h:selectOneMenu>
    ...
</h:form>
```



# <core:ajax> : les attributs

- execute et render doivent indiquer un ou plusieurs identificateurs de composants
- Quelques valeurs particulières : @all, @this (le composant qui englobe <f:ajax> ; défaut pour execute), @form (tous les composants du formulaire), @none (défaut pour render) ; on peut aussi donner une expression EL qui fournit une collection de String (qui désignent des identificateurs) ; expression évaluée durant le rendu de la page

# Conversion de données : principe

- Conversion des données du modèle et de la vue
  - les paramètres d'une requête sont de type chaînes de caractères
  - les Beans sont définis par des attributs représentés par des types de données (int, long, Object, ...)
  - la conversion de données s'assurent alors qu'une chaîne de caractères peut être traduit en un type de donnée précis
- Respect du formatage en entrée et en sortie
  - Formatage en sortie : affichage d'une date
    - Afficher la date suivant un format compréhensible par l'utilisateur (fonction de la localisation et du type d'utilisateur)
  - Conformité en entrée : saisie d'une date
    - Associé à un composant de type saisie
    - Assurer que l'utilisateur a saisi une chaîne de texte respectant le format de l'entrée

# Conversion de données : principe

- Les convertisseurs de données sont appelés **Converters**
- Tous les composants affichant une valeur et utilisés lors de la saisie fournissent une conversion de données
- Les converters sont utilisables dans une JSP via une balise
- Pour utiliser un converter :
  - soit utiliser les standards fournis par la bibliothèque JSF dans une JSP
  - soit le définir programmatiquement (dans un Bean) et l'utiliser dans une JSP
- Converters standards de la bibliothèque JSF :
  - `core:convertDateTime` : pour la conversion de date et d'heure
  - `core:convertNumber` : pour la conversion de nombre

# Conversion de données : cycle de vie (via l'API)

- Tous composants qui implémentent l'interface ValueHolder peuvent exploiter des Converters (UIOutput et UIInput)
- L'implémentation de l'interface ValueHolder doit fournir des propriétés pour :
  - Effectuer la conversion via un objet Converter
  - Stocker la valeur convertie par l'objet Converter
  - Stocker la valeur propre du composant utilisée pour modifier le modèle de la vue et fournir une donnée à la réponse

# Conversion de données : cycle de vie (via l'API)

- Un objet Converter fournit deux méthodes :
  - Object getAsObject(...) throws ConverterException appelée lors de l'étape de validation du formulaire
  - String getAsString(...) throws ConverterException appelée lors de l'affichage du formulaire
- Dans le cas des objets UIInput les valeurs des paramètres sont stockées dans l'attribut submittedValue (interface EditableValueHolder)

# Conversion de données : cycle de vie (via l'API)

Processus de conversion par l'API

- submittedValue est convertie par getObject(...) et stockée dans localValue (si exception de getObject(...) arrêt du processus)
- Si localValue != null, sa valeur est transmise à value
  - value transmet la valeur à une propriété du modèle de la vue (via un modifieur du Bean)
  - value récupère la valeur d'une propriété du modèle de la vue (via un accesseur du Bean)
- Si localValue == null, value récupère la valeur d'une propriété du modèle
- value est convertie par getString(...) et envoyée à la réponse (si exception de getString(...) arrêt du processus)

# Conversion de données : tag convertNumber

- La balise core:convertNumber permet la conversion de chaînes de caractères en valeur numérique
- À utiliser dans le corps d'une balise graphique JSF
- Différents attributs disponibles
  - type : type de valeur (number, currency ou percent)
  - pattern : motif de formatage
  - (max/min)FractionDigits : (int) nombre maxi/mini sur la décimale
  - (max/min)IntegerDigits : (int) nombre maxi/mini sur la partie entière
  - integerOnly : (booléen) seule la partie entière est prise en compte
  - groupingUsed : (booléen) précise si les caractères de regroupement sont utilisés (exemple : « , », « ; », « : », ... )
  - locale : définit la localisation de la conversion
  - currencyCode : code de la monnaie utilisée pour la conversion
  - currencySymbol : spécifie le caractère (exclusif avec currencyCode)

# Conversion de données : tag convertNumber

- Exemple : utilisation d'un converter en « sortie »

```
<html>
    ...
<body>
<core:view>
    <html:form>
        <html:outputText value="#{beanconverter1.price}">
            <core:convertNumber type="currency" currencyCode="EUR"
                minFractionDigits="4" />
        </html:outputText>
    </html:form>
</core:view>
</body>
</html>
```

Utilisation d'un Converter e...

**50,5000 EUR**

```
public class BeanConverter1 {
    private double price;
    public BeanConverter1() {
        this.price = 50.50d;
    }

    public double getPrice() {
        return price;
    }

    public void setPrice(double pPrice) {
        this.price = pPrice;
    }
}
```



# Conversion de données : tag convertNumber

- Exemple (suite) : utilisation d'un converter en « entrée »

```
<html>
...
<body>

    <core:view>
    <html:form>
    Prix :

        <html:inputText value="#{beanconverter1.price}" >
            <core:convertNumber minFractionDigits="2" />
        </html:inputText><br>
        <html:commandButton value="Envoyer" />

    </html:form>
    </core:view>

</body>
</html>
```

Utilisation d'un Converter e...

**Prix :**

Envoyer

Avant la conversion, au moment de la saisie ...

Après la conversion

Utilisation d'un Converter e...

**Prix :**

Envoyer

# Conversion de données : tag convertDateTime

- La balise core:convertDateTime permet la conversion de chaînes de caractères en date ou heure
- À utiliser également dans le corps d'une balise graphique JSF

```
<html:outputText value="#{packageBean.facturedate}">  
    <core:convertDateTime type="date" dateStyle="Long"/>  
</html:outputText>
```

- Différents attributs disponibles
  - type : type de valeur (date, time ou both)
  - dateStyle : style possible de la date (short, medium, long, full, both)
  - timeStyle : style possible de l'heure (short, medium, long, full, both)
  - Pattern : motif utilisé pour une Date
  - Locale : choix de la localisation

# Conversion de données : tag convertNumber

- Exemple : utilisation d'un converter date en « entrée »

```
<body>
<core:view>
    <html:form>
        <html:outputText value="Nom : " />
        <html:inputText value="#{beanconverter2.name}" /><br>
        <html:outputText value="Nombre de tickets : " />
        <html:inputText value="#{beanconverter2.tickets}" >
            <core:convertNumber type="number" integerOnly="true" />
        </html:inputText><br>

        <html:outputText value="Date : " />
        <html:inputText value="#{beanconverter2.date}">
            <core:convertDateTime type="date" dateStyle="medium" />
        </html:inputText>

        <html:commandButton value="Valider" />
    </html:form>
</core:view>
</body>
```

**Nom :**

**Nombre de tickets :**

**Date :**

# Conversion de données : tag convertNumber

- Exemple (suite) : utilisation d'un converter Date en « entrée »

```
public class BeanConverter2 {  
    private long tickets;  
    private String name;  
    private Date date;  
  
    public BeanConverter2() {  
        tickets = 50;  
        name = "Mon Nom";  
        date = new Date();  
    }  
  
    public long getTickets() { return tickets; }  
    public void setTickets(long pTickets) { tickets = pTickets; }  
    public String getName() { return name; }  
    public void setName(String pName) { name = pName; }  
    public Date getDate() { return date; }  
    public void setDate(Date pDate) { date = pDate; }  
}
```

# Conversion de données : messages

- Dans le cas où des erreurs de conversion sont provoquées, des messages sont ajoutés au FacesContext
- En utilisant les balises `html:message` et `html:messages` possibilité d'afficher un message précis ou l'ensemble
- Généralement à chaque composant est associé un id, ce qui permettra de localiser précisément l'origine de l'erreur
- Exemple :

```
<html:outputText value="Prix : " />
<html:inputText id="priceId" value="#{beanconverter1.price}" >
    <core:convertNumber minFractionDigits="2" />
</html:inputText><html:message for="priceId" /><br>
<html:commandButton value="Envoyer" />
```

# Conversion de données : messages

- Exemple : convertir date en « entrée » avec messages

```
<core:view>
<html:form>
    <html:outputText value="Nom : " />
    <html:inputText value="#{beanconverter2.name}" /><br>
    <html:outputText value="Nombre de tickets : " />
    <html:inputText id="ticketsId" value="#{beanconverter2.tickets}" >
        <core:convertNumber type="number" integerOnly="true" />
    </html:inputText><html:message for="ticketsId" /><br>
    <html:outputText value="Date : " />
    <html:inputText id="dateId" value="#{beanconverter2.date}">
        <core:convertDateTime type="date" dateStyle="medium" />
    </html:inputText><html:message for="dateId" /><br>
    <html:outputText value="Liste des messages : " /><br>
    <html:messages />
    <html:commandButton value="Valider" />
</html:form>
</core:view>
```

Nom :

Nombre de tickets :  "ticketsId": La donnée n'est pas un nombre valide.

Date :  "dateId": Conversion en Date/Heure impossible.

- Problème pour le champs "Nombre de tickets"
- Problème pour le champs "Date"

Valider

# Conversion de données : messages personnalisés

- Les messages sont déclarés par des clés stockées dans un fichier ressource dont le nom dépend de la localisation
- Les fichiers ressources (fichier au format properties) sont stockés dans le répertoire javax.faces de la librairie implémentation de JSF (myfaces-impl-1.x.x.jar)
- Pour modifier un message pour un convertir donné
  - Créer un fichier properties dans le répertoire source
  - Déclarer le fichier properties dans le faces-config.xml
  - Copier les clés à redéfinir
  - Modifier la valeur du texte dans les clés

# Conversion de données : messages personnalisés

- Exemple : modification des messages d'erreurs de conversion

```
...  
    <application>  
        <message-bundle>beanPackage.MyMessages</message-bundle>  
    </application>  
</faces-config>
```

**javax.faces.convert.NumberConverter.CONVERSION = Problème pour le champs  
"Nombre de tickets"**  
**javax.faces.convert.NumberConverter.CONVERSION\_detail = "{0}" : La donnée  
n'est pas un nombre valide**

MyMessages.properties



# Conversion de données : attribut immediate

- Rappelons que toute requête JSF passe obligatoirement par un traitement défini par le cycle de vie JSF
- La conversion de données est obligatoirement traitée par le cycle de vie JSF même si par exemple une requête donnée n'est pas de valider un formulaire (annuler le traitement)
- Si une requête est d'annuler la saisie du formulaire (redirection vers une autre page), le processus effectuera toujours la conversion de données de tous les composants d'une vue JSF
- Il peut donc être utile de passer outre le processus de conversion : utilisation de l'attribut immediate pour une balise de composant qui doit soumettre le formulaire
- La présence de l'attribut immediate (valeur à true) modifie le déroulement du cycle de vie d'une requête JSF

# Conversion de données : attribut immediate

- Le composant passé à immediate sera géré en premier dans le cycle de vie
- Le fonctionnement donné ci-dessous ne concerne que les composants de type Command (CommandButton par exemple)
  - Après la phase Apply Request Values, le traitement de la requête va être redirigé directement vers Render Response
  - De cette manière les phases Process Validations, Update Model Values, Invoke Application ne sont pas exécutées
- Il n'y a donc pas de conversion, ni de modification dans le modèle
- Les différentes saisies de l'utilisateur ne sont jamais traitées et il n'y a donc pas d'erreurs qui pourraient bloquées l'utilisateur

# Conversion de données : attribut immediate

- Exemple : passer outre le processus de conversion

Nom :

Nombre de tickets :

Date :

Liste des messages :

## Exemples des Converters

[Utilisation d'un Converter en sortie \(currency\)](#)

[Utilisation d'un Converter en entrée \(currency\)](#)

[Utilisation d'un Converter en entrée/sortie \(currency et date\)](#)

[Utilisation d'un Converter en entrée/sortie \(currency et date\) avec possibil](#)

Les problèmes de conversion ne sont pas pris en compte

Nom :

Nombre de tickets :

Date :  "dateId": Conversion en D

Liste des messages :

- Problème pour le champs "Date"

Les problèmes de conversion sont pris en compte. Obligation de saisir correctement les données pour annuler

# Conversion de données : attribut immediate

- Exemple (suite) : passer outre le processus de conversion

```
<core:view>
<html:form>
...
    <html:outputText value="Date : " />
    <html:inputText id="dateId" value="#{beanconverter2.date}">
        <core:convertDateTime type="date" dateStyle="medium" />
    </html:inputText><html:message for="dateId" /><br>
    <html:outputText value="Liste des messages : " /><br>
    <html:messages />
    <html:commandButton value="Valider" />
    <html:commandButton value="Annuler (sans immediate)" action="CancelAction" />
    <html:commandButton value="Annuler (avec immediate)" action="CancelAction"
immediate="true" />
</html:form>
</core:view>

<faces-config>
    <navigation-rule>
        <from-view-id>*</from-view-id>
        <navigation-case>
            <from-outcome>CancelAction</from-outcome>
            <to-view-id>/index.jsp</to-view-id>
        </navigation-case>
    </navigation-rule>
...
</faces-config>
```

# Validation de données : principe

- L'objectif de la validation de données est de s'assurer que les informations qui ont été converties sont valides
- Les objets utilisés pour la validation sont appelés **Validators**
- Tous les composants utilisés pour la saisie de données peuvent exploiter des Validators
- Pourquoi utiliser des Validators (vérification de surface et de fond) ?
  - vérifier la présence de données
  - vérifier le contenu d'une données
  - vérifier la plage de valeurs (entier compris entre 20 et 50)
  - vérifier le format d'une donnée
  - ...

# Validation de données : cycle de vie

- Composant de la famille de type UIInput (déroulement proche du processus de conversion)
  - Dans une première étapes les paramètres sont récupérés et éventuellement convertis
  - Si la conversion et/ou la validation ont réussi, le traitement continu
- Si la conversion et/ou la validation ont échouées, le traitement est abandonné (pas de changement de page) et des messages d'erreur sont générés

# Validation de données : cycle de vie (via l'API)

- Les composants implémentant l'interface EditableValueHolder peuvent exploiter des Validators (UIInput)
- EditableValueHolder est une sous interface de ValueHolder
- Un objet EditableValueHolder fournit les propriétés pour ajouter un objet Validator et stocker l'état de validation
- Un objet Validator fournit une méthode
  - validate(FacesContext ct, UIComponent component, Object value) throws ValidatorException : en charge d'effectuer la validation

Note : s'il existe une erreur de validation, lancer une exception de type ValidatorException

# Validation de données : principe d'utilisation

- Pour utiliser un Validator ...
  - soit utiliser les standards fournis par la bibliothèque JSF
  - soit définir programmatiquement (dans un Bean)
- Utiliser les standards fournis
  - `core:validateDoubleRange` : valide les données de type Double
  - `core:validateLongRange` : valide les données de type Long
  - `core:validateLength` : valide la longueur de type Integer
- Dans tous les cas, ces balises fournissent trois attributs
  - `maximum` : valeur maximale
  - `minimum` : valeur minimale
  - `binding` : évalue une instance du validator en question



# Validation de données : validators standards

- Exemple : un formulaire, des champs et des validators ...

```
...
<html:form>
    <p>
        <html:outputText value="Veuillez saisir votre numéro de compte (10 chiffres)
        : " />
        <html:inputText id="compteId" value="#{validatorbean.compte}" >
            <core:validateLength maximum="10" minimum="10" />
        </html:inputText><html:message for="compteId" />
    </p>
    <p>
        <html:outputText value="Veuillez saisir votre taille (comprise entre 1.2 et
        2.0) : " />
        <html:inputText id="tailleId" value="#{validatorbean.taille}" >
            <core:validateDoubleRange maximum="2.80" minimum="1.20"/>
        </html:inputText><html:message for="tailleId" /><br>
        <html:commandButton value="Envoyer" />
    </p>
</html:form>
...
```

Veuillez saisir votre numéro de compte (10 chiffres) :

Veuillez saisir votre taille (comprise entre 1.2 et 2.0) :  "tailleId": La donnée n'est pas comprise entre 1,2 et 1,8.

# Validation de données : validators personnalisés

- Nous avons vu que JSF proposait en standard un ensemble restreint de Validators
- Il peut être intéressant de pouvoir développer ses propres Validators
- JSF offre la possibilité de définir programmatiquement les validators souhaités en utilisant des classes Java
- Deux approches de mise en oeuvre sont à distinguer :
  - Utilisation de la balise <core:validator>
  - Utilisation de l'attribut validator
- Rappelons que la mise en place de validators ne concerne que les composants de type UIInput

# Validation de données : validators personnalisés

- L'utilisation d'une balise pour lier un validator à un composant passe par l'implémentation de l'interface Validator
- Un objet Validator fournit une méthode
  - `validate(FacesContext ct, UIComponent cp, Object value)` throws `ValidatorException` : en charge d'effectuer la validation
    - `ct` : permet d'accéder au contexte de l'application JSF
    - `cp` : référence sur le composant dont la donnée est à valider
    - `value` : la valeur de la donnée
- L'exception permet d'indiquer si la validation ne s'est pas correctement passée
- Si exception déclenchée, au développeur à prendre en charge les messages à retourner via l'objet `FacesContext`

# Validation de données : validators personnalisés

- Pour utiliser une classe de type Validator, il faut la déclarer dans le fichier de configuration faces-config.xml
- Il faut identifier dans la balise validator la classe utilisée pour implémenter le Validator

```
<validator>
    <validator-id>myValidatorId</validator-id>
    <validator-class>beanPackage.MyValidator</validator-class>
</validator>
```

- Pour utiliser le validator dans une page JSP, il faut utiliser la balise <validator> dans le corps d'un composant JSF

```
<html:inputText value="#{...}" ... >
    <core:validator validatorId="myValidatorId" />
</html:inputText>
```

# Validation de données : validators personnalisés

- Exemple : utilisation de la balise <core:validator>

```
public class PersoValidator implements Validator {  
  
    public void validate(FacesContext fc, UIComponent comp, Object ref)  
    throws ValidatorException {  
        String myValue = null;  
        if (fc == null || comp == null || ref == null) {  
            throw new NullPointerException();  
        }  
  
        myValue = ref.toString();  
  
        final String magicNumber = "123456789";  
        if (!myValue.equals(magicNumber)) {  
            throw new ValidatorException(new FacesMessage(  
                FacesMessage.SEVERITY_ERROR,  
                "Problème de validation", "numéro magique erroné"));  
        }  
    }  
}
```

```
<validator>  
    <validator-id>myValidatorId</validator-id>  
    <validator-class>beanPackage.PersoValidator</validator-class>  
</validator>  
</faces-config>
```

# Validation de données : validators personnalisés

## ■ Exemple : utilisation de la balise <core:validator>

```
<core:view>
  <h4><html:outputText value="Utilisation d'un validator personnalisé :
    balise validator" /></h4>
  <html:form>
    <p>
      <html:outputText value="Veuillez saisir votre numéro magique (10 chiffres) : " />
      <html:inputText id="compteId" value="#{persobean.magicnumber}" >
        <core:validator validatorId="persoValidatorId"/>
      </html:inputText>
      <html:message for="compteId" showDetail="true" showSummary="true"/>
    </p>
    <p>
      <html:commandButton value="Envoyer" />
    </p>
  </html:form>
</core:view>
```

### Utilisation d'un validator personnalisé : balise validator

Veuillez saisir votre numéro magique (10 chiffres) :  Problème de validation numéro magique erroné

Envoyer

# Validation de données : validators personnalisés

- L'utilisation de l'attribut validator du composant UIInput à valider s'appuie sur l'utilisation d'un Bean
- Il faut donc fournir à l'attribut validator une expression qui désigne la méthode de validation

```
<html:inputText value="#{...}" validator="#{myVal.validate}" ... >  
...  
</html:inputText>
```

- Le nom de la méthode n'est pas imposé, toutefois elle devra respecter la signature de la méthode validate() de l'interface Validator
- L'inconvénient de cette approche est qu'elle est fortement liée à l'application et il devient difficile de réutiliser le validator

# Validation de données : validators personnalisés

- Exemple : utilisation de l'attribut validator

```
public class PersoBeanValidator {
    private String magicNumber;
    public String getMagicnumber() {
        return magicNumber;
    }
    public void setMagicnumber(String magicNumber) {
        this.magicNumber = magicNumber;
    }

    public void validatePerso(FacesContext fc, UIComponent comp, Object ref)
    throws ValidatorException {
        String myValue = null;
        if (fc == null || comp == null || ref == null) {
            throw new NullPointerException();
        }

        myValue = ref.toString();
        final String magicNumber = "123456789";
        if (!myValue.equals(magicNumber)) {
            throw new ValidatorException(new FacesMessage(
                FacesMessage.SEVERITY_ERROR, "Problème de validation",
                "numéro magique erroné"));
        }
    }
}
```



# Validation de données : validators personnalisés

- Exemple (suite) : utilisation de l'attribut validator

```
<core:view>
  <h4><html:outputText value="Utilisation d'un validator personnalisé :
  attribut validator" /></h4>

  <html:form>
    <p>
      <html:outputText value="Veuillez saisir votre numéro magique (10
      chiffres) : " />
      <html:inputText id="compteId" value="#{persobeanvalidator.magicnumber}"
      validator="#{persobeanvalidator.validatePerso}" />
      <html:message for="compteId" showDetail="true" showSummary="true"/>
    </p>

    <p><html:commandButton value="Envoyer" /></p>
  </html:form>
</core:view>
```

## Utilisation d'un validator personnalisé : attribut validator

Veuillez saisir votre numéro magique (10 chiffres) :  Problème de validation numéro magique erroné

Envoyer