GLG 203 : MVC et JSF

Serge Rosmorduc
Conservatoire national des arts et métiers

Bibliographie

- *The Java EE 6 Tutorial*, http://docs.oracle.com/javaee/6/tutorial/doc/javaeetutorial6.pdf

- JSR 344

- A. Goncalves, « Beginning Java EE 6 Platform »

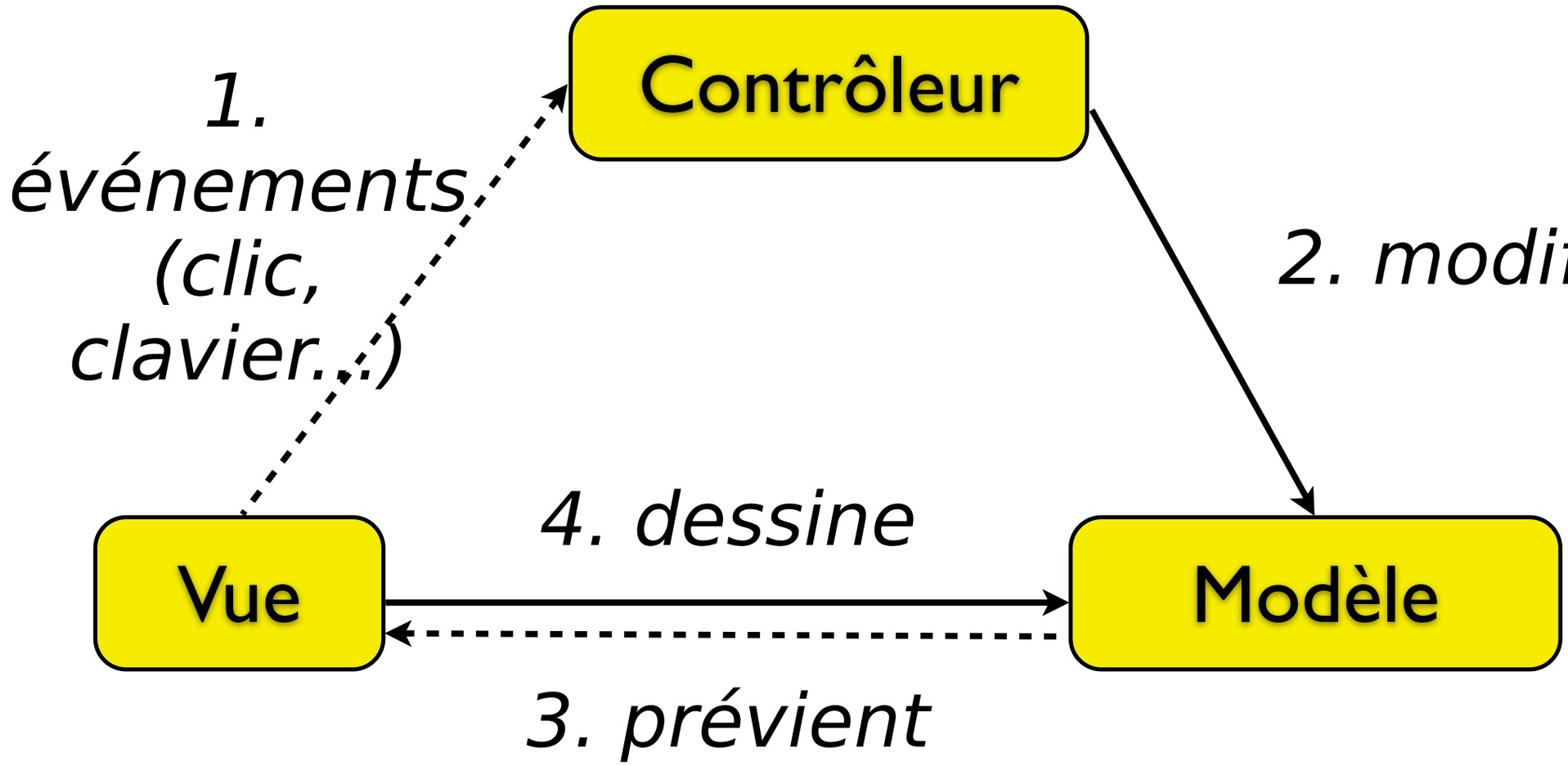
- Cours de J-L Dewez GLG203/2009

- Cours de J-M Douin GLG203/2012

Bibliographie (suite)

- http://www.ibm.com/developerworks/library/j-jsf1/
- http://www.ibm.com/developerworks/library/j-jsf2/
- http://www.ibm.com/developerworks/library/j-jsf3/
- http://www.ibm.com/developerworks/library/j-jsf4/
- http://www.ibm.com/developerworks/java/library/j-facelets/
- Série « JSF-2Fu »sur http://www.ibm.com/developerworks/ :
  - http://www.ibm.com/developerworks/java/library/j-jsf2fu1/index.html
- http://www.jsftutorials.net/

MVC: version desktop

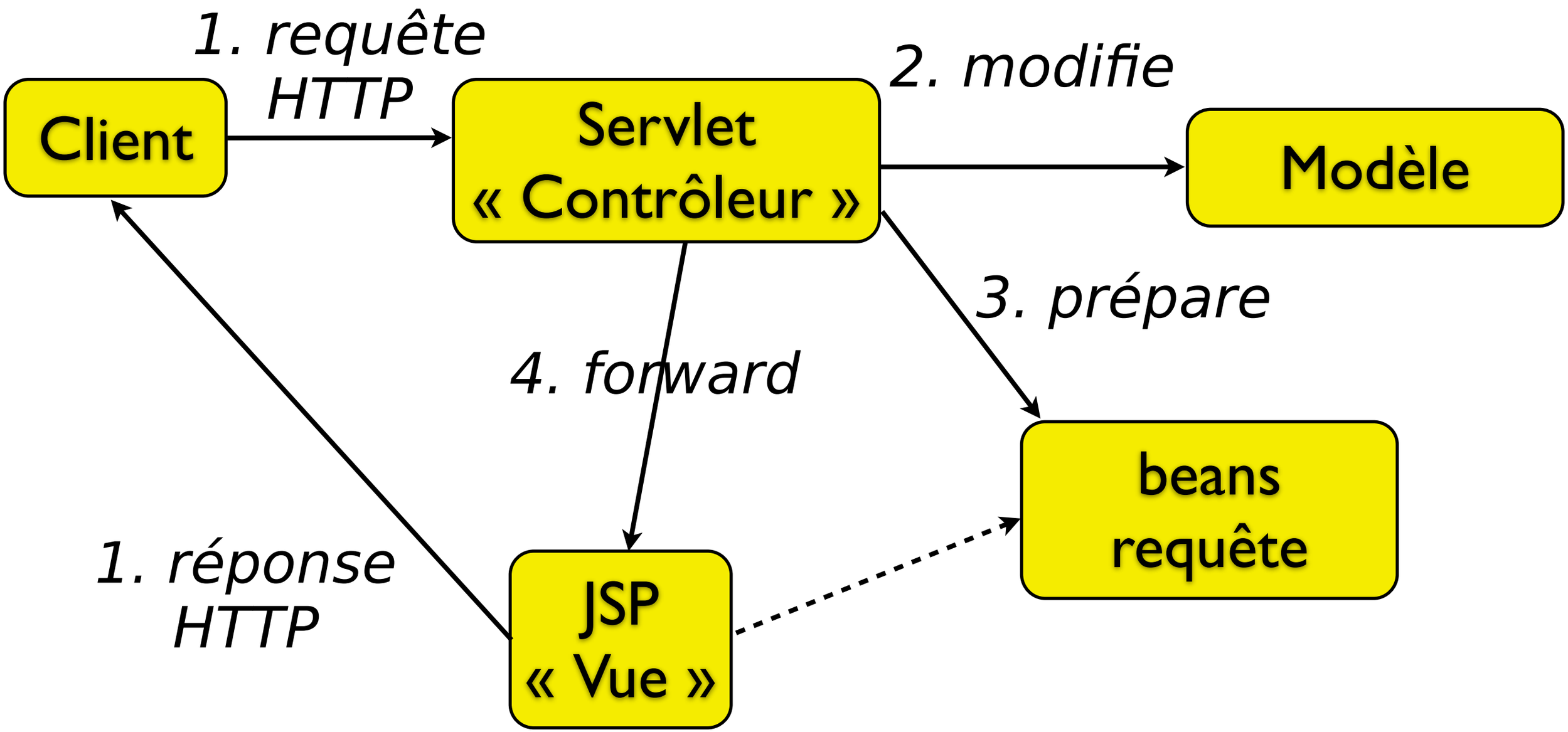


⟶ connaît le type

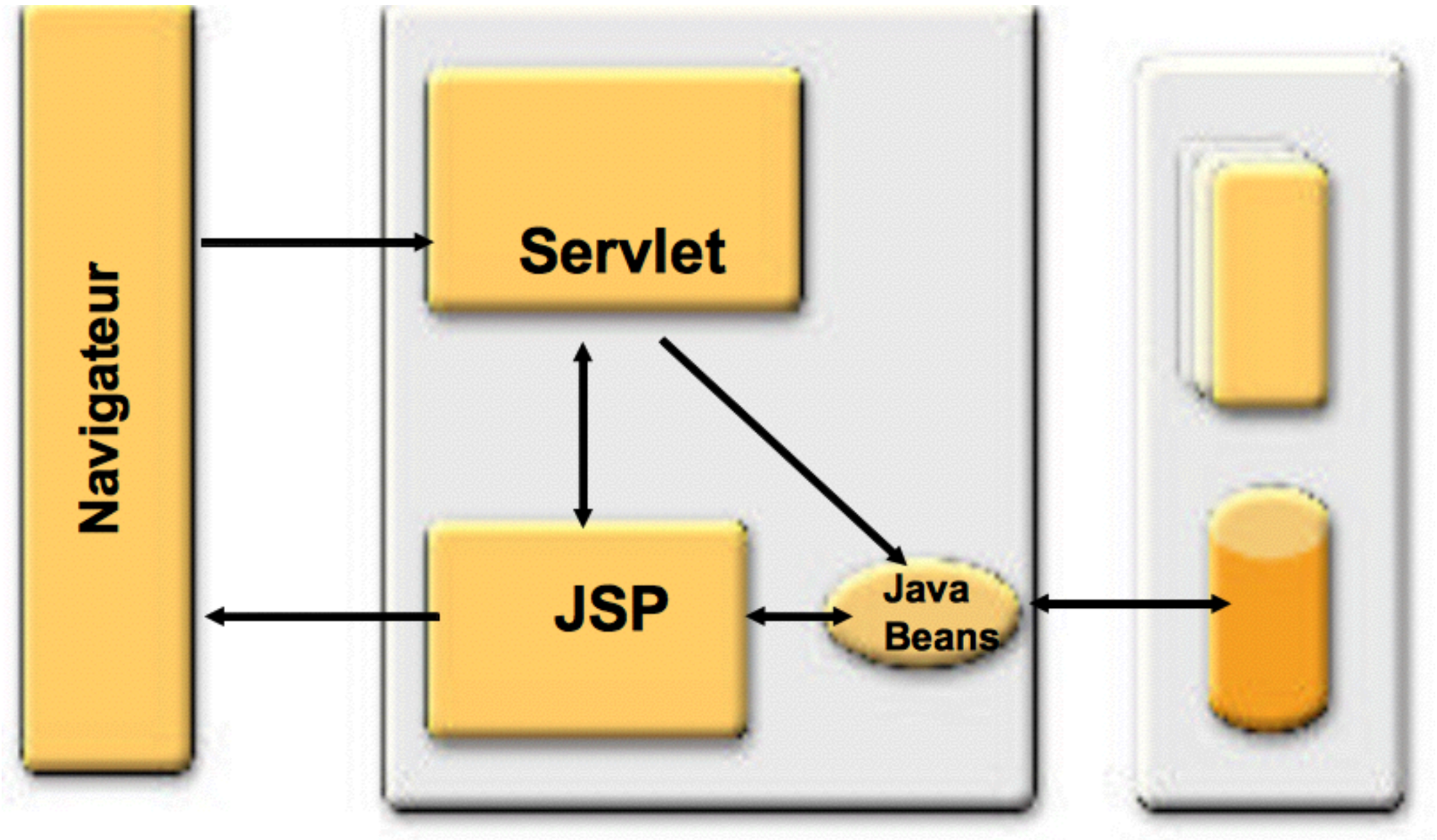
⇢ lien par une interface

(Il y existe des variantes)

MVC: version Web

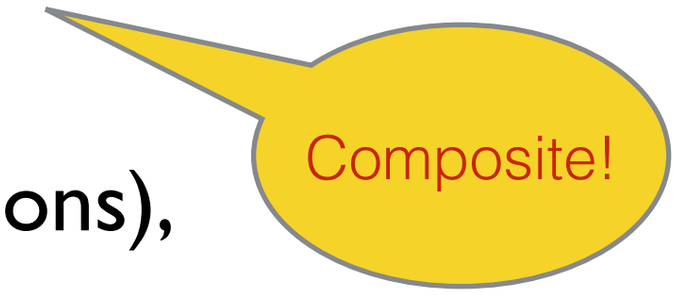


Architecture Servlet+JSP



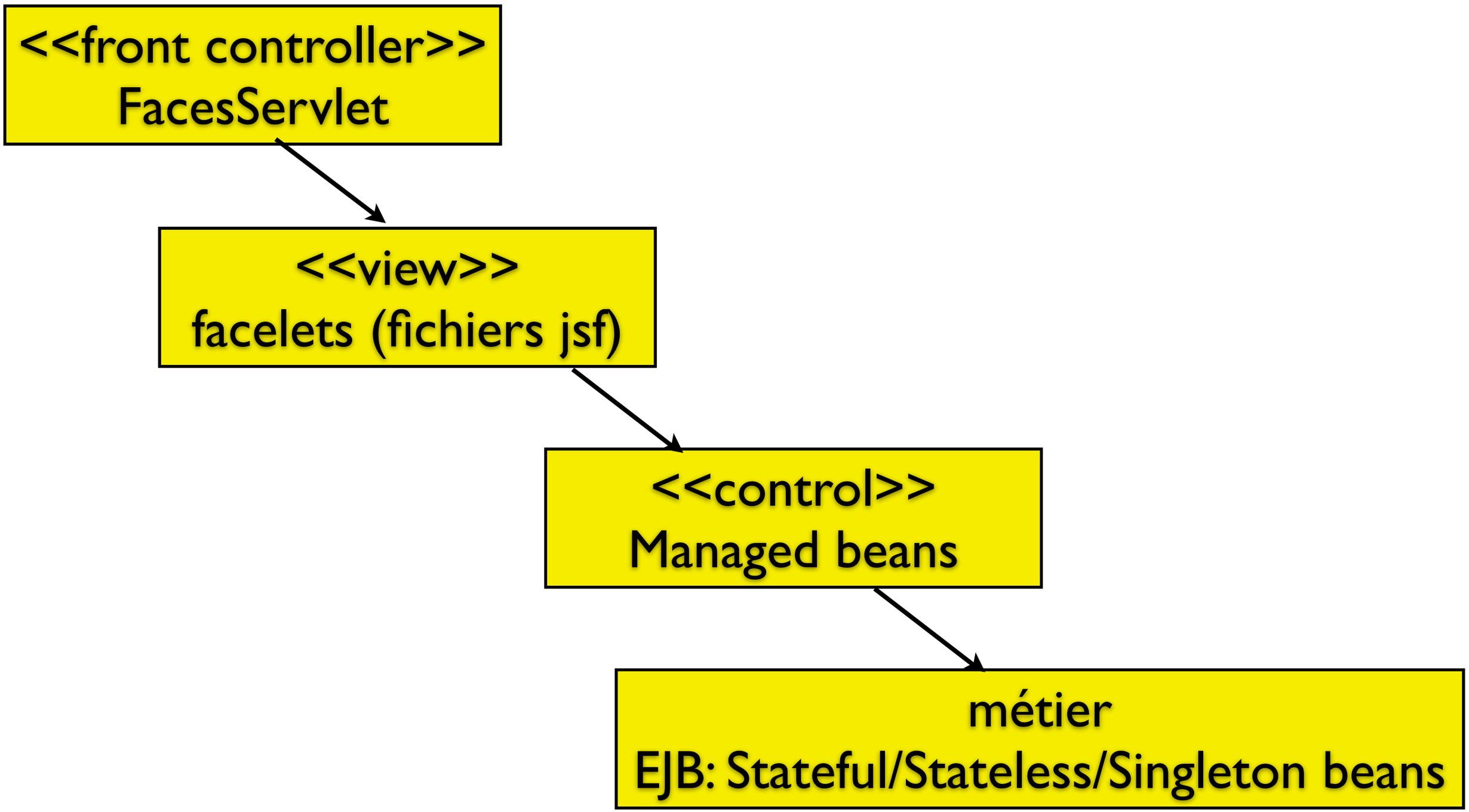
JSF

- Fondé sur l'architecture des servlets
- Une Servlet « **front-controller** » est fournie
- architecture à base de composants: une page JSF décrit un arbre de composants objets java, qui sont utilisés pour engendrer le résultat final.



- accès facile aux beans (à l'aide d'annotations), injection de dépendances
- gestion de la navigation, de la validation des données, ajax, etc.

architecture JSF



Mise en place des JSF

- 1) On déclare la servlet FrontController : FacesServlet

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
   <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>
web-app>
```

Mise en place des JSF

- 2)(Optionnel)
  Fichier faces-config.xml dans WEB-INF

```
<?xml version='1.0' encoding='UTF-8'?>
<faces-config version="2.1">
</faces-config>
```

ManagedBeans et Facelets

- Deux éléments principaux pour la partie interface utilisateur

- Facelets : vue, plus sophistiquées que les JSP (qui peuvent aussi être utilisées)

- ManagedBeans : implantent le contrôle, contiennent les données des formulaires…

Managed beans

- **Contrôle (donc, font partie de l'UI)**
- Annotés avec @ManagedBean ou @Named
- **Portées** (scope): @SessionScoped, @RequestScoped, @ApplicationScoped, @ViewScoped
- Accès possible aux EJB par injection.

Portées des beans

- @Dependant : le bean est utilisé uniquement pour un composant. Si plusieurs références sont faites dans la même page, il s'agit d'un bean différent à chaque fois

- @RequestScoped : le bean « vit » le temps de l'exécution de la requête (donc du retour du résultat à l'utilisateur)

- @SessionScoped : le bean vit le temps d'une session.

- @ViewScoped : le bean vit le temps qu'une même vue est affichée, c'est à dire qu'aucune action effectuée ne change de page (ne retourne une valeur non nulle)

- @ConversationScoped : le bean vit le temps d'une « conversation ». Permet d'écrire des « wizards » (suites de dialogues).

- @ApplicationScoped : bean partagé au niveau de l'application.

Remarques sur la portée des beans

- @RequestScoped : temps d'une requête… bien comprendre quand démarre et quand s'arrête la requête

- @ViewScoped : durée de vie complètement différente de @RequestScoped.

  - finit quand on quitte la page (RequestScoped « vit » jusqu'à l'affichage de la page résultat)

  - survit par contre à des requêtes faites en restant sur la même page

  - bien adapté à l'utilisation d'Ajax.

Facelets

- **VUE**
- Ressemblent à des JSP, mais :
  - Le fichier XML est analysé, et **une représentation objet du document** est construite (JSP= texte) ;
  - le contenu HTML final est engendré par cette représentation, appliquée aux valeurs contenues dans les Managed beans ;
  - (pas de représentation objet intermédiaire en JSP)
  - il faut utiliser '#' à la place de '$' pour accéder aux beans.

Exemple

Managed bean

```
import javax.faces.bean.*;
@ManagedBean @SessionScoped
public class CounterControl {
    private int count=0;
    public int getCount() {
        return count++;
    }
}
```

Facelet

```
<!DOCTYPE HTML>
<html xmlns="http://www.w3.org/1999/xhtml"
        xmlns:h="http://java.sun.com/jsf/html">
    <h:head></h:head>
    <h:body>
    Count value: <h:outputText value="#{counterControl.count}"/>
    </h:body>
</html>
```

Formulaires en JSF

- Inclus dans une balise <h:form>
- gérés par un Managed bean
- les champs du formulaire utilisent des balises JSF (et non HTML)
- le managed bean va automatiquement récupérer les valeurs des champs, et réaliser la ou les actions du formulaire.

Formulaires (exemple simple)

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
  <h:head></h:head>
  <h:body>
    <h:form>
      <h:panelGrid columns="2">
        a <h:inputText value="#{addControl.first}"/>
        b <h:inputText value="#{addControl.second}"/>
        result <h:inputText value="#{addControl.result}"/>
        <h:commandButton value="add"
                         action="#{addControl.add()}"/>
      </h:panelGrid>
    </h:form>
  </h:body>
</html>
```

Lie le contrôle à une propriété du bean

méthode appelée quand le bouton est pressé.

Formulaires:
le bean de support

```
import javax.faces.bean.*;
@ManagedBean
@RequestScoped
public class AddControl {
    private int first, second, result;

    public void add() {
        result= first+ second;
    }
    .... accessors for properties ....

}
```

pas de valeur de retour: on reste sur la même page

Formulaires

- Les données du formulaire sont *injectées* dans le bean
- validation automatique ; affichage de messages d'erreurs si la validation n'est pas correcte
- le formulaire est réaffiché s'il n'est pas valide.

Formulaires et messages d'erreur

```
<h:inputText label="a" id="a" value="#{addControl.first}"/>
<h:message for="a"/>
```

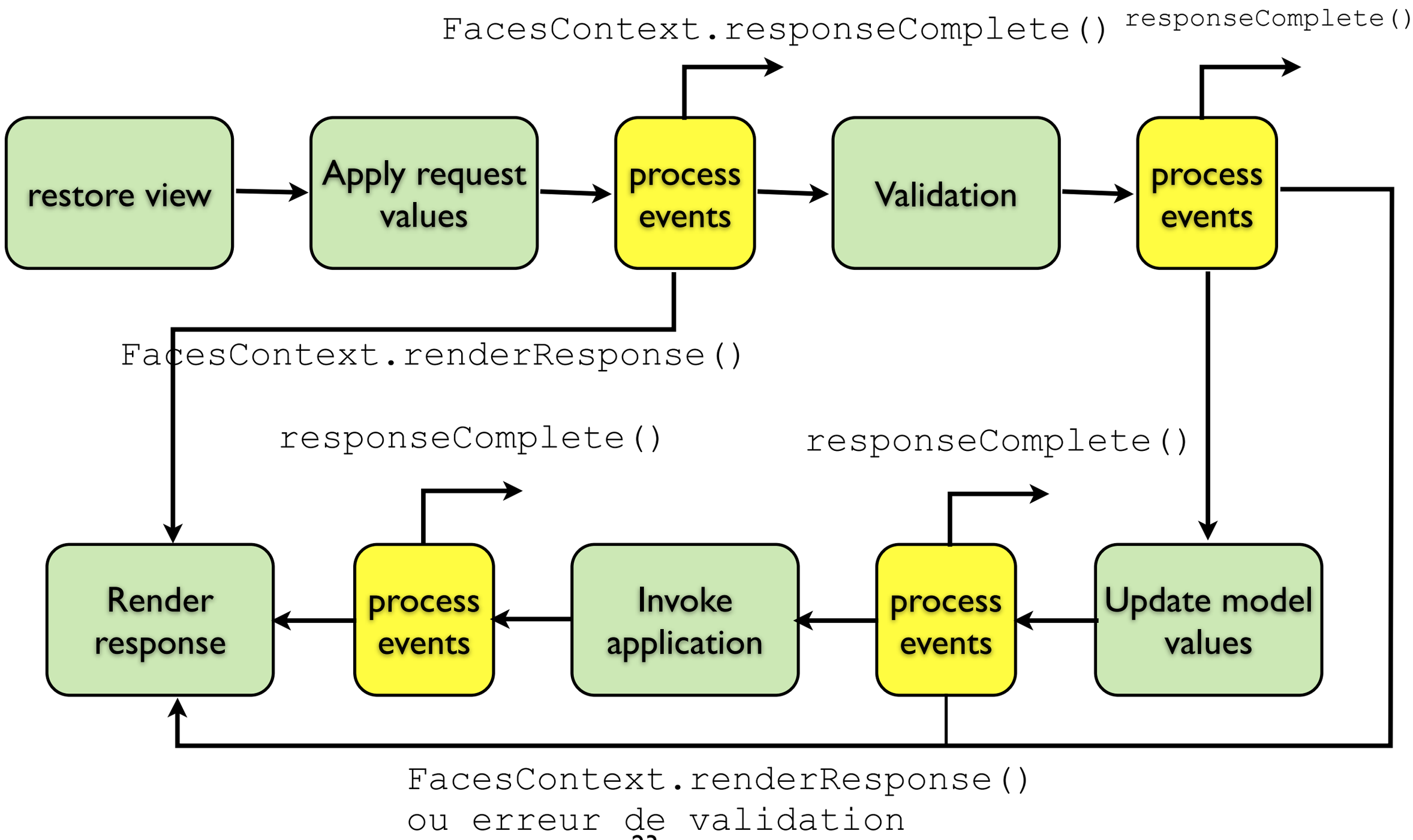
- id: identifie le champ
- label : nom utilisé dans le message d'erreur
- for : réfère à l'identifiant du champ

Cycle de vie des JSF

- Pour chaque requête, la séquence des interactions entre la *facelet* et les beans est bien déterminée

- On peut y « brancher » des gestionnaires d'événements pour modifier les données en cours de route

Gestion d'une requête (source : Oracle)



Cycle de vie

- Restore view: construire l'arbre des composants
- Apply request values: fixe les valeurs des paramètres
- Validation : vérification des valeurs pour la validation jsf
- Update model : copie des valeurs dans les beans. validation lors de la copie
- Invoque application : exécution de la commande
- Render response: création du résultat
- Après la plupart des phases, les événements peuvent être traités.
- Interruption du traitement:
  - appel de FacesContext.renderResponse : saute directement à la dernière phase
  - appel de FacesContext.responseComplete() : production directe du résultat final (par exemple PDF, image…)

Autre vision

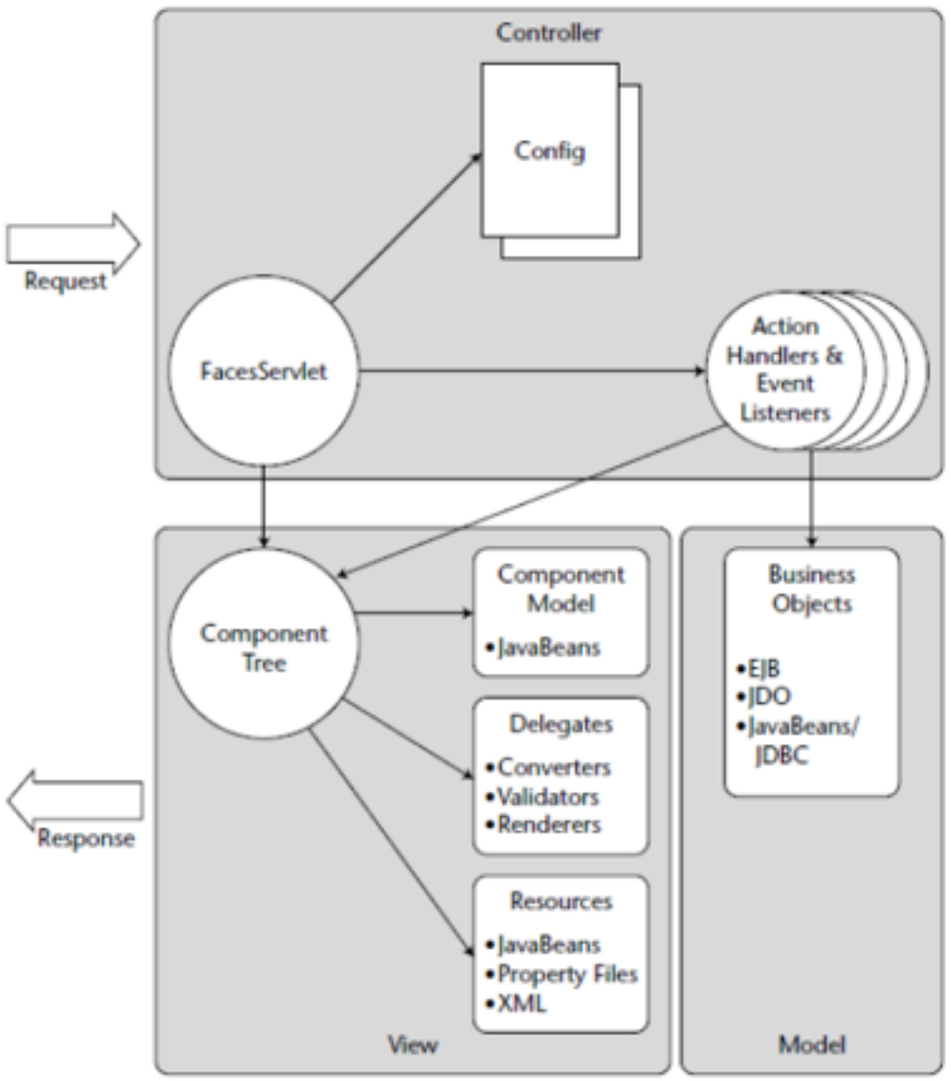


Figure 1.6 JSF Model-2.

(Mastering JSF page 19)

Navigation en JSF

- Les actions peuvent renvoyer une String, qui dit quelle est la prochaine vue.
- On peut aussi, dans un lien de la JSF, donner le nom d'une vue au lieu d'une action
- La chaîne renvoyée par l'action ou le lien peut être le nom d'une vue, ou un résultat abstrait (comme « success » ou « error ») qui sera utilisé par le fichier faces-config.xml pour décider de la prochaine vue.

Exemple très simple

(sur la page page index.xhtml)

<h:link value="exemple" outcome="add"/>

- crée un lien avec le texte «exemple»
- saute à la vue «add», si elle existe (s'il y a un fichier add.xhtml dans les *facelets*).
- **ou** : cherche une règle de navigation pour «add»

Exemple avec une règle de navigation

dans add2.xhtml

```
<h:form>
a <h:inputText label="a" id="a" value="#{addControl.first}"/>
b <h:inputText label="b" id="b"  value="#{addControl.second}"/>
 <h:commandButton value="add" action="#{addControl.add2()}"/>
 </h:form>
```

dans addResult.xhtml

```
<h:body>
 Addition result :
 <h:outputText value="#{addControl.result}"/>
 <h:link value="back to index" outcome="index"/>
</h:body>
```

managed bean:

```
@ManagedBean
@RequestScoped
public class AddControl {
    private int first, second, result;
     ....
     public String add2() {
        result= first+ second;
        return "success";
     }
}
```

Exemple avec une règle de navigation

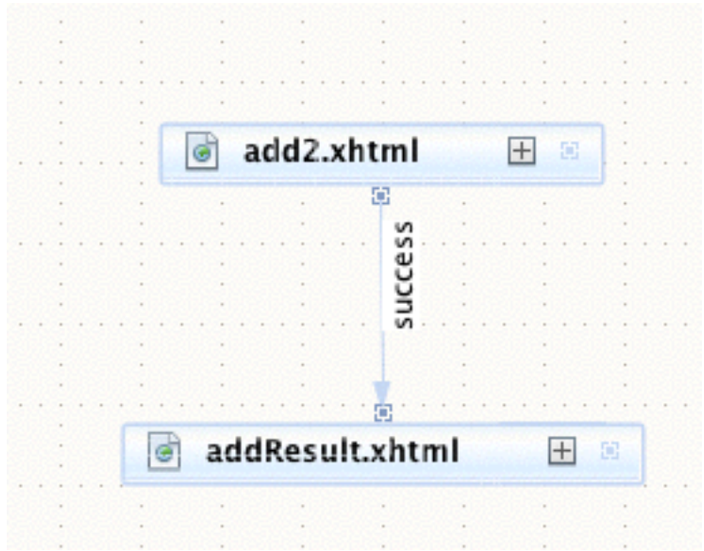
faces-config.xml

```
<navigation-rule>
      <from-view-id>/add2.xhtml</from-view-id>
      <navigation-case>
         <from-outcome>success</from-outcome>
         <to-view-id>/addResult.xhtml</to-view-id>
    </navigation-case>
</navigation-rule>
```

- Quand une action appelée depuis add2.xhtml retourne «success», montre addResult.xhtml.
- Permet de séparer la navigation du résultat des actions.

Règles de navigation

- Le flux de navigation peut être affiché par netbeans:



De quelques composants JSF (liste pratique à http://www.exadel.com/tutorial/jsf/jsftags-guide.html#output)

Le composite JSF

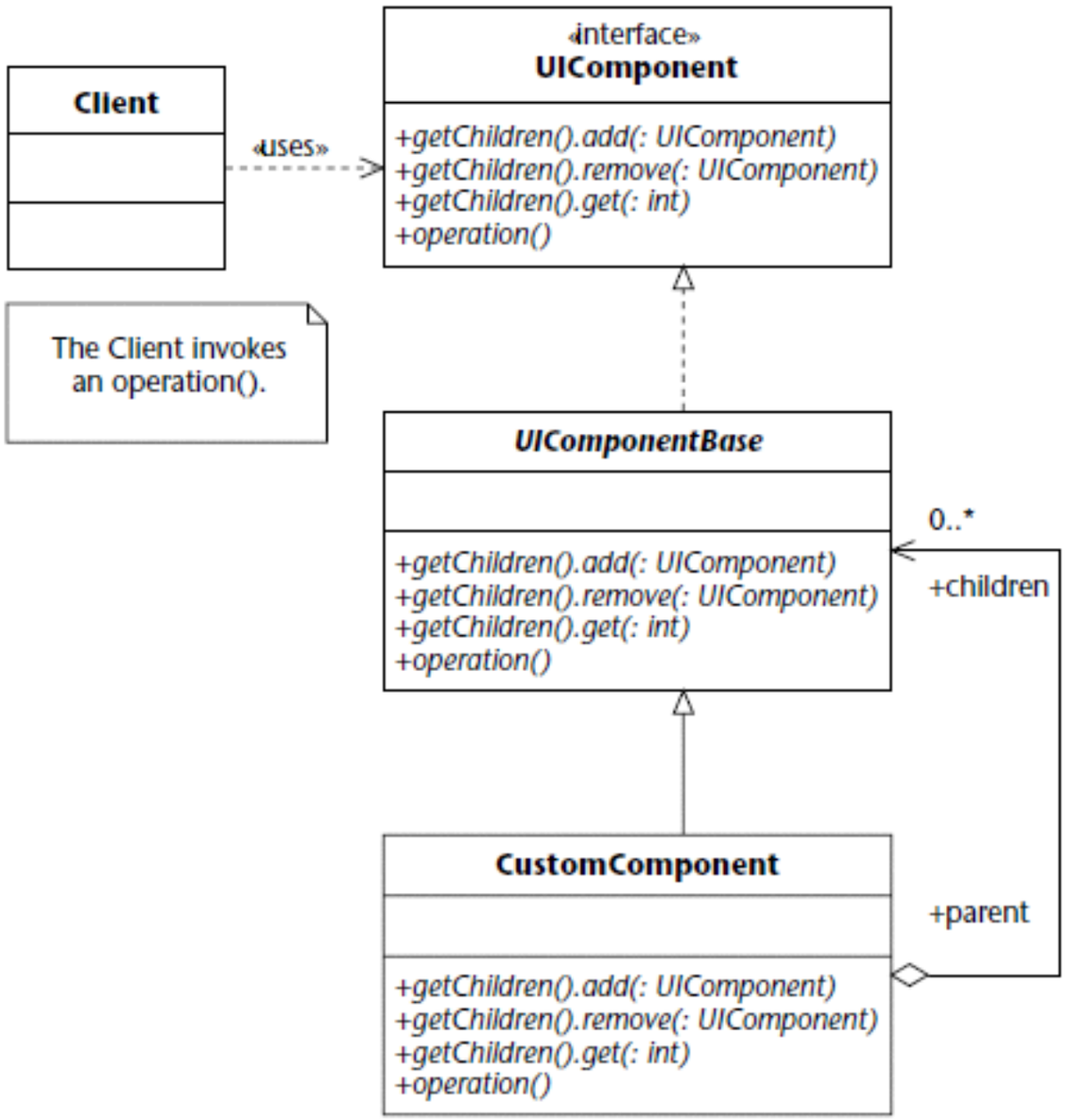


Figure 1.12 JSF Composite structure.

Un peu de structure

- <h:form>: délimite un formulaire

- <h:panelGrid columns="2"> présente son contenu sur deux colonnes

Affichage

```
<h:outputText value="#{bean.texte}"/>
```

un texte

```
<h:outputLabel value="Nom" for="idNom"/>
<h:inputText id='idNom' value="#{bean.value}"/>
```

Nom

```
<h:graphicImage value=‘toto.png'/>
```

Boutons et liens

- Deux types: liés aux actions (command) ou non.

```
<h:commandButton action="#{bean.faire()}" value="allez!"/>
```

allez!

```
<h:commandLink action="#{bean.faire()}" value="allez!"/>
```

allez!

```
<h:outputLink value="http://www.google.com/search">
   Google
<f:param name="q" value="glg203"/>
</h:outputLink>
```

Google

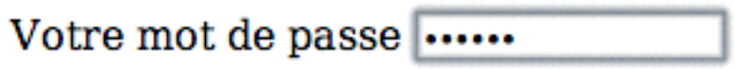
```
<h:button outcome="index" value="page d'accueil"/>
```

page d'accueil

Entrées

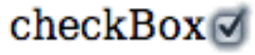
```
<h:panelGrid columns="2">
  Votre nom <h:inputText value="#{bean.value}"/>
</h:panelGrid>
```


```
Votre mot de passe
<h:inputSecret  value="#{bean.value}"/>
```




```
<h:inputTextarea  value="#{bean.value}" cols="80" rows="10"/>
```

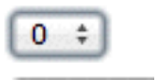
```
<h:selectBooleanCheckbox value="#{bean.value}">
checkBox
</h:selectBooleanCheckbox>
```



Listes

- Avec entrées « en dur »:

```
<h:selectOneMenu value="#{bean.value}">
  <f:selectItem itemDescription="lundi" itemValue="0"/>
  <f:selectItem itemDescription="mardi" itemValue="1"/>
</h:selectOneMenu>
```




```
<h:selectOneMenu value="#{bean.value}">
  <f:selectItems value="#{bean.choixPossibles}"/>
</h:selectOneMenu>
```

De même : selectOneListBox, selectOneRadio

choixPossibles retourne une liste.
le toString() des éléments est utilisé pour l'affichage

Affichage de tableaux

```
<h:dataTable var="p" value="#{personControl.personList}">
        <h:column>
            <f:facet name="header">
                <h:outputText value="Surname"/>
            </f:facet>
            <h:outputText value="#{p.surname}"/>
        </h:column>
        <h:column>
            <f:facet name="header">
                <h:outputText value="Name"/>
            </f:facet>
            <h:outputText value="#{p.name}"/>
        </h:column>
</h:dataTable>
```

Affichage conditionnel en JSF

- facile: les balises JSF ont un attribut booléen «rendered».
- pour un contenu complexe, on peut le mettre dans un **panelGroup**

```
<h:outputText rendered="#{empty personControl.personList}"
                    value="The person list is empty"/>
<h:panelGroup
        rendered="#{not empty personControl.personList}">
    The person list contains
   <h:outputText value="#{personControl.personList.size()}"/>
   persons.
</h:panelGroup>
```

JSF et ajax

- En JSF 2, l'utilisation d'ajax est simple : on insère f:ajax dans le composant à « ajaxifier »
- l'action liée au composant est exécutée de manière asynchrone

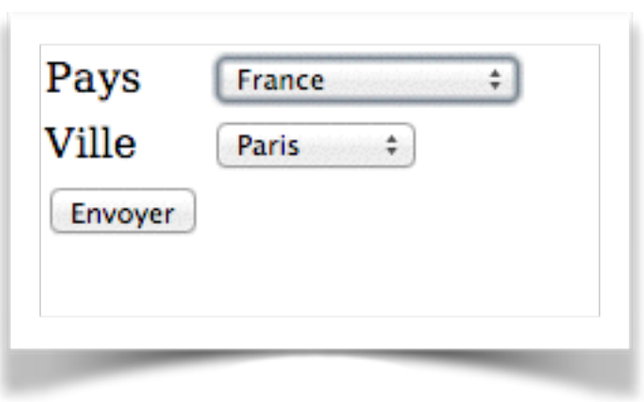
```
<h:form>
   a <h:inputText label="a" id="a" value="#{addControl.first}"/>
   b <h:inputText label="b" id="b"  value="#{addControl.second}"/>
   result <h:outputText id="res" value="#{addControl.result}"/>
   <h:commandButton value="add" action="#{addControl.add()}">
      <f:ajax execute="@form" render="res"/>
   </h:commandButton>
</h:form>
```

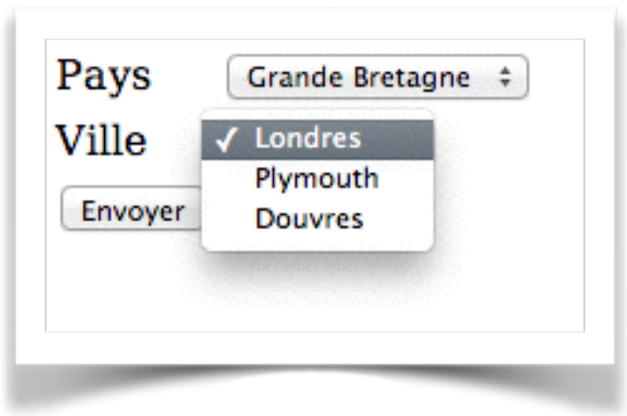
Ajax

- f:ajax associe un comportement ajax aux actions de certains éléments.
- La balise f:ajax est placée dans l'élément (au sens XML) à ajaxifier, ou autour d'un certain nombre d'éléments.
- attributs
  - event : dépend du composant (plusieurs événements possibles parfois). Exemple « click », « keyup »… voir événements javascript
  - render: quels éléments doivent être redessinés (id, ou @form, @this, @all)
  - execute : quels éléments doivent être transmis aux beans @this, @all, @none, @form..., ids (défaut: @this)

Exemple d'ajax

- Sélection d'un pays et d'une ville
- Quand on sélectionne un pays, la liste des villes doit être mise à jour





- Solution: le changement de choix de pays doit déclencher une mise à jour en ajax.

Exemple ajax: la JSF

```
<h:form id="adresse">
<h:panelGrid columns="2">
   Pays <h:selectOneMenu id="pays" value="#{adresseBean.pays}">
                        <f:selectItems value="#{adresseBean.listePays}"/>
                        <f:ajax render="pays ville" execute="@this"/>
     </h:selectOneMenu>

Ville <h:selectOneMenu id="ville" value="#{adresseBean.ville}">
        <f:selectItems value="#{adresseBean.listeVilles}"/>
    </h:selectOneMenu>

   <h:commandButton action="#{adresseBean.envoyer()}" value="Envoyer"/>
</h:panelGrid>
</h:form>
```

Exemple Ajax… suite

```
<h:selectOneMenu id="pays" value="#{adresseBean.pays}">
   <f:selectItems value="#{adresseBean.listePays}"/>
   <f:ajax render="pays ville" execute="@this"/>
</h:selectOneMenu>
```

- <f:ajax render="pays ville" execute="@this"/> signifie:

- quand l'événement par défaut de cet objet est exécuté (ici, quand on sélectionne une nouvelle valeur), appelle les setters correspondant à **execute** (ici, @this: le composant selectOneMenu)

- … puis mets à jour les composants qui correspondent à **render**, ici le composant d'id **pays** et le composant d'id **ville**

Valeurs possibles pour render et execute

- a priori, une liste de valeurs séparées par des espaces :
  - id d'un élément dans le même formulaire
  - id d'un élément dans un autre formulaire, avec notation « :idparent:idElement »
  - @this désigne l'élément sur lequel porte la balise ajax
  - @form désigne le formulaire dans son ensemble
  - @all désigne tous les composants
  - @none aucun composant

Du côté du Managed Bean

```
@Named(value = "adresseBean")
@ViewScoped
public class AdresseBean implements Serializable {
    private String pays, ville;

    private TreeMap<String, List<String>> villeMap = …

    @PostConstruct
    public void init() {
      … (remplit la map) …
    }

    public List<String> getVilles(String pays) {return villeMap.get(pays);}

    public Collection<String> getListeVilles() {
        if (pays == null) {
            return new ArrayList<>();
        } else {
            return getVilles(pays);
        }
    }

    public Set<String> getListePays() {
        return villeMap.keySet();
    }

    … getters et setters pour ville et pays…
    public String envoyer() {
      …
    }
}
```

Le problème de la portée du bean

- Le bon choix de portée ici est @ViewScoped.

- @RequestScoped: le bean est recréé à chaque appel ajax : il oublie les valeurs choisies

- @SessionScoped: durée de vie trop longue. Utilise des ressources inutilement.

Injecter les paramètres d'une requête dans un bean

- par exemple pour des pages qui sont chargées en mode GET
- Pour les beans de scope « request » uniquement
- Annoter la propriété dans le bean avec:
  - @ManagedProperty(value="#{param.*NAME*}")
  - où « name » est le nom du paramètre

Méthode GET en JSF

- Pour forcer l'usage de la méthode GET, utilisez h:button ou h:link.

```
<h:link value="view"  outcome="viewMessage">
    <f:param name="messageId" value="#{m.id}"/>
</h:link>
```

appellera http://adresse/viewMessage?messageId=**N**

FacesContext

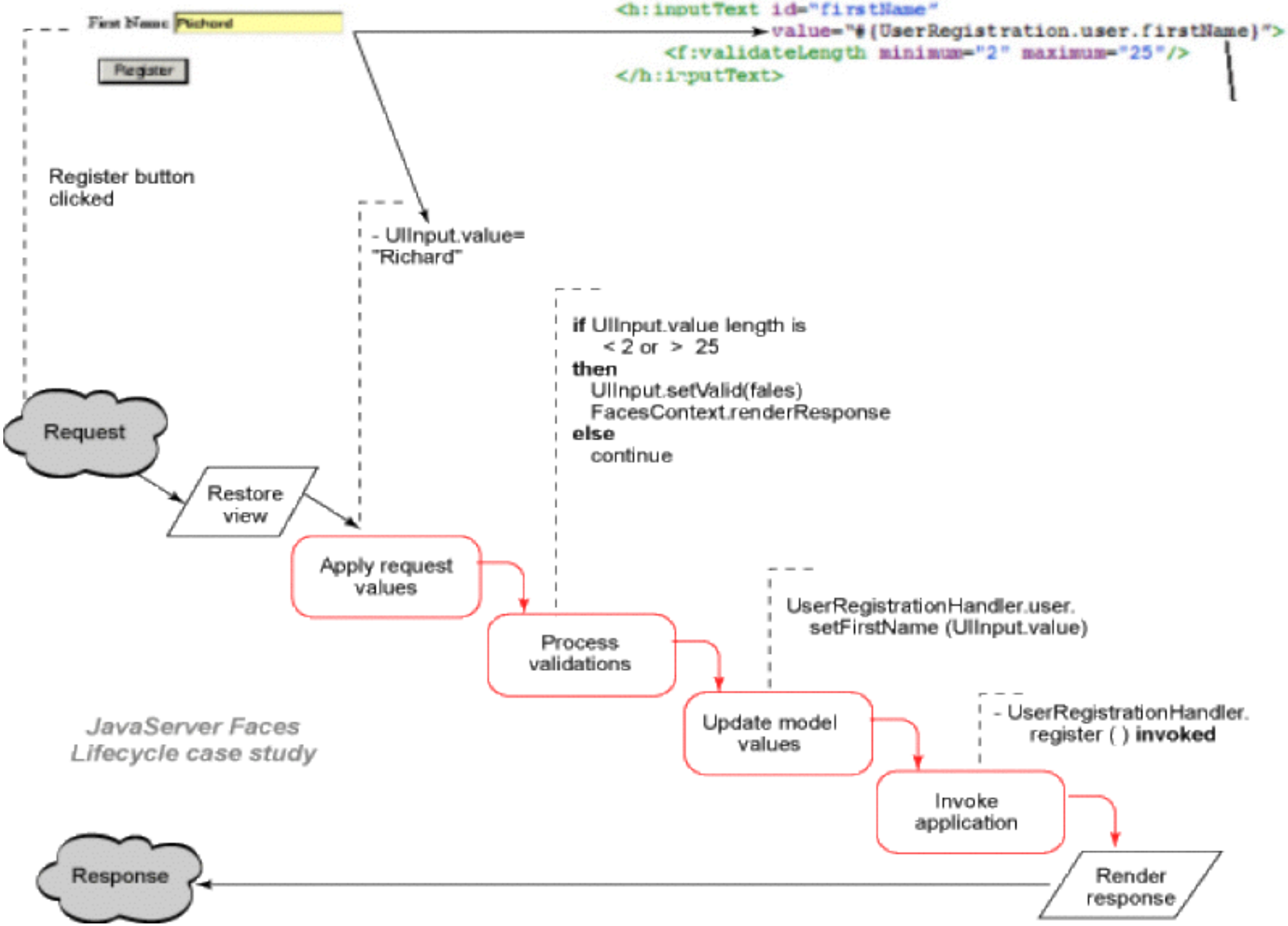
- Permet l'accès bas niveau aux données de la requête, à la session, etc.
- Se récupère dans un ManagedBean avec:

  FacesContext ctx=
  FacesContext.getCurrentInstance();

- ctx.getExternalContext() donne accès à la requête http, à la session…

Validation

- Possible à plusieurs niveaux:
  - validation JSF : installation de valideurs associés à des champs
  - validation de bean : règles associées aux propriétés des managed beans


First Name: Richard
Register
<h:inputText id="firstName"
value="#{UserRegistration.user.firstName)">
<f:validateLength minimum="2" maximum="25"/>
</h:inputText>
Register button clicked
- UIInput.value= "Richard"
if UIInput.value length is
< 2 or > 25
then
UIInput.setValid(fales)
FacesContext.renderResponse
else
continue
Request
Restore view
Apply request values
Process validations
UserRegistrationHandler.user.
setFirstName (UIInput.value)
Update model values
- UserRegistrationHandler.
register ( ) invoked
Invoke application
Render response
Response
JavaServer Faces
Lifecycle case study

Valideurs JSF

- Il est possible de contraindre la validation des champs dans le textes des JSF, soit en spécifiant en validator comme argument, soit en fournissant un validator dans l'annotation <f:validator…>

Validation par méthode

```
@Named(value = "demoValidation")
@RequestScoped
public class DemoValidation {
    private String val1="0", val2="0";
    private int resultat;

    public void validerHexa(FacesContext context,
                UIComponent toValidate,
                Object value) {
            String string = (String) value;
            if (! string.matches("^[0-9A-Fa-f]+$")) {
                ((UIInput) toValidate).setValid(false);
                FacesMessage message =
                        new FacesMessage("Mauvais format de nombre");
                context.addMessage(toValidate.getClientId(context),
                        message);
            }
    }
  //… reste des méthodes
}
```

Objet Validator

- avantage par rapport à la méthode: réutilisable

```
@FacesValidator("hexValidator")
public class HexValidator implements Validator {
   public void validate(FacesContext context, UIComponent component,
Object value) throws ValidatorException {
        String string = (String) value;
        if (!string.matches("^[0-9A-Fa-f]+$")) {
            ((UIInput) component).setValid(false);
            FacesMessage message =
                 new FacesMessage("Mauvais format de nombre");
            context.addMessage(component.getClientId(context),
                               message);
        }
    }
}
```

Utilisation d'un valideur

- Le valideur doit implanter l'interface javax.faces.validator.Validator
- On doit le déclarer et lui donner un ID, **soit** dans faces-config.xml, **soit** en l'annotant avec @FaceValidator("**ID**")
- On l'emploie avec <f:validator id="…">

```
<?xml version='1.0' encoding='UTF-8'?>
<faces-config version="2.2" …>
    <!-- déclaration du valideur (en dehors de « application »!!!)->
    <validator>
        <validator-id>hexValidator</validator-id>
        <validator-class>glg203.ui.HexValidator</validator-class>
    </validator>

    <application>
        <locale-config>
            <default-locale>fr</default-locale>
        </locale-config>
    </application>
</faces-config>
```

c'est optionnel, les annotations étant plus simples

Utilisation dans la JSF

```
<h:form>

  <!– champ validé par un appel de méthode –>
  <h:inputText value="#{demoValidation.val1}"
               validator="#{demoValidation.valideurHexa}"/>

  <!– champ validé par un objet valideur –>
  <h:inputText value="#{demoValidation.val2}">
     <f:validator validatorId="hexValidator"/>
  </h:inputText>

  <h:outputText value="#{demoValidation.resultat}"/>
  <h:commandButton action=« #{demoValidation.action()} »
                   value="Somme"/>
</h:form>
```

méthode à appeler

id du valideur

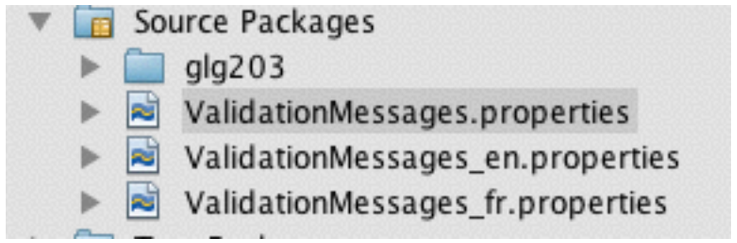
Validation des propriétés

- Automatique pour le tapage
- Plus précis et général : annotations de javax.validation (JSR 303)

```
@Named("helloBean")
@RequestScoped
public class HelloBean {

    @Size(min = 1, max = 100,message = "{erreur.longueur}")
    private String nom;

    @Pattern(regexp = "(\\p{L}| )+", message = "{erreur.prenom}")
    private String prenom;

    @Min(0)
    @Max(200)
    private int age;
```

Messages d'erreurs internationaux pour JSR 303

- messages par défaut disponibles
- pour en donner d'autres: utiliser un fichier nommé **ValidationMessages.properties**, *placé **impérativement** dans le package par défaut*
- format usuel des fichiers de propriétés
- **concerne uniquement JSR 303.**



- on peut évidemment utiliser d'autres ressources pour traduire le reste de l'interface.

```
# Ce fichier DOIT s'appeler ValidationMessages.properties
# (avec éventuellement les suffixes pour fr, en...)
# et NE DOIT PAS se trouver dans un package.

erreur.longueur=Bad length
erreur.prenom=Bad surname
```

Conversions: phase « update model values »

- Parcours de l'arbre des composants, de l'IHM interne par JSF
- Affectation des propriétés côté serveur
  - Des conversions sont peut-être à faire… JJ/MM/AAAA …
  - Et inversement ….
- Deux fonctions : String -> Object et Object -> String
  - Bijection réciproque ?
- Mise à jour des propriétés du Bean
- Conversion standard
- Ou personnalisée

Exemple de convertisseurs standards

```
<h:inputText value="#{demoConverterBean.dateEtHeure}">
                <f:convertDateTime/>
</h:inputText>
```

```
<h:inputText value="#{demoConverterBean.valeur}">
      <f:convertNumber pattern="#000.00" />
</h:inputText>
```

Format d'affichage

Conversion personnalisée

```
@FacesConverter("hexConverter")
public class HexConverter implements Converter{

    @Override
    public Object getAsObject(FacesContext context, UIComponent
component, String value) {
        return Integer.parseInt(value, 16);
    }

    @Override
    public String getAsString(FacesContext context, UIComponent
component, Object value) {
        Integer i= (Integer) value;
        return Integer.toHexString(i);
    }
}
```

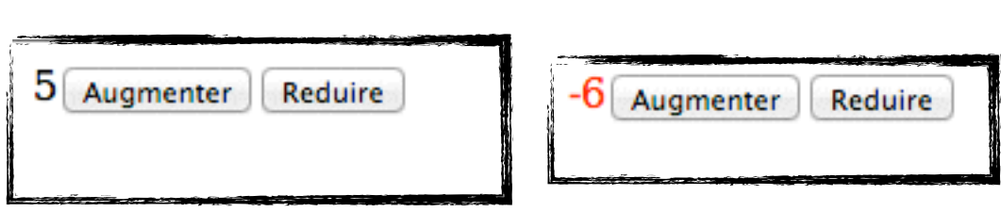
Conversion personnalisée (utilisation)

le champ affichera et saisira la représentation hexadécimale de **valeur**.

```
<h:form>
  <h:panelGrid columns=« 1">
      <h:inputText value="#{demoConverterBean.valeur}">
          <f:converter converterId="hexConverter"/>
      </h:inputText>
    <h:outputText value="#{demoConverterBean.affichage}"/>
        <h:commandButton value="faire"
                         action="#{demoConverterBean.faire()}"/>
  </h:panelGrid>
</h:form>
```

Bindings

- Permet d'avoir accès à toutes les caractéristiques d'un composant dans un Managed Bean, et pas seulement à sa valeur.
- Exemple : un composant qui change de couleur selon la valeur qu'il affiche




```
<h:form>
<h:outputText binding="#{compteurBinding.component}"
value="#{compteurBinding.value}"/>
<h:commandButton action="#{compteurBinding.augmenter()}"
value="Augmenter"/>
<h:commandButton action="#{compteurBinding.reduire()}"
value="Reduire"/>
</h:form>
```

Bindings (2)

```
@Named(value = "compteurBinding")
@SessionScoped
public class CompteurBinding implements Serializable{
    private int value= 0;
    private HtmlOutputText component;

    public int getValue() {return value;}

    public void augmenter() {
          value++;
          if (value <0) {
              component.setStyle("color: red");
          } else {
              component.setStyle("color: black");
          }
    }
  …
    public void setComponent(HtmlOutputText component) {
          this.component = component;
    }

    public HtmlOutputText getComponent() {return component;}
}
```

Événements

- Utilité : les gestionnaires d'événement ont plus d'information sur l'interface que les « actions » normales
- Ils sont appelés **avant** les actions
- Utilisation possible: préparer le terrain pour une action (par exemple: action « éditer » dans une liste; le listener peut déterminer quelle entrée éditer).
- Autre utilisation: modifier les éléments de ***l'interface***
- Souvent l'utilisation d'ajax est une alternative
- Exemple : http://docs.oracle.com/javaee/7/tutorial/doc/jsf-page-core002.htm

ActionListener

L'interface ActionListener

```
    public abstract processAction(ActionEvent ae);
```

```
class UserActionListener implements ActionListener{

  public processAction(ActionEvent ae)
                    throws AbortProcessingException{
   // accès au contexte, aux paramètres …
   FacesContext ctxt = FacesContext.getCurrentInstance();
   UICommand cmd = (UICommand) ae.getComponent();
   …
   …

    }
}
```

ValueChangeListener

```
<h:commandButton id= "incrementButton"   value="save" action= "#{SaveBean.save}"/>
  <f:valueChangeListener type="UserSaveListener"  />
</h:outputLink>

class UserSaveListener implements ValueChangeListener{

  public processValueChange(ValueChangeEvent vce)
                        throws AbortProcessingException{
    // accès au contexte, aux paramètres …
   FacesContext ctxt = FacesContext.getCurrentInstance();
    …
}
```

Résumé: événements

Actions <h:commandButton action="#{bean.submit}"/>
Action Listener <h:commandButton actionListener="#{bean.listen}" action="#{bean.submit}"/>
Value Change Listeners

```
<h:selectOneMenu value="#{bean.type}" onchange="submit()" immediate="true"
                 valueChangeListener="#{bean.change}">
    <f:selectItems value="#{bean.types}"/>
 </h:selectOneMenu>
```

- listeners : généralement pour des modifications qui portent sur l'interface

- immediate='true' : tout le traitement du composant est effectué lors de la phase « apply request values ».

Templates

- Système permettant de factoriser la présentation du site.
- les jsf précisent quelles templates elles utilisent, et quelle partie de la template elles remplacent.

La template

template.xhtml

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">

    <h:head>
    </h:head>

    <h:body>
        <div id="top" class="top">
            <ui:insert name="top">My Nice forum</ui:insert>
        </div>
        <div>
             <div id="content" class="left_content">
                 <ui:insert name="content">Content</ui:insert>
             </div>
        </div>
    </h:body>
</html>
```

ui:insert définit une zone dont le contenu peut être remplacé par une facelet

Utilisation d'une template

```
<?xml version='1.0' encoding='UTF-8' ?>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:ui="http://java.sun.com/jsf/facelets">

    <ui:composition template="template.xhtml">
        <ui:define name="content">
            <h:form>
                <h:panelGrid columns="2">
                    <h:outputText value="Login:"/>
                    <h:inputText value="#{login.login}" title="Login"/>
                    <h:outputText value="Password:"/>
                    <h:inputSecret value="#{login.password}" title="Password"/>
                </h:panelGrid>
                <h:commandButton type="submit" action="#{login.doLogin()}"
label="login" value="login"/>
            </h:form>
            <h:messages/>
        </ui:define>
    </ui:composition>
</html>
```

utiliser le fichier template.xhtml

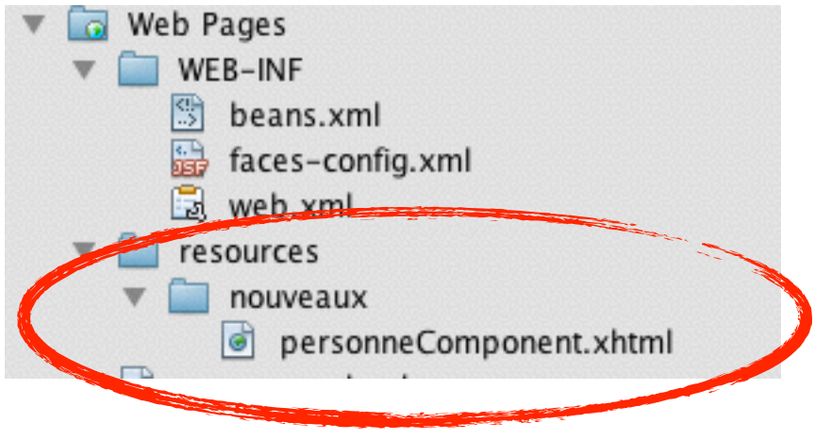
dont on redéfinit la partie « content »

Composites

- Création assez simple de « composants » à partir des composants existants

- À la racine de l'appli, créer un dossier « resources », et un sous dossier pour y placer les composites

- Écrire le composite dans ce dossier (comme une jsf)



Composite

personneComponent.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:cc="http://java.sun.com/jsf/composite"
      xmlns:h="http://java.sun.com/jsf/html">

    CETTE TEXTE N'EST PAS UTILISÉ….
    <!-- INTERFACE -->
    <cc:interface>
        <cc:attribute name="nom" required="true"/>
        <cc:attribute name="prenom" required="true"/>
    </cc:interface>

    <!-- IMPLEMENTATION -->
    <cc:implementation>
        <h:outputLabel value="Nom"></h:outputLabel>
        <h:inputText value="#{cc.attrs.nom}"></h:inputText>
        <h:outputLabel value="Prénom"></h:outputLabel>
        <h:inputText value="#{cc.attrs.prenom}"></h:inputText>
    </cc:implementation>
</html>
```

attributs de la nouvelle balise « personneComponent »

nom des attributs précédés de cc.attrs

Composites (utilisation)

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:nouv="http://java.sun.com/jsf/composite/nouveaux">
    <h:head>
        <title>Facelet Title</title>
    </h:head>
    <h:body>
        <h:form>
        <nouv:personneComponent nom="toto" prenom="titi"/>
        </h:form>
    </h:body>
</html>
```

« nouveaux »= dossier dans resources

Bibliothèques de composants JSF

- http://www.primefaces.org/
- ICEFaces
- RichFaces

PrimeFaces

- Très Nombreux champs supplémentaires : ColorPicker, Calendar, TagCloud…
- Panneaux, graphiques, multimédia, gestion de fichiers…
- Gestion par JSF d'images dynamiques (en JSF pures, gérées par… une servlet).
- Extension de la spécification : langage plus riche pour désigner des éléments,

Gag…

- La *plupart* des portées existent en deux versions distinctes:
  - javax.faces.bean.SessionScoped
  - vs javax.enterprise.context.SessionScoped
- Pour que ça fonctionne:
  - Si on utilise l'annotation @ManagedBean, utiliser javax.faces.bean
  - Si on utilise l'annotation @Named, utiliser javax.enterprise.context
- @Named= utilisation de CDI (injection de dépendance générale) ;
- @Managed = injection spécifique JSF
- @Named permet d'utiliser @ConversationScoped. @Managed permet d'utiliser @ViewScoped (résolu avec **JSF 2.2** et javax.faces.view.ViewScoped. Notez bien le package…)