JSF 2.0 (*Java Server Faces*) Partie 3

Université de Nice - Sophia Antipolis
Richard Grin
Version 0.9.1 – 18/11/12

- *Avertissement : les supports sur JSF sont en cours d'écriture. Il peuvent comporter des pages vides ou des points d'interrogation (le plus souvent uniquement à cause du manque de temps, pas d'un problème réel)*

- Les 2 premières parties de ce support ont présenté les notions de base de JSF
- Cette dernière partie du cours JSF présente plus en détails la programmation Java dans les backing beans et les aspects plus avancés de JSF, tel que l'écriture de composants personnalisés

Plan (1)

Plan (2)

Classes Java de base

FacesContext

- Classe de base pour la programmation
- Interlocutrice principale du développeur
- Permet d'obtenir le contexte de l'application

FacesContext

- Les backing bean utilisent des classes JSF de base pour interagir avec les pages JSF
- L'instance courante de **`FacesContext`** contient toutes les informations liées au traitement d'une requête ; elle peut être récupérée par le code **`FacesContext.getCurrentInstance()`**
- A partir de cette instance, qui sera appelée *facesContext* dans les transparents suivants, on peut récupérer des informations sur la requête et en modifier certaines (les messages par exemple)

Exemple de code

- Changer la locale de la vue en cours :

```
FacesContext facesContext =
  FacesContext.getCurrentInstance();
facesContext.getViewRoot().setLocale(locale);
```


Utilisation directe de FacesContext – renderResponse()

- La méthode renderResponse() fait passer tout de suite à la phase « Render response » pour envoyer la réponse au client en sautant les phases suivantes du cycle de vie
- Elle peut être utilisée, par exemple, dans un listener déclenché par des composants qui ont l'attribut immediate="true" pour éviter de convertir/valider les autres composants
- Exemple tiré du livre « Core JSF » : une liste permet de changer la langue de la page

Exemple

- Page JSF :

```
<h:selectOneMenu value=… onChange="submit()"
    immediate="true"
    valueChangeListener="#{bean.change}">
  ...
```

- Méthode **change** du backing bean :

```
public void change(ValueChangEvent event) {
  FacesContext context = ...;
  // Modifier la locale
  ...
  context.getViewRoot().setLocale(...);
  context.renderResponse();
}
```


Utilisation directe de FacesContext – responseComplete()

- La méthode responseComplete() indique à JSF qu'il ne faut pas continuer le cycle de vie normal
- Elle peut être utilisée dans les cas où la réponse au client est générée par des moyens indépendants de JSF
- Par exemple, si l'utilisateur clique sur un bouton pour obtenir l'exportation des données d'une datatable au format d'un tableur, les données binaires sont générées par la méthode action du bouton et responseComplete est appelée à la fin

Exemple de méthode action du bean

```
HttpServletResponse response =
  (HttpServletResponse) FacesContext
  .getCurrentInstance().getExternalContext()
  .getResponse();
byte[] donneesTableur = genTableau.getDonnees();
response.setContentType("application/vnd.ms-excel");
response.setContentLength(excelData.length);
response.setHeader("Content-Disposition",
              "attachment; filename=\"data.xls\"");
OutputStream out = response.getOutputStream();
out.write(donneesTableur);
out.flush();
FacesContext.getCurrentInstance().responseComplete();
```


`ExternalContext`

- **`FacesContext`** permet en particulier de récupérer l'environnement d'exécution de la requête, instance de **`ExternalContext`** : *facesContext*`.getExternalContext()`
- Le code peut ainsi être rendu indépendant de cet environnement d'exécution, que ce soit un environnement « Servlet » ou « Portlet » (ce dernier n'est pas étudié dans ce cours)
- Cette instance sera appelée *externalContext* dans les transparents suivants

Informations de `ExternalContext`

- *externalContext* permet de récupérer de nombreuses informations sur l'environnement : cookies, requête en cours, paramètres d'initialisation de l'application, locale, nom et rôle de l'utilisateur (pour la sécurité), …
- Il permet aussi d'interagir avec les différentes maps qui correspondent aux différentes portées (application, session, requête,…), de donner l'encodage de la réponse, d'enregistrer des messages dans les journaux (log), etc.

Des méthodes de `ExternalContext`

- Login de l'utilisateur (s'il s'est authentifié) :
  `String getRemoteUser()`
- Retourne **`true`** si l'utilisateur a le rôle passé en paramètre :
  `boolean isUserInRole(String role)`
- URL d'une ressource définie par un chemin qui commence par « / » :
  `URL getResource(String chemin)`
- Contexte de l'application (la portion de l'URI qui identifie l'application ; par exemple « /appliweb ») :
  `String getRequestContextPath()`

Des méthodes de `ExternalContext`

- Requête en cours (doit le plus souvent être casté en HttpServletRequest) :
  `Object getRequest()`
- Réponse en cours (caster en HttpServletResponse) :
  `Object getResponse()`
- Valeur d'un paramètre d'initialisation :
  `String getInitParameter()`
- Récupère la mémoire Flash : `Flash getFlash()`
- Chemin réel (de la machine) d'un chemin virtuel (celui de l'application Web) :
  `String getRealPath(String path)`

Des méthodes de `ExternalContext`

- Les maps des différentes portées (pour les attributs) : `getRequestMap()`, `getSessionMap()`, `getApplicationMap()`
- La map des paramètres d'initialisation :
  `getInitParameterMap()`
- La map des paramètres de la requête :
  `getRequestParameterMap()`
- La map des headers de la requête :
  `getRequestHeaderMap()`
- La map des noms des cookies de la requête :
  `getRequestCookieMap()`

Exemple

```
ExternalContext extContext =
  facesContext.getExternal();
HttpServletRequest response =
  (HttpServletRequest)extContext.getResponse();
response.sendRedirect(url);
```


Forward et redirection

- Dans une méthode « action » d'un backing bean
- Forward :
  `externalContext.dispatch("/faces/page1");`
- Redirection :
  `externalContext.redirect("/appli/faces/page1");`
  (si sur même serveur Web)
  `externalContext.redirect("h://serveur.unice.fr/appli/faces/page1");`
  (si sur un autre serveur Web)

Gestion des erreurs

2 types d'erreurs

- Une exception Java peut arriver
  - durant l'exécution d'une méthode « action » ou d'une méthode appelée par une méthode action
  - durant l'exécution d'une méthode qui récupère les données à afficher dans une page
  - durant l'exécution d'une méthode « écouteur »
- Une erreur peut aussi être liée à HTTP ; par exemple lorsqu'une page demandée n'existe pas ou que son accès n'est pas autorisé pour l'utilisateur

Exception dans une méthode action

- Le plus souvent l'exception est attrapée par la méthode qui dirige vers une page appropriée (par exemple qui affiche une message d'erreur à l'utilisateur)

Configuration pour les exceptions

- Le fichier web.xml peut comporter des balises <error-page> pour la configuration des exceptions
- Une balise <error-page> permet de désigner une page JSF à afficher si une exception Java non traitée a été rencontrée ou si une erreur HTTP est survenue
- Le navigateur n'est pas conscient qu'une autre page que la page qui a provoqué l'erreur est affichée et il affiche donc son URL plutôt que la page d'erreur, ce qui peut être un problème

Rappel codes erreurs HTTP

- Une réponse du serveur HTTP contient un code
- Si tout s'est bien passé, le code est compris entre 200 et 299
- Le code compris entre 300 et 399 correspond à une redirection
- Le code compris entre 400 et 499 correspond à une erreur du client (par exemple, syntaxe requête erronée ou accès refusé)
- Le code 500 (ou plus) correspond à une erreur du serveur (par exemple erreur interne)

Exemples

```
<error-page>
    <error-code>400</error-code>
    <location>faces/exception.xhtml</location>
</error-page>
```

```
<error-page>
    <exception-type>java.lang.Exception</exception-type>
    <location>faces/exception.xhtml</location>
</error-page>
```


Ecriture page d'erreur

- Lorsqu'une exception arrive jusqu'au servlet « faces », celui-ci ajoute des attributs (de type String) à la requête
- La page d'erreur peut les utiliser pour afficher des informations sur l'erreur
- Les noms des attributs commencent tous par javax.servlet.error, avec l'un des suffixes suivants : .status_code (code erreur HTTP), .exception (classe Java d'exception), .request_uri (URL demandée au départ), .servlet_name (servlet qui traitait la requête quand l'erreur a eu lieu)

Exemple de page d'erreur

- Extrait de la page :

```
Vous n'êtes pas autorisé à voir la page
#{visiteur.uriPasAutorise()}
```

- Extrait du backing bean « **visiteur** » :

```
public String uriPasAutorise() {
  HttpServletRequest requete =
    (HttpServletRequest)
    FacesContext.getCurrentInstance()
    .getExternalContext().getRequest();
  return requete.getAttribute
    ("javax.servlet.error.request_uri");
}
```


Exceptions Java JSF

- Les exceptions JSF sont des exceptions non contrôlées par le compilateur
- Elles héritent javax.faces.FacesException qui est une classe fille de RuntimeException

Types d'exception

- **ConverterException** : indique qu'une conversion de type n'a pu avoir lieu (voir convertisseurs)
- **ValidatorException** : indique qu'une valeur n'a pas été validée (voir validateurs)
- **ViewExpiredException** : indique qu'une vue n'a pu être restaurée (souvent à cause d'un timeout pour la session) ; exception fréquente si la durée de la session est courte

Gérer **`ViewExpiredException`**

- On peut faire afficher une certaine page pour gérer au mieux cette situation
- Il suffit d'écrire ce type de code dans web.xml :

```
<error-page>
  <exception-type>
    javax.faces.application.ViewExpiredException
  </exception-type>
  <location>/faces/vePage.xhtml</location>
</error-page>
```


Événements

Utilité

- Avec les événements, JSF essaie de reproduire la programmation des interfaces utilisateurs non Web (type Swing)
- Des événements sont générés par JSF et des écouteurs peuvent réagir à ces événements

Types d'événements

- 2 grands types d'événements :
  - Générés par des actions de l'utilisateur (de type « application »)
  - Générés à certains moments du cycle de vie (de type « lifecycle »)

Événements « application »

- ValueChangeEvent : lorsque la valeur d'un composant a été modifié par l'utilisateur (champ de saisie de texte, menu, liste déroulante,…) ; liés aux composants de type <h:selectOneMenu> ou <h:inputText>
- ActionEvent : provoqué par un clic sur un bouton ou un lien (ce qui peut provoquer un changement de page) ; liés aux composants « UICommand » tels que les <h:commandButton> ou <h:commandLink>

Événements « cycle de vie »

- PhaseEvent : provoqué par un changement de phase du cycle de vie. Par exemple, APPLY_REQUEST_VALUES
- Système (il y a beaucoup d'événement de ce type) : provoqué par divers « sous-phases » du cycle de vie ; introduit par JSF 2.0. Par exemple, PreRenderComponentEvent ou PostValidateEvent

Événements système

- Les plus utilisés :
  - PreRenderViewEvent
  - PreValidateEvent et PostValidateEvent
- Plusieurs autres :
  - PostConstructApplicationEvent et PreDestroyApplicationEvent
  - PostAddToViewEvent et PreRemoveToViewEvent
  - PostRestoreStateEvent
  - PreRenderComponentEvent
  - ExceptionQueueEvent
  - …

Écouteur

- Les événements sont traités par des écouteurs
- Un écouteur peut être une méthode (attribut xxxListener) ou une classe (sous-balise <f:xxxListener>)

Lors du changement d'une valeur

- Un événement **`ValueChangeEvent`** est généré lorsqu'une valeur est modifiée dans un formulaire, par exemple dans une balise **`<h:inputText>`**
- Cet événement peut être traité par une méthode déclarée dans un attribut **`valueChangeListener`** ou par une classe déclarée dans une sous-balise **`<f:valueChangeListener>`**

Méthode pour écouter

- Les méthodes déclarées comme écouteur d'un changement de valeur doivent prendre un **`ValueChangeEvent`** en paramètre et ne rien renvoyer

Exemple

Classes d'événements

- Les événements sont représentés par des classes du paquetage **`javax.faces.event`** qui héritent de la classe **`FacesEvent`** du même paquetage
- Cette classe contient la méthode **`getSource()`** qui renvoie le **`UIComponent`** qui est la source de l'événement

Traitement d'un action event

- Les événements de type « action » sont générés lors de la phase « invoke application » du cycle de vie
- Ils peuvent être traités par une méthode « action listener » (attribut **`actionListener`** du composant source) ou par une méthode « action » (attribut **`action`** du composant source)
- Les méthodes « action listener » sont exécutées avant les méthode « action »

Traitement d'un action event

- Si le traitement n'induit pas de navigation vers une autre page, les action listeners et les actions sont équivalentes (à part le fait que les actions ne savent pas grand-chose de l'interface utilisateur car elles n'ont aucun paramètre)
- Pour certains traitement il est possible de combiner des action listeners (qui ont des informations sur les composants de l'interface utilisateur) et une action (qui peut effectuer une navigation vers une autre page)

Attacher des action listeners

- Il est possible d'attacher plusieurs listeners à un composant
- L'attribut actionListener du composant ne permet d'attacher qu'un seul listener
- Des balises incluses <f:actionListener> permettent d'en attacher plusieurs :

```
<h:commandButton … listener="#{bb.ml}"
                   action="#{bb.m1}">
  <f:actionListener type="p1.X1Listener" />
  <f:actionListener type="p1.X2Listener" />
</h:commandButton>
```


Ordre d'exécution pour les action events

1. La méthode indiquée par l'attribut actionListener
2. Ensuite les méthode processAction des listeners définie par les balises incluses <f:actionListener>
3. Finalement la méthode « action » définie par l'attribut action (ordre d'exécution à vérifier**??**)

`actionListener` ou **`action`** ?

- Les 2 attributs permettent de traiter un ActionEvent
- action lance un traitement métier (en tout cas, pas seulement lié à l'interface utilisateur) et elle permet de changer de page à la suite de l'exécution du code
- actionListener sert pour les traitements liés à l'interface utilisateur (par exemple, afficher les personnes d'un département qui vient d'être choisi par l'utilisateur)

`actionListener` ou **`action`** – signatures des méthodes

- La méthode désignée par action ne prend pas de paramètre et renvoie une chaîne qui indique la page suivante à afficher ; le plus souvent la méthode appelle une méthode d'un EJB pour lancer un processus métier
- La méthode désignée par actionListener ne renvoie rien (void) et prend un paramètre ActionEvent qui donne accès au composant qui a engendré l'événement ; on a ainsi plus de prise sur l'interface utilisateur

Récupérer un composant

- À partir d'un événement on peut récupérer le composant source par

```
UIComponent source = event.getComponent();
```

- De là on peut aussi récupérer les autres composants qui sont dans le même formulaire que le composant source, par exemple,

```
UIInput mdp =
  (UIInput)source.findComponent("motDePasse");
```


Récupérer un composant

- Une autre façon de pouvoir utiliser un composant dans le code Java d'un backing bean est de lier le composant à une propriété du backing bean par l'attribut binding :

```
<h:inputText binding="#{bb.composant1}" …/>
```

- Le code Java contiendra

```
private UIComponent composant1;
public UIComponent getComposant1() {
  return composant1;
}
public void setComposant1(UIcomponent c) {
  this.composant1 = c;
}
```


Exemples d'utilisations des événements système

- Valider un groupe de composants comme 2 champs de mot de passe ou une date décomposée en année, mois, jour (événement de type postValidate)
- Effectuer une action juste avant l'affichage d'une page JSF (type preRenderView) ; par exemple, récupérer des données dans une base de données pour les afficher dans la page, ou naviguer vers une autre page que la page normale

Cas d'utilisation pour **`postValidate`**

- Il faut vérifier que l'utilisateur a bien taper 2 fois le même mot de passe ; on ne sait pas quel mot de passe va être saisi en dernier
- Un listener **`postValidate`** convient parfaitement ici : il sera exécuté après les autres validations

Schéma pour **`postValidate`**

- Placer un

```
<f:event type="postValidate"
              listener="#{bb.valider}"  />
```

dans le composant le plus proche qui contient les 2 mots de passe

- Ecrire une méthode valider dans le backing bean bb :

```
public void valider(ComponentSystemEvent ev) {
   ... // voir transparent suivant
}
```


Listener pour **`postValidate`**

```
public void valider(ComponentSystemEvent ev) {
  UIComponent source = ev.getComponent();
  UIInput mdp1 = (UIInput)source.findComponent("mdp1");
  UIInput mdp2 = (UIInput)source.findComponent("mdp2");
  String smdp1 = mdp1.getLocalValue();
  String smdp2 = mdp2.getLocalValue();
  ...// Comparaison de smdp1 et de smdp2
  if (pasEgaux) {
    FacesMessage message = ...
    FacesContext context =
      FacesContext.getCurrentInstance();
    context.addMessage(source.getClientId(), message);
    context.renderResponse();
  }
}
```

Autre exemple pour **postValidate**

- Validation multi-champs en utilisant l'événement système « postValidate », par exemple pour valider une date donnée par 3 champs séparés année, mois et jour
- Mettre les 3 champs dans un **<h:gridPanel>** et à l'intérieur du gridPanel une balise f:event :

```
<f:event type="postValidate"
         listener="#{bean.validerDate}"/>
```


Cas d'utilisation pour **preRenderView**

- Les transparents suivants donnent un schéma pour faire afficher les détails sur un client en cliquant sur un bouton dans la ligne d'une table qui affiche une liste de clients
- Ce schéma « PRG » permet de pouvoir garder un marque-page sur la page du client pour pouvoir éventuellement la réafficher directement plus tard
- Un preRenderView est utilisé pour afficher les informations sur un client à partir de son id

Schéma pour **preRenderView**

- Une liste de clients est affichée dans une dataTable :

```
<h:dataTable value="#{clientB.clients}"
var="item" …>
```

- Chaque ligne de la table contient un lien pour afficher les détails sur le client de la ligne :

```
<h:commandLink
action="#{clientB.details(item)}" value=…/>
```

- Le code de la méthode **details** dans le bean :

```
public String showDetails(Client client) {
  return "detailsClient?id=" + client.getId()
         + "&faces-redirect=true";
}
```


Page qui affiche les détails d'un client

```
  <f:metadata>
    <f:viewParam
       name="id" value="#{clientB.idDetails}"/>
  </f:metadata>
</head>
<h:body>
  <f:view>
    <f:event type="preRenderView"
        listener="#{clientB.lireClient}"/>
  ...
    #{clientB.client.nom}
```


Code de la méthode lireClient

```
public void
    lireClient(ComponentSystemEvent event) {
  this.client =
    clientManager.findById(idDetails);
}
```

Bean session « DAO » pour récupérer un client à partir de son id

Code pour naviguer vers une page

- L'écouteur peut faire naviguer vers une autre page (par exemple si le login n'est pas correct) :

```
if (loginIncorrect) {
  FacesContext context =
    FacesContext.getInstance();
  ConfigurableNavigationHandler handler =
    context.getApplication()
      .getNavigationHandler();
  handler.performNavigation("login");
}
```

Evénements de phase

- Pour définir un listener de phase, il faut
  - soit ajouter une balise **`<f:phaseListener>`** n'importe où dans une page :
    `<f:phaseListener type="classe écouteur"/>`
  - soit l'indiquer dans un fichier de configuration de JSF (le plus souvent **`faces-config.xml`** de **`WEB-INF`**) :

```
<lifecycle>
  <phase-listener>classe écouteur
  </phaseListener>
</lifecycle>
```


Exemple d'écouteur de phases

```
public class MonPhaseListener
       implements PhaseListener {
  public void afterPhase(PhaseEvent event) {
    System.out.println("Après "
                       + event.getPhaseId());
  }
  public PhaseId getPhaseId() {
    // ou, par exemple, PhaseId.INVOKE_APPLICATION
    return PhaseId.ANY_PHASE;
  }
  public void beforePhase(PhaseEvent event) {
    ...
  }
}
```

Indique quand délivrer un événement de phase

Échanges de valeurs entre l'interface utilisateur et le code Java

Passer une valeur de l'interface vers le code Java

- Plusieurs façons de passer des valeurs
1. Le plus simple est de passer un paramètre à une méthode Java appelée par l'interface
2. On peut aussi utiliser les paramètres des composants
3. Ou les attributs des composants
4. L'interface peut aussi donner directement la valeur d'une propriété d'un bean

1. *Method expression parameters*

- Depuis JSF 2.0, il est possible de passer directement les valeurs des paramètres de méthodes

Exemple

- Dans l'interface :

```
<h:commandLink
    action="#{bean.methode('val')}"
```

- Méthode Java du bean :

```
public String methode(String param) {
```

Autre exemple (1/2)

- Dans une table JSF, chaque ligne de la colonne contient un lien (ou un bouton) qui permet de supprimer la ligne du modèle utilisé par la table :

```
<h:datatable value="#{bean.personnes}"
    var="pers" ...>
  ...
  <h:column>
    ...
    <h:commandLink value="Supprimer"
      action="#{bean.suprPersonne(pers)}"/>
    ...
```


Autre exemple (2/2)

- La méthode du backing bean :

```
public void supprPersonne(Personne pers) {
  personnes.remove(pers);
}
```

- Remarque : en interne JSF utilise le numéro de la ligne de la table pour récupérer l'objet Personne ; cette méthode risque donc de ne pas fonctionner si le modèle de la table est modifié entre le moment de l'affichage de la table et le moment où la requête de suppression arrive sur le serveur. En ce cas il faut utiliser une autre méthode, par exemple un paramètre ou setPropertyActionListener

2. Paramètre de composant

- Un paramètre peut être ajouté à un composant par une balise <f:param> (voir exemple)
- Chaque type de composant interprète ses paramètres à sa façon
- Un paramètre d'un bouton ou d'un lien est transformé en paramètre de la requête
- Le code Java peut alors récupérer la valeur (la 1ère valeur s'il y en a plusieurs) du paramètre par la méthode **`Map<String, String> getRequestParameterMap()`** de la classe **`javax.faces.context.ExternalContext`**

Récupérer toutes les valeurs

- Un composant peut avoir plusieurs valeurs pour un même paramètre
- Si on veut récupérer toutes les valeurs d'un paramètre qui en a plusieurs, il faut utiliser la méthode **`getRequestParameterValuesMap()`** de la classe **`ExternalContext`** qui retourne une valeur de type **`Map<String, String[]>`**

Exemple

- Dans l'interface :

```
<h:commandLink …
    action="#{bean.methode}">
    <f:param name="param" value="val"/>
</h:commandLink>
```

- Méthode Java du bean :

```
public String methode() {
  …
  String param =
    FacesContext.getCurrentInstance().
    getExternalContext().
    getRequestParameterMap().get("param");
```


3. Attribut de composant

- Il est possible d'ajouter une propriété à un composant avec la balise **`<f:attribute>`**
- Dans le code Java il faut ensuite commencer par récupérer le composant (ce qui impose d'avoir une référence vers l'événement, donc une méthode désignée par l'attribut **`actionListener`** et pas **`action`**), puis la propriété du composant par **`composant.get`*`NomAttribut`*`()`**

Exemple

- Dans l'interface :

```
<h:commandLink …
    actionListener="#{bean.methode}">
    <f:attribute name="prop" value="val"/>
</h:commandLink>
```

- Méthode Java du bean :

```
public void methode(ActionEvent event) {
  …
  UIComponent composant =
    event.getComponent();
  String valeur = composant.getProp();
```


4. setPropertyActionListener

- La balise setPropertyActionListener permet de placer directement une valeur dans la propriété d'un bean
- Comme les actionListener sont exécutés avant les méthodes action, cette balise permet de donner des valeurs aux propriétés du bean qui sont utilisées par les méthodes action

Exemple

- Dans l'interface :

```
<h:commandLink ...
    action="#{bean.methode}">
    <f:setPropertyActionListener
      target="#{bean.prop}" value="val"/>
</h:commandLink>
```

- Méthode Java du bean :

```
private String prop;
...
public void methode(ActionEvent event) {
  ...
  String valeur = this.getProp();
```


Autre exemple (1/2)

- On reprend l'exemple (solution 1) qui permet de supprimer une ligne d'une table
- Dans la page JSF :

```
<h:datatable value="#{bean.personnes}"
    var="pers" ...>
  ...
  <h:commandLink value="Supprimer"
      action="#{bean.suprPersonne}">
    <f:setPropertyActionListener
      target=#{bean.idPers}
      value=#{pers.id}/>
  </h:commandLink>
```


Autre exemple (2/2)

- Dans le backing bean :

```
private int idPers;
...
public void supprPersonne() {
  // Supprime la personne qui a comme
  // identifiant idPers
  ...
}
```


Conversion

Conversion de type

- Quand le serveur reçoit une requête POST, les valeurs saisies par l'utilisateur arrivent sous la forme de chaînes de caractères
- Ces chaînes doivent souvent être traduites en objets Java
- Par exemple, si un candidat à une formation a indiqué sa nationalité par une liste déroulante, ce choix doit être traduit en un objet **`Pays`** qui sera associé à l'objet **`Candidat`** qui représente l'utilisateur

Utilité des convertisseurs

- Cette conversion peut être faite par le code Java qui récupère la chaîne de caractères qui correspond au choix de la liste
- Si ce type de traitement apparaît à plusieurs endroits, il est plus simple de factoriser le code dans un convertisseur qui va convertir une chaîne de caractères dans une instance de la classe Pays

Conversion implicite

- JSF fait une conversion automatique des types primitifs, **`BigDecimal`** et **`BigInteger`**
- Par exemple, le développeur n'a pas besoin d'écrire quoi que ce soit pour que la valeur saisie par l'utilisateur soit enregistrée dans une propriété de type **`int`** du backing bean
- Si la valeur saisie ne peut être convertie dans le type voulu, un message d'erreur standard sera enregistré dans la file d'attente des messages et ensuite affiché à l'utilisateur (voir section « Messages d'erreur et d'information »)

Désigner un convertisseur

- Lorsqu'un composant a besoin d'un convertisseur, on peut le désigner
  - par une sous-balise spéciale pour les convertisseurs standards
  - par une sous-balise **`<f:converter>`** en donnant l'identificateur du convertisseur : <f:converter converterId="xxx">
  - par l'attribut converter du composant <h:outputText value="xxx" converter="yyy"/>
  - en définissant un convertisseur par défaut pour le type de la donnée à convertir

Exemple

```
<h:selectOneMenu
    value="#{myBean.selectedItem}">
  <f:selectItems
      value="#{myBean.selectItems}" />
  <f:converter converterId="fooConverter" />
</h:selectOneMenu>
```


Convertisseurs standards

- JSF fournit des convertisseurs pour les types primitifs, pour les dates et pour les classes **`BigInteger`** et **`BigDecimal`**
- Des balises spéciales sont fournies par JSF pour les convertisseurs standards mais on peut aussi utiliser la balise **`<f:convert>`** en donnant l'identificateur du convertisseur standard
- Par exemple, **`<f:convertDateTime/>`** est équivalent à **`<f:convert convertId="javax.faces.DateTime"/>`**

Exemples

- `<inputText value="#{employe.salaire}">`
  `<f:convertNumber type="currency"/>`
  `</inputText>`
- `<outputText value="#{employe.salaire}">`
  `<f:convertNumber type="currency"`
  `currencyCode="EUR" />`
  `</outputText>`
  (code ISO 4217 pour le code de la devise)
- `<inputText value="#{employe.dateNaissance}">`
  `<f:convertDateTime`
  `pattern="dd/MM/yyyy"/>`
  `</inputText>`

Convertisseur personnalisé

- Le développeur doit écrire les convertisseurs associés aux types qui ne sont pas pris en charge par les convertisseurs standards de JSF (*custom converter*)
- Il faut
  - écrire la classe Java du convertisseur
  - indiquer quand utiliser le convertisseur

Écrire un convertisseur

- Classe Java qui implémente l'interface **`Converter`** (paquetage **`javax.faces.convert`**)
- Cette interface contient les 2 méthodes
  - **`Object getAsObject(FacesContext, UIComponent, String)`**
    qui transforme une chaîne de caractères en objet
  - **`String getAsString(FacesContext, UIComponent, Object)`**
    qui transforme un objet en chaîne de caractères

En cas de problème

- Si la conversion ne peut être faite, le convertisseur
  - doit lancer une **`javax.faces.convert.ConverterException`** ce qui provoquera le réaffichage de la page courante
  - ajouter un message d'erreur (**`FacesMessage`**) qui sera affiché dans la page courante ; voir section sur les messages d'erreur plus loin sur ce support

Utiliser un EJB dans le convertisseur

- Comme pour un validateur, il n'est pas possible d'utiliser **`@Inject`** dans un convertisseur
- Mais on peut écrire ce type de code pour utiliser un EJB ou une ressource :

```
try {
   ic = new InitialContext();
   myejb= (MyEJB) ic
     .lookup("java:global/xxxx/MyEJB");
} catch (NamingException e) {
   …
}
```


Désigner un convertisseur

- Lorsqu'un composant doit utiliser un convertisseur, si aucun convertisseur particulier n'est désigné par son identificateur c'est le convertisseur par défaut du type qui est utilisé
- Pour désigner un convertisseur particulier ou un convertisseur par défaut d'un type, le développeur peut utiliser une annotation ou écrire une configuration dans le fichier faces-config.xml

Annotation @FacesConverter

- La classe du convertisseur doit être annotée (ou la classe doit être indiquée dans le fichier faces-config.xml comme dans un transparent suivant)
- L'attribut par défaut donne un identificateur : **`@FacesConverter("fr.unice.Pays")`**
- L'attribut **`forClass`** indique que le convertisseur est le convertisseur par défaut d'une classe : **`@FacesConverter(forClass=fr.unice.Pays.class)`**

Convertisseur par défaut

- Un convertisseur par défaut pour une classe sera utilisé pour convertir les données de cette classe si aucun autre convertisseur n'est explicitement désigné
- Une erreur est lancée si aucun convertisseur n'est donné par son identificateur et qu'aucun convertisseur par défaut n'existe pour le type

Exemple – liste

```
<h:selectOneMenu value="#{bean.compte}">
  <f:selectItems value="#{bean.allComptes}"
     var="compte"
     itemLabel="#{compte.nom}"/>
</h:selectOneMenu>
```


Exemple – backing bean

```
@Named(value = "ajoutRetrait")
public class Bean implements Serializable {
  @EJB
  private GestionnaireCompte ejb;
  private Compte compte;

  // getter et setter pour compte
  ...

  public List<Compte> getAllComptes() {
    return ejb.getAllComptes();
  }
```


Exemple – convertisseur (1/2)

```
@FacesConverter(forClass=Compte.class)
public class CompteConverter implements Converter {
  @Override
  public Object getAsObject(FacesContext context,
      UIComponent component, String value) {
    try {
      Context c = new InitialContext();
      GestionnaireCompte ejb = (GestionnaireCompte)
        c.lookup("java:global/appli/GestCompte");
      return ejb.getById(Long.parseLong(value));
    } catch (NamingException ne) {
    ...
    }
  }
```


Exemple – convertisseur (2/2)

```
  @Override
  public String getAsString(FacesContext context,
      UIComponent component, Object value) {
    Compte compte = (Compte) value;
    return Long.toString(compte.getId());
  }
```

Fichier faces-config.xml

- Donne un identificateur à la classe du convertisseur :

```
<converter>
   <converter-id>fr.unice.Pays</converter-id>
   <converter-class>fr.unice.PaysConverter</converter-class>
</converter>
```

- Indique la classe du convertisseur par défaut d'un type :

```
<converter>
   <converter-for-class>fr.unice.Pays</converter-for-class>
   <converter-class>fr.unice.Pays</converter-class>
</converter>
```


Balise personnalisée pour convertisseur

- Il est possible de créer de nouvelles balises pour avoir des convertisseurs qui ont des attributs comme, par exemple, l'attribut pattern de la balise <f:convertDateTime>
- Pour cela, comme pour toutes les balises personnalisées, il faut écrire un fichier de description de balises (voir section sur les balises et composants personnalisés) ou créer un fichier jar à part

Validation

Bean validation

- Si JSF est utilisé avec un serveur Java EE 6, il est possible d'utiliser la « bean validation » (JSR 303)
- Celle-ci permet d'annoter les entités pour donner des contraintes sur les valeurs acceptées pour les propriétés
- Par exemple, @NotNull impose une valeur non null
- Il existe des annotations standards mais il est aussi possible (et pas trop complexe) d'ajouter ses propres annotations

Annotations standard (1/2)

- Toutes les annotations standard ont 2 attributs :
  - message pour donner un message d'erreur pour le cas où la valeur ne serait pas validée
  - groups pour donner un groupe de validation (pas étudié ici ; voir documentation sur la validation de bean)

Annotations standard (2/2)

- @Null, @NotNull
- @Min, @Max, @DecimalMin, @DecimalMax ; float et double pas supportés
- @Digits (attributs integer et fraction) ; indique le nombre maximum de chiffres
- @Size (attributs min et max) ; taille d'une String, d'un tableau, d'une collection ou d'une map
- @Pattern (attributs regexp et flags)
- @AssertTrue, @AssertFalse
- @Past, @Future

Fonctionnement

- Comment et quand se passe la validation ?
- Que se passe-t-il si une valeur n'est pas validée ?

Validateur

- En dehors de la validation par les beans, JSF offre de nombreuses facilités pour valider les valeurs données par l'utilisateur :
  - Attribut required
  - Validateurs standards
  - Validateurs écrits pour l'application
- Les transparents suivants étudient ces facilités

Attribut required (1/2)

- Les composants standards ont l'attribut required qui indique que l'utilisateur doit fournir une valeur si sa valeur est true
- Attention, si l'attribut required n'a pas la valeur true et si l'utilisateur ne saisit rien dans un champ, aucune autre validation n'est faite et la valeur retournée est la chaîne vide

Attribut required (2/2)

- Pour faire retourner null à la place de la chaîne vide, écrire dans web.xml :

```
<context-param>
   <param-name>
javax.faces.INTERPRET_EMPTY_STRING_SUBMITTED_
VALUES_AS_NULL
  </param-name>
  <param-value>true</param-value>
</context-param>
```


Validateurs standards

- <f:validateDoubleRange>, attributs minimum et maximum
- <f:validateLongRange>, attributs minimum et maximum
- <f:validateLength>, attributs minimum et maximum
- <f:validateRequired>
- <f:validateRegex>, attribut pattern
- <f:validateBean>, attribut validationGroups

Messages d'erreur

- Si les messages d'erreur standards ne conviennent pas, on peut utiliser les attributs requiredMessage et validatorMessage des composants standards ; voir aussi la section suivante

Utilisation d'un validateur standard

```
<h:inputText id="carte" value="#{bb.carte}"
  requiredMessage="Carte obligatoire"
  validatorMessage="Numéro de carte invalide">
    <f:validateLength minimum="16"/>
</h:inputText>
```


Écriture d'un validateur

- Pour écrire sa propre validation, on peut écrire une classe qui implémente l'interface **`javax.faces.validator.Validator`** ou écrire une méthode dans un backing bean
- La classe doit implémenter la méthode **`void validate(FacesContext context, UIComponent composant, Object valeur)`** où valeur représente la valeur à valider
- La méthode lance une **`ValidatorException`** si la valeur n'est pas validée (ce qui entraînera le réaffichage de la page courante)

Identificateur du validateur

- Le composant doit contenir une balise **`<f:validator>`** avec un identificateur qui correspond à l'identificateur de la classe
- L'identificateur est donné par l'annotation **`@FacesValidator`** ou par la balise **`<validator>`** du fichier de configuration XML

Exemple

```
<h:inputText id="carte" value="#{bb.carte}"
  <f:validator
    validatorId="fr.unice.ValidateurCarte"/>
```

```
@FacesValidator("fr.unice.ValidateurCarte")
public class ValidateurCarte
       implements Validator {
  public void validate(FacesContext context,
    UIComponent composant, Object valeur) {
      ...
  }
  ...
}
```


Écriture de la validation

- Pas possible d'utiliser @Inject dans un validateur ou un convertisseur pour récupérer un EJB
- On peut donner le nom JNDI de l'EJB :

```
try {
   ic = new InitialContext();
   myejb = (MyEJB) ic
     .lookup("java:global/xxxx/MyEJB");
} catch (NamingException e) {
   …
}
```


Exemple de validateur

```
public void validateLogin(FacesContext ctxt,
UIComponent composant, Object valeur) {
  // Validation
  ...
  if (probleme) {
    FacesMessage message =
      new FacesMessage("...");
      throw new ValidatorException(message);
    }
  }
```

Obtenir les autres valeurs

- Parfois la validation requiert d'obtenir les autres valeurs saisies par l'utilisateur (pas encore validées) ; voici comment faire :

```
public void validateXxx(FacesContext ctxt,
    UIComponent comp, Object valeur) {
  UIInput autreComposant =
    (UIInput)comp.findComponent("autre");
  Object o1 = autreComposant.getLocalValue();
```


Valeur locale et Ajax

- La valeur locale (getLocalValue()) est la valeur convertie à partie de la valeur soumise (getSubmittedValue()) qui est la valeur récupérée à partir des paramètres de la requête POST envoyée après la soumission du formulaire
- Elle est donc nulle pour tous les composants qui ne sont pas traités par la requête Ajax
- Conclusion : si on veut utiliser la valeur locale d'un composant dans une requête Ajax, il faut inclure le composant dans le « execute » de la requête

Validation avec Ajax

- Ajouter une balise <f:ajax> dont l'attribut execute ne porte que sur les champs à valider (@this par défaut) et dont l'attribut render désigne la zone qui va afficher le message d'erreur éventuel
- Exemple :
  <f:ajax event="blur" render="messages" />
- La validation ne portera que sur les champs désignés par l'attribut execute

Méthode pour valider

- La méthode du backing bean doit avoir exactement le même type retour et les mêmes paramètres que la méthode **`validate`** de l'interface **`Validator`**
- Le composant à valider doit avoir un attribut **`validator`** qui désigne la méthode qui valide
- Exemple :

```
<h:inputText id="carte" value="#{bb.carte}"
    validator="#{bb.validateNumeroCarte}"/>
```


Messages d'erreur ou d'information

Affichage des messages

- La partie 1 de ce cours a étudié les balises <h:messages> et <h:message> qui permettent d'afficher les messages
- Cette section étudie
  - la génération de message par programmation
  - la modification des messages standards fournis pas JSF

Utilité

- Les conversions et validations peuvent conduire à l'affichage de messages d'erreur
- Les convertisseurs et validateurs standards produisent des messages d'erreur standard
- Il est possible de modifier les messages d'erreur standard
- Lorsqu'une action s'est bien déroulée et que la page courante reste la même, il faut le signaler à l'utilisateur par un message d'information

Classe **`FacesMessage`**

- Un message peut être généré automatiquement en liaison avec une erreur de conversion ou de validation
- Il peut aussi être généré par du code Java en utilisant la classe **`FacesMessage`**
- Il suffit de passer une instance de **`FacesMessage`** à la méthode **`addMessage`** de **`FacesContext`**
- Le message sera affiché si la page courante est réaffichée

Ajouter un message

- **`FacesContext.getCurrentInstance().addMessage`** permet d'ajouter un message par programmation
- Les paramètres de la méthode **`addMessage`** permettent d'indiquer
  - L'id du composant avec lequel le message est associé (peut être **`null`**)
  - Le message (classe **`FacesMessage`**)

Id d'un composant

- Exemple de désignation d'un composant : si le composant a l'id « montant » dans le formulaire d'id « transfert », l'id du composant à passer en 1er paramètre de addMessage est « transfert:montant »

Classe FacesMessage

- Le constructeur peut prendre en paramètre
  - La « gravité » du message par une constante de la classe FacesMessages.Severity : **`SEVERITY_FATAL`** (erreur grave), **`SEVERITY_ERROR`**, **`SEVERITY_WARN`** (avertissement), **`SEVERITY_INFO`** (information ; valeur par défaut))
  - Message résumé (*summary*) affiché par défaut par <h:message>
  - Message complet (*detail*) affiché par défaut par <h:messages>

Exemple de code

```
FacesContext.getCurrentInstance()
  .addMessage(
     component.getClientId(
       FacesContext.getCurrentInstance()),
     new FacesMessage(
        FacesMessage.SEVERITY_INFO,
        "Détails du message", null));
```

Pas de résumé

Message d'information

- Il suffit d'ajouter un message au contexte sans lancer d'exception
- Si la page courante est réaffichée, le message y apparaîtra dans une balise **<h:messages>** (ou, plus rarement, dans une balise **<h:message>** si le message comporte un id de composant)

Messages standards (1/2)

- Les messages standard sont dans le fichier jar jsf-api.jar, répertoire javax/faces, fichier Messages.properties (pour les messages en anglais ; en français : Messages_fr.properties)
- Il y a des messages qui concernent un composant, une erreur de conversion en un type, une erreur de validation
- Exemple de Messages_fr.properties :

```
javax.faces.component.UIInput.CONVERSION=
{0} : une erreur de conversion est survenue
```

- {0} remplacé par la valeur saisie

Messages standards (2/2)

- Les messages standard « détaillés » sont suffixés par « _detail » :
- javax.faces.converter.IntegerConverter.INTEGER _detail={2} : {0} doit être compris entre -2147483648 et 2147483647. Exemple : {1} {0} : une erreur de conversion est survenue
- Les paramètres : {0} est la valeur saisie ; {1} est un exemple de bonne saisie ; {2} est l'identifiant du composant sur lequel il y a eu l'erreur

Améliorer les messages standards

- Un message d'erreur peut comporter l'identifiant du composant concerné par l'erreur
- Pour rendre les messages plus clairs, il faut donner des identifiants significatifs (attribut **id**) aux formulaires et aux composants pour éviter les « j_idt6:j_idt8 » dans les messages
- On peut aussi utiliser l'attribut **label** des composants ; ce label sera utilisé pour désigner le composant dans les messages

Modifier les messages affichés

- Il ne faut pas toucher aux fichier standard mais
  - ajouter un autre fichier de propriétés,
  - modifier la valeur d'une clé (en prenant exemple sur la valeur standard pour utiliser les paramètres du message)
  - Modifier faces-config.xml pour désigner le nouveau fichier
- On peut aussi utiliser les attributs des composants de type **requiredMessage**, **converterMessage**,…

Exemple

- Fichier mesMessages :
  javax.faces.validator.LengthValidator.MINIMUM=
  …
- Dans faces-config.xml :

```
<application>
  <message-bundle>
    fr.unice.appli.MesMessages
  </message-bundle>
</application>
```

Messages par les attributs

- Toutes les balises de saisie (par exemple **`<h:inputText>`**) offrent des attributs pour modifier les messages d'erreur éventuels :
  - **`requiredMessage`**
  - **`converterMessage`**
  - **`validatorMessage`**
- Exemple :

```
<h:inputText id="nom"
  requiredMessage="Vous devez saisir le nom"/>
```


Avec l'API de validation

- Cette API permet de centraliser des contraintes sur les propriétés des beans (le plus souvent de type entité) ; ces contraintes doivent être respectées dans toutes les couches de l'application qui utilisent les beans (y compris JSF)
- Voir le cours sur l'API de validation des beans pour voir comment configurer les messages d'erreur

GET - PRG

- Par défaut, JSF travaille avec des requêtes POST
- Depuis JSF 2.0 il est plus simple de travailler avec des requêtes GET, ce qui facilite l'utilisation de l'historique et marque-pages (*bookmarks*) des navigateurs et évite des doubles validations de formulaire non voulues par l'utilisateur

Utiliser GET

- 2 composants de JSF 2.0 permettent de générer des requêtes GET : **`<h:button>`** et **`<h:link>`**

Le problème avec POST

- Avec une requête POST envoyée pour soumettre un formulaire
  - le refresh de la page affichée (ou un retour en arrière) après un POST soumet à nouveau le formulaire
  - l'adresse de la page affichée après le POST est la même que celle du formulaire ; il est impossible de garder un marque-page dans le navigateur pour les pages affichées après un POST

La raison du problème

- C'est le handler de navigation de JSF qui redirige vers la page désignée par le modèle de navigation JSF (à la fin de la phase « Invoke Application »)
- Le navigateur n'a pas connaissance de cette direction et pense être toujours dans la page qui contient le formulaire qui a été soumis

Les conséquences du problème

- Le navigateur est en retard d'une page pour afficher l'URL de la page en cours
- Il ne garde donc pas la bonne adresse URL si l'utilisateur veut garder un marque-page
- Le navigateur pense être toujours dans la page qui contient le formulaire après un retour en arrière ou un refresh et il le soumet à nouveau (il demande malgré tout une confirmation lors de la soumission multiple d'un formulaire, par une fenêtre pop-up inquiétante et difficile à comprendre pour l'utilisateur)

La solution : POST, REDIRECT, GET (PRG)

- Le modèle POST– REDIRECT– GET préconise de
  - Ne jamais montrer une page en réponse à un POST
  - Charger les pages uniquement avec des GET
  - Utiliser la redirection pour passer de POST à GET

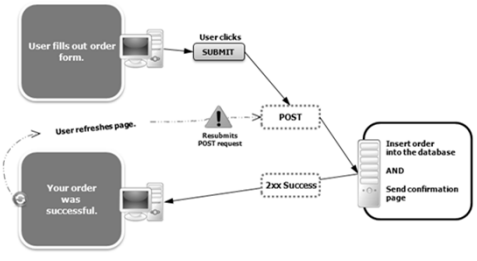
Redirection avec JSF

- Il est possible de configurer une adresse avec une redirection en donnant le paramètre
- Exemple :

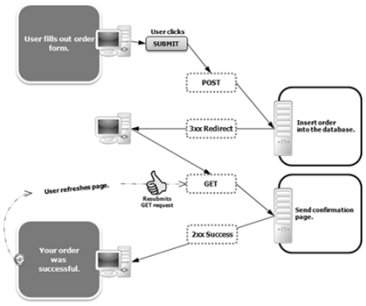
```
<h:commandButton value="…"
   action="page2?faces-redirect=true"
/>
```


Sans PRG

Cette image et la suivante viennent de Wikipedia



Avec PRG



Problème de PRG

- PRG peut poser un problème lorsque la page vers laquelle l'utilisateur est redirigée (le GET), doit afficher des données manipulées par le formulaire
- C'est le cas, par exemple, lorsque cette page est une page qui confirme l'enregistrement dans une base de données des informations saisies par l'utilisateur dans le formulaire
- En effet, les informations conservées dans la portée de la **requête** du POST ne sont plus disponibles

Solutions (1/2)

- Une des solutions est de ranger les informations dans la session plutôt que dans la requête
- Cependant cette solution peut conduire à une session trop encombrée
- Une autre solution est de passer les informations d'une requête à l'autre

Solutions (2/2)

- JSF 2.0 offre 3 nouvelles possibilités qui facilitent la tâche du développeur pour conserver les informations plus longtemps que la requête :
  - la mémoire flash
  - les paramètres de vue
  - la portée conversation de CDI
- Si on veut réafficher le même formulaire, on peut aussi utiliser la portée « vue »

Paramètres de vue

- Les paramètres d'une vue sont définis par des balises **`<f:viewParam>`** incluses dans une balise **`<f:metadata>`** (à placer au début de la page destination de la navigation, avant les **`<h:head>`** et **`<h:body>`**) :

```
<f:metadata>
  <f:viewParam name="n1"
               value="#{bean1.p1}" />
  <f:viewParam name="n2"
               value="#{bean2.p2}" />
</f:metadata>
```


`<f:viewParam>`

- L'attribut name désigne le nom d'un paramètre de requête GET
- L'attribut value désigne (par une expression du langage EL) le nom d'une propriété d'un bean dans laquelle la valeur du paramètre est rangée
- Important : la valeur d'un paramètre de vue est de type **`String`** ; il est possible d'indiquer une conversion ou une validation, comme sur les valeurs des composants saisis par l'utilisateur, ce qui permet d'utiliser d'autres types que **`String`**

`<f:viewParam>` - `includeViewParams`

- Un lien vers une page qui contient des balises **`<f:viewParam>`** contiendra tous les paramètres indiqués par les **`<f:viewParam>`** s'il contient « **`includeViewParams=true`** »
- Exemple (action sur une seule ligne) :

```
<h:commandButton value=…
  action="page2?faces-redirect=true
&amp;includeViewParams=true"
```

- Si l'action est une expression de méthode, « includeViewParams=true » doit être inclus dans la valeur renvoyée par la méthode

Fonctionnement de **includeViewParams**

1. La page de départ a un URL qui a le paramètre **includeViewParams**
2. Elle va chercher les **<f:viewParam>** de la page de destination. Pour chacun, elle ajoute un paramètre à la requête GET en allant chercher la valeur qui est indiquée par l'attribut value du **<f:viewParam>**
3. A l'arrivée dans la page cible, la valeur du paramètre est mise dans la propriété du bean indiquée par l'attribut value

Fonctionnement de **includeViewParams**

- Ca revient à faire passer une valeur d'une page à l'autre
- Si la portée du bean est la requête et qu'il y a eu redirection, ça revient plus précisément à faire passer la valeur d'une propriété d'un bean dans un autre bean (de la même classe)

Donner une valeur à un paramètre

- Il y a plusieurs façons de donner une valeur à un paramètre de requête GET ; les voici dans l'ordre de priorité inverse (la dernière façon l'emporte)
  - Dans la valeur du outcome
    <h:link outcome="page?p=4&p2='bibi' " …>
    <h:link outcome="page?p=#{bean.prop + 2} " …>
  - Avec les paramètres de vue
    <h:link outcome="page" includeViewParams="true" …>
  - Avec un <f:param>
    <h:link outcome="page" includeViewParams="true" …>
    <f:param name="p" value=…/> </h:link>

Exemple ; **page2.xhtml**

```
<!DOCTYPE …>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns=...>
<f:metadata>
  <f:viewParam name="param1"
               value="#{bean.prop1}"/>
  <f:viewParam name="param2" value="#{…}"/>
</f:metadata>
<h:head>…</h:head>
<h:body>
  Valeur : #{bean.prop1}
```


Fonctionnement

- Si **page2** est appelé par la requête GET suivante **page2.xhtml?param1=v1&param2=v2** la méthode **setProp1** du bean est appelée avec l'argument v1 (idem pour param2 et v2)
- Si un paramètre n'apparait pas dans le GET, la valeur du paramètre de requête est **null** et le *setter* n'est pas appelé (la propriété du bean n'est donc pas mise à null)

Conserver des adresses utiles

- Outre le fait qu'un refresh ne provoque plus de soumission du formulaire, le modèle PRG permet aussi de permettre de conserver un URL utile dans un marque-page ou dans l'historique
- En effet, l'URL contient les paramètres qui permettront de réafficher les mêmes données à un autre moment
- Sans PRG, les données utiles sont conservées dans l'entité de la requête et pas dans l'URL

preRenderView Listener (1)

- Une situation courante : il est possible de ne mettre en paramètre qu'un identificateur des données qui ont été traitées
- Par exemple, que la clé primaire des données enregistrées dans la base de données
- Avec les procédés précédents on peut faire passer cet identificateur dans la page qui va afficher les données

preRenderView Listener (2)

- Mais ça ne suffit pas ; il faut utiliser une autre nouveauté de JSF 2.0, les listeners « preRenderView » qui permet d'exécuter une action avant l'affichage de la vue
- On peut ainsi aller rechercher (par exemple dans une base de données) les informations identifiées par l'identificateur
- Pour le GET, le code du *listener* joue en quelque sorte le rôle de l'action pour un POST

Exemple de preRenderView

```
<f:metadata>
  <f:viewParam name="id"
               value="#{bean.IdClient}"/>
  <f:event type="preRenderView"
           listener="#{bean.chargeClient}"/>
</f:metadata>
<h:head>...</head>
...
Nom : #{bean.client.nom}
...
```


Exemple de listener

```
public class Bean {
private int idClient;
private Client client;

  public void chargeClient() {
    client = facadeClient.findById(idClient);
  }
  ...
}
```


Cas où on ne veut pas afficher la page

- Si la méthode du listener « preRenderView » a un problème ou si les paramètres ne sont pas validés on peut vouloir afficher une page d'erreur plutôt que la page envisagée au départ avec un message d'erreur
- On peut diriger vers une autre page avec la méthode **`handleNavigation`** de la classe **`NavigationHandler`** , et utiliser **`isValidationFailed()`** de la classe **`FacesContext`** pour détecter les erreurs de validation. Inutile avec <f:viewAction> de JSF 2.2

Exemple

```
public void chargeClient() {
  FacesContext ctx =
    FacesContext.getCurrentInstance();
  if (ctx.isValidationFailed()) {
    ctx.getApplication().getNavigationHandler()
      .handleNavigation(ctx, null, erreur);
    return;
  }
  try {
    client = facadeClient.findById(idClient);
  } catch(NoSuchEntryException) {
    ctx. getApplication().getNavigationHandler()
      .handleNavigation(ctx, null, erreur);
  }
}
```

<f:viewAction>

- JSF 2.2 devrait introduire une nouvelle possibilité (pas encore disponible) qui peut remplacer avantageusement un listener preRenderView dans le cas d'un problème
- Cette balise se place dans la section <f:metadata> comme <f:viewparam> :

```
<f:metadata>
  <f:viewParam id=… name=…
               value="#{bb.val}"/>
  <f:viewAction action="#{bb.checkVal}" />
</f:metadata>
```


<f:viewAction> dans le cycle de vie

- Par défaut, la méthode désignée par action est exécutée avant l'affichage de la page suivante durant la phase « Invoke Application » comme pour les actions des composants UICommand (POST)
- Elle ne sera donc pas exécutée si les valeurs des <f:viewParam> n'ont pas été validées, ce qui peut économiser du code par rapport aux versions pré 2.2 de JSF
- La méthode action peut retourner une String qui indique une autre page à afficher à la place de la page qui contient <f:viewAction>

<f:viewAction> dans le cycle de vie

- La méthode désignée par action peut être exécutée durant n'importe quelle phase du cycle
- Ce qui permet d'écourter un traitement si on aperçoit un problème
- Exemple :

```
<f:viewAction action="#{bb.checkVal}"
              phase="UPDATE_MODEL_VALUES"/>
```

- Comme pour les composants UICommand, il est aussi possible de le faire exécuter durant la phase « Apply Request » avec l'attribut `immediate="true"`

Navigation déclarative

- La navigation après l'exécution d'une action peut être déclarée dans un fichier faces-config.xml comme pour les actions des composants UICommand

Aussi pour les POST

- Par défaut, l'action ne sera exécutée que pour une requête GET
- On peut cependant la faire exécuter aussi après un POST :

```
<f:viewAction action="#{bb.checkVal}"
              onPostback="true"/>
```


Protection de pages par login / mot de passe

Utilité

- Des pages Web ne doivent être accessibles que par certains utilisateurs ou dans certaines circonstances
- Le support « Sécurité avec Java EE » contient une section qui explique comment restreindre l'accès d'une partie des pages d'une application à certains utilisateurs

Cookies

Utilité

- Les cookies peuvent servir à récupérer des préférences de l'utilisateur pour lui rendre la vie plus facile
- Ils permettent de conserver des informations entre 2 sessions

Obtenir les cookies

- En Java, il faut passer par la requête HTTP
- Dans le langage EL : objet prédéfini cookie qui est une map dont les clés sont les noms des cookies de la requête en cours

Exemple en Java

```
Map<String, Object> cookies =
  externalContext.getRequestCookieMap();
// nom contient le nom d'un cookie
Cookie cookie = (Cookie) cookies.get(nom);
/* Ajouter un cookie ; proprietes est une map
(qui peut être null) qui contient des clés
pour initialiser le cookie : comment, domain,
maxAge, secure, path */
externalContext.addResponseCookie(nom,
valeur, proprietes);
```


Exemple avec le langage EL

- #{cookie['nomCookie']} ou #{cookie.nomCookie}

Composants composés

Définition

- Introduits pas JSF 2.0
- Nouveaux composants créés sans écrire de code Java, à partir d'autres composants
- Il suffit de créer un fichier xhtml qui contient la définition du composant en 2 parties :
  - l'interface du composant (donne les attributs de la balise associée au composant)
  - l'implémentation du composant (donne le code XHTML du composant)

Conventions à respecter

- Le fichier de définition du composant devra se trouver dans le répertoire resources/ezcomp de l'application
- En effet, le composant sera considéré comme une ressource, dans la bibliothèque ezcomp
- Le nom du fichier de définition sera le nom du composant (avec en plus le suffixe .xhtml) ; dans l'exemple suivant le fichier s'appellera **`adresse.xhtml`** et le composant **`adresse`**

Exemple – entête

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

inspiré d'un exemple du livre
« Java EE 6 Development with NetBeans 7 »
de David R. Heffelfinger, Packt Publishing

Exemple – interface

```
<cc:interface>
  <cc:attribute name="typeAdresse"/>
  <cc:attribute name="managedBean" required="true"/>
</cc:interface>
```

Attribut qui sera un bean géré

Un attribut peut être obligatoire (devra être donné avec la balise du composant) ou non

Exemple – implémentation (1/2)

```
<cc:implementation>
  <h:panelGrid columns="2">
    <f:facet name="header">
      <h:outputText
        value="Adresse #{cc.attrs.typeAdresse}"/>
    </f:facet>
    <h:outputLabel for="rue" value="Rue"/>
    <h:inputText id="rue"
                 value="#{cc.attrs.managedBean.rue}"/>
    <h:outputLabel for="ville" value="Ville"/>
    <h:inputText id="ville"
                 value="#{cc.attrs.managedBean.ville}"/>
```

Attribut **`typeAdresse`** du composant

Exemple – implémentation (2/2)

```
    <h:outputLabel for="pays" value="Pays "/>
    <h:inputText id="pays"
                 value="#{cc.attrs.managedBean.pays}"
                 size="2" maxlength="2"/>
  </h:panelGrid>
</cc:implementation>
</html>
```


Utilisation du composant

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC ...>
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:ezcomp="http://java.sun.com/jsf/composite/ezcomp">
  <h:head>
    <title>Adresse</title>
  </h:head>
  <h:body>
    <h:form>
      <h:panelGrid columns="1">
        <ezcomp:address managedBean="#{addressBean}"
                        addrType="domicile"/>
  ...
```


Balises personnalisées

3 façons d'ajouter des balises

- Balise personnalisée (*custom tag*), utilise les templates, <ui:composition>, librairie de tag avec <tag><source>
- Composant composite (*composite component*), composants composés d'autres composants, sans écrire de code Java
- Composant personnalisé (*custom component*), composants qui nécessitent du code Java, librairie de tag avec <tag><component><component-class>

Balise personnalisée

- On a vu qu'il est possible de créer des balises personnalisées pour les convertisseurs
- Plus généralement le développeur peut écrire très simplement ses propres balises sans écrire du code Java, avec un fichier XML
- La création de ces balises s'appuie sur les templates ; la définition d'une balise est incluse dans <ui:composition>

Étapes pour ajouter une balise personnalisée

1. Écrire la définition de la balise dans un fichier XHTML, à l'intérieur d'un <ui:composition>
2. Ajouter un fichier de librairie de tag qui donne le nom de la balise et l'associe au fichier XHTML qui contient sa définition
3. Déclarer le fichier de librairie de tag dans le fichier web.xml

Exemple

Limites des balises personnalisées

- Cependant, ces balises personnalisées ne permettent pas tout ce qu'une balise standard permet ; par exemple, il n'est pas possible d'ajouter un validateur ou un écouteur à une balise personnalisée

Travailler avec un serveur local et un serveur distant dans NetBeans

Installation d'un serveur distant

- Quand on déclare le serveur distant, on indique n'importe quel emplacement local pour le répertoire d'installation

Choix du serveur

- Pour choisir le serveur à utiliser, il suffit de faire un clic droit sur le projet, de choisir les propriétés, cliquer sur Run et de choisir le serveur (en haut de la fenêtre)
- Attention de bien choisir au moment du lancement de l'application ou du déploiement, pour ne pas, par exemple, écraser une version de production sur le serveur distant par un version de développement sur le serveur local

Voir la version de Mojarra utilisée

- Au déploiement de l'application dans GlassFish il s'affiche des informations
- Une des informations a la forme suivante : INFO: Initialisation de Mojarra 2.1.3 (FCS b02) pour le contexte «/NomApplication»

Changer version Mojarra dans GlassFish

- Shutdown GlassFish
- Replace the Mojarra files jsf-impl.jar and jsf-api.jar in the glassfishv3/glassfish/modules directory with the new Mojarra files jsf-impl.jar and jsf-api.jar
- Delete everything in the glassfish/domains/domain1/osgi-cache directory
- Start GlassFish
- Verify that the good version of Mojarra appears in the GlassFish log file when Web applications are started
- **??**

Changer version Mojarra dans GlassFish

- Apparently it's also possible to copy the Mojarra 2.0.3 files jsf-impl.jar and jsf-api.jar to the Web applications WEB-INF/lib directory, but I've never tried it. In this case, the following is needed in glassfish-web.xml:

```
<sun-web-app>
   <class-loader delegate="false"/>
</sun-web-app>
```

- **??**