INTRODUCTION À JSF2

Adapté du cours de Richard Grin (grin@unice.fr)

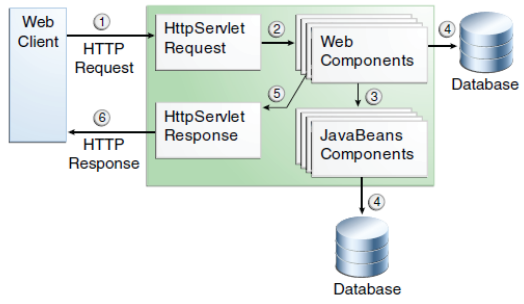
Adaptation Michel Buffa (buffa@unice.fr), UNSA 2012

Version Richard Grin 1.2 du 30/11/11

Remarque

- Ce support est une introduction à JSF 2.0 utilisé dans un cadre Java EE 6, avec un serveur d'applications du type de Glassfish, et CDI (Contexts and Dependency Injection) activé dans les projets.

Architecture typique d'une application web



Utilité de JSF

- Créer des pages Web dynamiques avec des composants construits sur le serveur

Services rendus par JSF (1/2)

- Permet de bien séparer l'interface utilisateur, la couche de persistance et les processus métier
- Conversion des données (tout est texte dans l'interface utilisateur)
- Validation des données saisies par l'utilisateur
- Automatisation de l'affichage des messages d'erreur en cas de problèmes de conversion ou de validation

Services rendus par JSF (2/2)

- Internationalisation
- Support d'Ajax sans programmation javascript (communication en arrière-plan et mise à jour partielle de l'interface utilisateur)
- Fournit des composants standards pour l'interface utilisateur, puissants et faciles à utiliser
- Possible d'ajouter ses propres composants

Standards

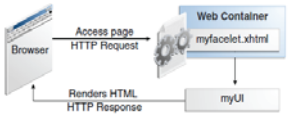
- JSF 2.0 est intégré dans Java EE 6
- JSF 2.0 peut (c'est conseillé) utiliser CDI (*Contexts and Dependency Injection*)

Page JSF

- Les pages JSF contiennent des balises qui décrivent les composants qui représenteront la page sur le serveur

Une page JSF

- Code XHTML qui contient des balises qui décrivent les composants sur le serveur
- Sera traduit en XHTML « pur » pour être envoyé au client Web



- Contient des parties en EL : #{…}
- Utilise souvent une (ou plusieurs) bibliothèque de composants (PrimeFaces ou RichFaces par exemple)

Architecture des applications JSF

Composants de l'architecture

- Dans l'architecture des applications qui utilisent JSF
  - des pages JSF
  - des *backing beans*
  - des EJB
  - des entités
  - des classes Java ordinaires

Répartition des tâches (1/2)

- Les pages JSF sont utilisées pour l'interface avec l'utilisateur ; elles ne contiennent pas de traitements (pas de code Java ou autre code comme dans les pages JSP)
- Les backing beans font l'interface entre les pages JSF et le reste de l'application
- Ces backing beans peuvent effectuer les traitements liés directement à l'interface utilisateur ; ils font appels à des EJB ou des classes Java ordinaires pour effectuer les autres traitements

Répartition des tâches (2/2)

- Les EJB sont chargés des traitements métier et des accès aux bases de données
- Les accès aux bases de données utilisent JPA et donc les entités

Backing bean

- Souvent, mais pas obligatoirement, un backing bean par page JSF
- Un backing bean fournit du code Java pour l'interface graphique

Backing bean

- Le traitement peut être exclusivement lié à l'interface graphique (par exemple si un composant n'est visible que si l'utilisateur a choisi une certaine valeur) ; dans ce cas le backing bean intervient seul
- Sinon le backing bean fait appel à des EJB ou quelquefois à des classes Java ordinaires
- Remarque importante : un EJB doit être totalement indépendant de l'interface graphique ; il exécute les processus métier ou s'occupe de la persistance des données

Container pour les JSF

- Pour qu'il sache traiter les pages JSF, le serveur Web doit disposer d'un container JSF
- On peut utiliser pour cela un serveur d'applications du type de Glassfish ou équivalent

Composants JSF sur le serveur

- JSF utilise des composants côté serveur pour construire la page Web
- Par exemple, un composant java UIInputText du serveur sera représenté par une balise <INPUT> dans la page XHTML
- Une page Web sera représentée par une vue, UIViewRoot, hiérarchie de composants JSF qui reflète la hiérarchie de balises HTML

Le cycle de vie

Cycle de vie JSF 2

- Pour bien comprendre JSF il est indispensable de bien comprendre tout le processus qui se déroule entre le remplissage d'un formulaire par l'utilisateur et la réponse du serveur sous la forme de l'affichage d'une nouvelle page.

Le servlet « Faces »

- Toutes les requêtes vers des pages « JSF » sont interceptées par un servlet défini dans le fichier **`web.xml`** de l'application Web
- ```
  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>
      javax.faces.webapp.FacesServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  ```

URL des pages JSF

- Les pages JSF sont traitées par le servlet parce que le fichier web.xml contient une configuration telle que celle-ci :
- ```
  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>/faces/*</url-pattern>
  </servlet-mapping>
  ```

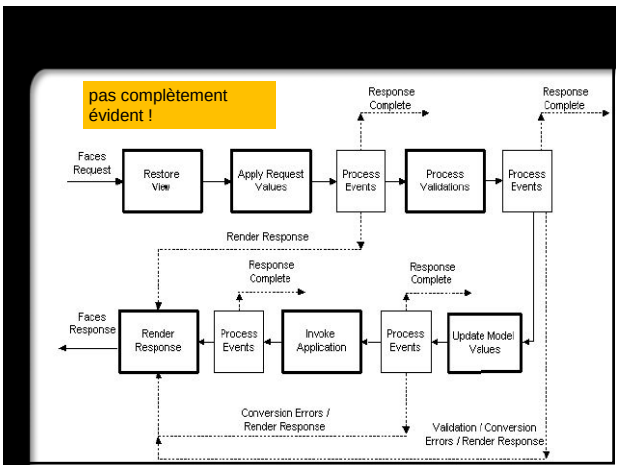
  Le pattern peut aussi être de la forme **`*.faces`** ou *.jsf

Codage - décodage

- Les pages HTML renvoyées par une application JSF sont représentées par la vue, arbre de composants Java
- L'encodage est la génération d'une page HTML à partir de l'arbre des composants
- Le décodage est l'utilisation des valeurs renvoyées par un POST HTML pour donner des valeurs aux variables d'instance des composants Java, et le lancement des actions associées aux « UICommand » JSF (boutons ou liens)

Cycle de vie

- Le codage/décodage fait partie du cycle de vie des pages JSF,
- Le cycle de vie est composé de 6 phases,
- Ces phases sont gérées par le servlet « Faces » qui est activé lorsqu'une requête demande une page JSF



Demande page HTML

- Etudions tout d'abord le cas simple d'une requête GET d'un client qui demande l'affichage d'une page JSF

La vue

- Cette requête HTTP est interceptée par le servlet Faces,
- La page HTML correspondant à la page JSF doit être affichée à la suite de cette requête HTTP
- La page HTML qui sera affichée est représentée sur le serveur par une « vue »
- Cette vue va être construite sur le serveur et transformée sur le serveur en une page HTML qui sera envoyée au client

Contenu de la vue

- Cette vue est un arbre dont les éléments sont des composants JSF qui sont sur le serveur (des instances de classes qui héritent de UIComponent),
- Sa racine est de la classe UIViewRoot

Construction vue

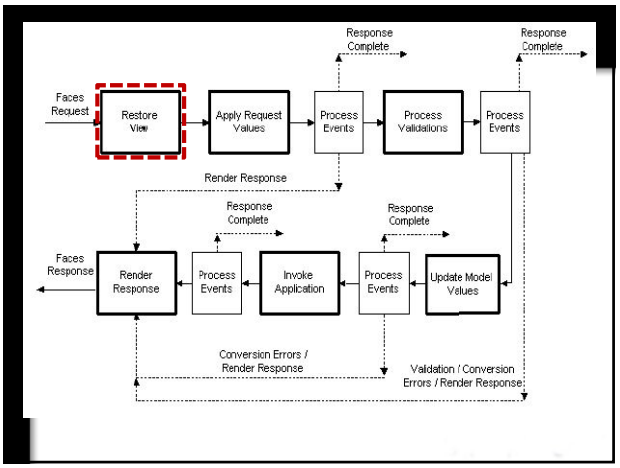
- Une vue formée des composants JSF est donc construite sur le serveur (ou restaurée si elle avait déjà été affichée) :
  phase « Restore View »
- La vue sera conservée sur le serveur pour le cas où on en aurait encore besoin

Rendu de la page HTML

- Puisqu'il n'y a pas de données ou d'événements à traiter, la vue est immédiatement rendue : le code HTML est construit à partir des composants de la vue et envoyé au client :
  phase « Render Response »

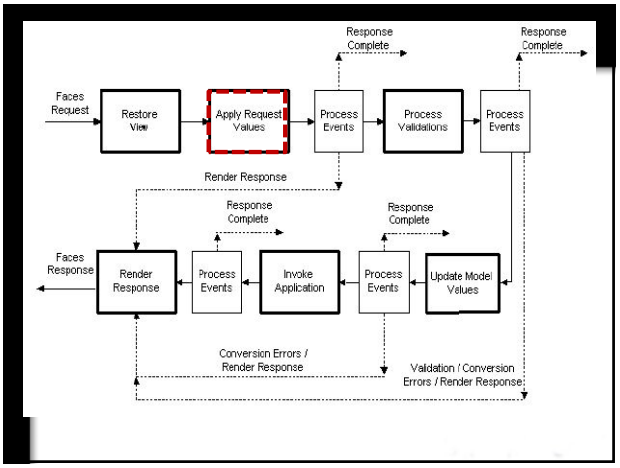
Traitement d'un formulaire

- Nous allons cette fois-ci partir d'une requête HTTP générée à partir d'une page HTML qui contient un formulaire
- L'utilisateur a saisi des valeurs dans ce formulaire
- Ces valeurs sont passées comme des paramètres de la requête HTTP ; par exemple, `http://machine/page.xhtml?nom=bibi&prenom=bob`
  si la requête est une requête GET, ou dans le corps d'un POST



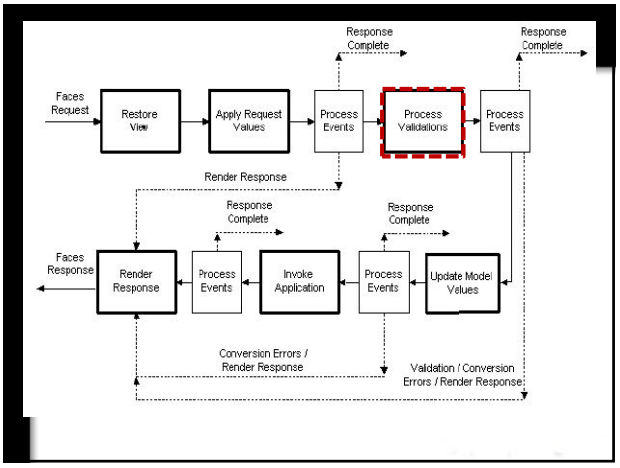
Phase de restauration de la vue

- La vue qui correspond à la page qui contient le formulaire est restaurée (phase « Restore View »)
- Tous les composants reçoivent la valeur qu'ils avaient avant les nouvelles saisies de l'utilisateur



Phase d'application des paramètres

- Tous les composants Java de l'arbre des composants reçoivent les valeurs qui les concernent dans les paramètres de la requête HTTP : phase « Apply Request Values »
- Par exemple, si le composant texte d'un formulaire contient un nom, le composant Java associé conserve ce nom dans une variable
- En fait, chaque composant de la vue récupère ses propres paramètres dans la requête HTTP

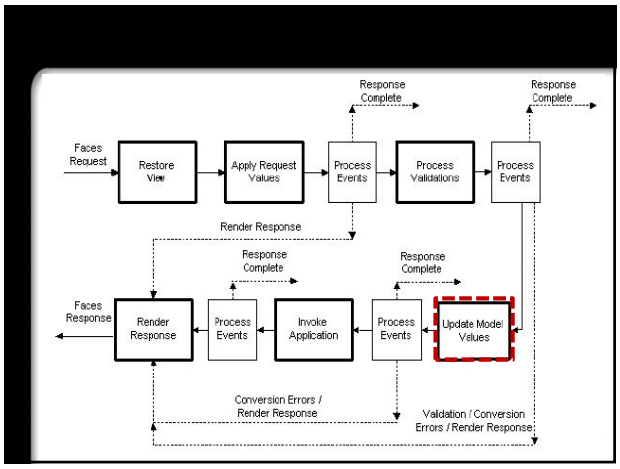


Phase de validation

- Les données traitées à l'étape précédentes sont converties dans le bon type Java
- Elles sont aussi validées (par exemple, si un âge doit être compris entre 18 et 65)
- Si une validation échoue, la main est donnée à la phase de rendu de la réponse

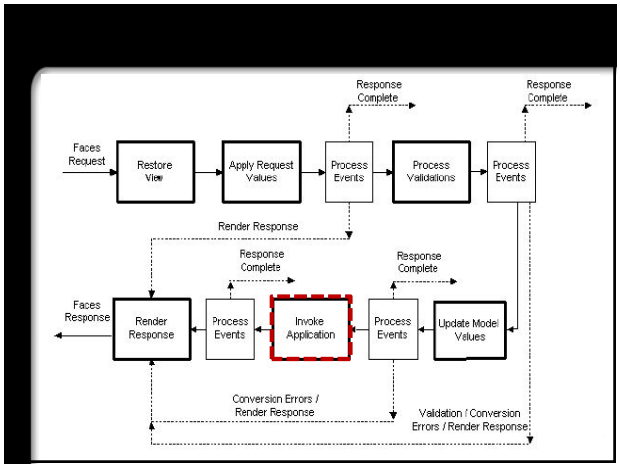
Phase de validation

- Comment c'est fait : chaque composant valide et convertit les données qu'il contient
- Si un composant détecte une valeur non valable, il met sa propriété « valid » à false et il met un message d'erreur dans la file d'attente des messages (ils seront affichés lors de la phase de rendu « render response ») et les phases suivantes sont sautées



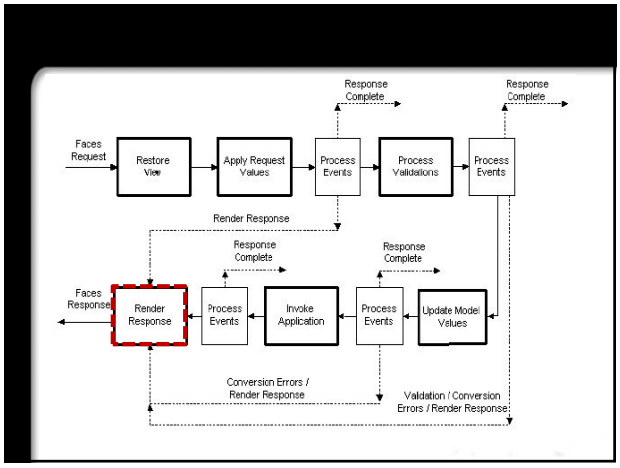
Phase de mise à jour du modèle

- Si les données ont été validées, elles sont mises dans les propriétés des Java beans associées aux composants de l'arbre des composants
- Exemple :
  <h:inputText id="nom"
  value="#{backingBean.utilisateur.nom}" />



Phase d'invocation de l'application

- Les actions associées aux boutons ou aux liens sont exécutées
- Le plus souvent le lancement des processus métier se fait par ces actions
- La valeur de retour de ces actions va déterminer la prochaine page à afficher (navigation)



Phase de rendu de la réponse

- La page déterminée par la navigation est encodée en HTML et envoyée vers le client HTTP

Sauter des phases

- Il est quelquefois indispensable de sauter des phases du cycle de vie
- Par exemple, si l'utilisateur clique sur un bouton d'annulation, on ne veut pas que la validation des champs de saisie soit effectuée, ni que les valeurs actuelles soient mises dans le modèle
- Autre exemple : si on génère un fichier PDF à renvoyer à l'utilisateur et qu'on l'envoie à l'utilisateur directement sur le flot de sortie de la réponse HTTP, on ne veut pas que la phase de rendu habituelle soit exécutée

immediate=true sur un UICommand

- Cet attribut peut être ajouté à un bouton ou à un lien (<h:commandButton>, <h:commandLink>, <h:button> et <h:link>) pour faire passer immédiatement de la phase « Apply request values » à la phase « Invoke Application » (en sautant donc les phases de validation et de mise à jour du modèle),
- Exemple : un bouton d'annulation d'un formulaire.

immediate=true sur un EditableValueHolder

- Cet attribut peut être ajouté à un champ de saisie, une liste déroulante ou des boîtes à cocher pour déclencher immédiatement la validation et la conversion de la valeur qu'il contient, avant la validation et la conversion des autres composants de la page
- Utile pour effectuer des modifications sur l'interface utilisateur sans valider toutes les valeurs du formulaire

Exemple (1/2)

- Formulaire qui contient champ de saisie du code postal et un champ de saisie de la ville,
- Lorsque le code postal est saisi, un ValueChangeListener met automatiquement à jour la ville :

```
<h:inputText valueChangeListener="#{…}"
...
  onChange = "this.form.submit()" />
```

- La soumission du formulaire déclenchée par la modification du code postal ne doit pas lancer la validation de tous les composants du formulaire qui ne sont peut-être pas encore saisis

Exemple (2/2)

- La solution est de mettre un « immediate=true » sur le champ code postal.

Navigation entre pages JSF

Handler de navigation

- La navigation entre les pages indique quelle page est affichée quand l'utilisateur clique sur un bouton pour soumettre un formulaire ou sur un lien
- La navigation peut être définie par des règles dans le fichier de configuration faces-config.xml ou par des valeurs écrites dans le code Java ou dans la page JSF

Navigation statique et dynamique

- La navigation peut être statique : définie « en dur » au moment de l'écriture de l'application
- La navigation peut aussi être dynamique : définie par l'état de l'application au moment de l'exécution (en fait, par la valeur retournée par une méthode)

Dans les pages JSF

- Dans les pages JSF on indique la valeur ou la méthode associée à un bouton qui déterminera la navigation.
- Avec une valeur :

```
<h:commandButton value="Texte du bouton"
    action= "pageSuivante"/>
```

- Avec une méthode (qui retourne un nom de page) :

```
<h:commandButton value="Texte du bouton"
    action="#{nomBean.nomMethode}"/>
```

Dans le fichier de configuration faces-config.xml

```
<navigation-rule>
  <from-view-id>/index.xhtml</from-view-id>
  <navigation-case>
    <from-outcome>succes</from-outcome>
    <to-view-id>/succes.xhtml</to-view-id>
  <navigation-case>
  <navigation-case>
    <from-outcome>echec</from-outcome>
    <to-view-id>/echec.xhtml</to-view-id>
  <navigation-case>
</navigation-rule>
```

Valeur retournée

Managed Beans / Backing Beans

Classes pour soutenir les pages JSF

Michel Buffa (buffa@unice.fr), UNSA 2012

Plan de la section

- Le contenu des beans dans JSF :
  - Des getters/setters qui correspondent aux <input…/> de formulaires (qu'on appelle "propriétés"), avec, le plus souvent les variables correspondantes
  - Des méthodes « action » par ex : verifieLoginPassword(…)
  - Des méthodes listener, des méthodes pour convertir et valider
- Les portées des beans
- Comment pré-remplir les champs des formulaires
  - Notamment les champs <input../> et les menus/listes déroulantes

Les composantes des beans gérés

- Des propriétés (définies par des get/set ou is…)
  - Une paire pour chaque élément input de formulaire,
  - Les setters sont automatiquement appelés par JSF lorsque le formulaire sera soumis.
- Des méthodes « action »
  - Une par bouton de soumission dans le formulaire
  - La méthode sera appelée lors du clic sur le bouton par JSF

Les composantes des beans gérés

- Des propriétés pour les données résultat
  - Seront initialisées par les méthodes « action » après un traitement métier,
  - Il faut au moins une méthode get sur la propriété afin que les données puissent être affichées dans une page de résultat.
- Des méthodes diverses pour écouter des événements (par exemple un changement de valeur dans un champ input), convertir et valider

Caractéristiques des Beans gérés

- Le container JSF « gère » le bean
  - Il l'instancie automatiquement,
    - Nécessité d'avoir un constructeur par défaut
  - Contrôle son cycle de vie
    - Ce cycle dépend de la portée (scope) (request, session, application, etc.)
  - Appelle les méthodes setter
    - Par ex pour <h:inputText value="#{customer.firstName"/>, lorsque le formulaire est soumis, le paramètre est passé à setFirstName(…)
  - Appelle les méthodes getter
    - #{customer.firstName} revient à appeler getFirstName()
- Déclaration par @ManagedBean avant la classe, ou par @Name si CDI

Un exemple simple (1)

- Scénario
  - Entrer le login d'un client et son password,
  - Afficher en résultat :
    - Une page affichant son nom, prénom et solde de son compte bancaire,
      - 3 pages différentes selon la valeur du solde,
    - La page initiale avec des messages d'erreur si le formulaire a été mal rempli (données manquantes ou incorrectes), ou une page d'erreur

Un exemple simple (2)

- De quoi a besoin le bean géré ?
  - Propriétés correspondant aux éléments du formulaire d'entrée : ex getCustomerLogin/setCustomerLogin, etc.
  - Méthode « action » :
    - Pour récupérer un Customer à partir du login.
  - De quoi stocker les résultats
    - Stocker le Customer résultat dans une variable d'instance, initialement vide, avec get/set associés

BanqueMBean partie 1 : propriétés pour les éléments de formulaire

```
@ManagedBean
public class BanqueMBean {
    private String customerLogin, password;

    public String getCustomerLogin() {
        return customerLogin;
    }

    public void setCustomerLogin(String customerLogin) {
        this.customerLogin = customerLogin;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }
}
```

Appelé par JSF lors de l'affichage du formulaire, comme la variable vaut null au départ le champs sera vide.

Lors de la soumission, le bean, est instancié à nouveau (puisqu'il est RequestScoped par défaut), et la valeur dans le formulaire est passée à cette méthode

get et setPassword sont identiques à get et setCustomerLogin, à part qu'on ne peut pré-remplir un champ password (interdit par les navigateurs), donc getPassword ne sera pas appelé initialement

BanqueMBean partie 2 : méthodes « action »

```
@Named
public class BanqueMBean {
    private String customerLogin, password;
    private Customer customer;

    //… ici get/set pour login, password, et customer

    public String checkConnexion() {
       customer =
         customerManager.checkLoginPassword(
              customerLogin, password);
       if(customer != null) {
           return "bienvenue";
       } else {
           return null; // affiche la même page
       }
    }
}
```

Nom d'un bean (version CDI)

- `@Named`
  `public class BanqueMBean {`
- Par défaut c'est le nom de la classe avec une minuscule au début : **banqueMBean**
- Pour donner un autre nom :
  `@Named("banqueController")`
  `public class BanqueMBean {`

Nom d'un bean (version JSF)

- `@ManagedBean`
  `public class BanqueMBean {`
- Par défaut c'est le nom de la classe avec une minuscule au début : **banqueMBean**
- Pour donner un autre nom :
  `@ManagedBean(name="banqueController")`
  `public class BanqueMBean {`

Création d'un backing bean

- Indique la durée de vie du backing bean
- Quand une page JSF fait référence à un bean, par exemple
  <h:inputText id="nom"
  value="#{backingBean.utilisateur.nom}" />
- le container JSF regarde si une instance de la classe existe déjà
- Si une instance est trouvée, elle est utilisée
- Sinon, une nouvelle instance est créée par le container

La portée d'un backing bean

- La durée de vie d'un bean dépend de sa portée
- Par exemple un bean de portée Request sera supprimé lorsque la requête en cours sera terminée
- Les beans « CDI » et les beans « JSF » ont plusieurs portées en commun et quelques portées particulières

Les différentes portées

- Valeurs communes à CDI et JSF : request, session, application
- Valeurs particulières à CDI : conversation, dependant
- Valeurs particulières à JSF : view, none ou custom
- RequestScope = valeur par défaut.
- On les spécifie avec des annotations (recommandé) ou dans faces-config.xml

Annotations pour les Scopes

- @RequestScoped
  - On crée une nouvelle instance du bean pour chaque requête.
  - Puisque les beans sont aussi utilisés pour initialiser des valeurs de formulaire, ceci signifie qu'ils sont donc généralement instanciés deux fois (une première fois à l'affichage du formulaire, une seconde lors de la soumission)
  - Ceci peut poser des problèmes…

Annotations pour les Scopes

- @SessionScoped
  - On crée une instance du bean et elle durera le temps de la session. Le bean doit être Sérialisable.
  - Utile par exemple pour gérer le statut « connecté/non connecté » d'un formulaire login/password.
  - On utilisera les attributs « render » des éléments de UI pour afficher telle ou telle partie des pages selon les valeurs des variables de session.
  - Attention à ne pas abuser du SessionScoped, pièges possibles avec les variables cachées ou les éditions de données multiples (cf tp1)

Annotations pour les Scopes

- @ApplicationScoped
  - Met le bean dans « l'application », l'instance sera partagée par tous les utilisateurs de toutes les sessions.
  - Pour des méthodes utilitaires ou pour mettre en cache des informations qui ne doivent pas varier (liste des pays par exemple).

Annotations pour les Scopes

- @ViewScoped
  - La même instance est utilisée aussi souvent que le même utilisateur reste sur la même page, même s'il fait un refresh (reload) de la page !
  - Le bean doit être sérialisable,
  - Convient bien pour les pages JSF faisant des appels Ajax (une requête ajax = une requête HTTP => une instance si on est en RequestScoped !)

Annotations pour les Scopes

- @ConversationScoped
  - Utilise CDI, ne fait pas partie de JSF, @Named obligatoire (pas @ManagedBean)
  - Semblable aux session mais durée de vie gérée « par programme »,
  - Utile pour faire des wizards ou des formulaires remplis partiellement au travers de plusieurs pages;
  - On va confier à une variable injectée le démarrage et la fin de la durée de vie du bean,

Ou placer ces annotations

- Après @Named ou @ManagedBean en général,
  - Named recommandé si on a le choix.
- Attention aux imports !!!!!
  - Si @ManagedBean (JSF), alors les scopes doivent venir du package javax.**faces**.bean
  - Si @Named (CDI) les scopes doivent venir de javax.**enterprise**.context
  - Cas particuliers : ConversationScoped que dans javax.enterprise.context donc utilisé avec @Named, et ViewScoped vient de javax.faces.bean donc utilisé avec @ManagedBean

@SessionScoped

- Idée : gestionnaire login password. Si l'utilisateur se trompe de password on réaffiche le champs login rempli, sinon on dirige vers la page d'accueil et on mémorise dans une variable le fait qu'on est authentifié.
- Backing Bean :
  - Sérializable

@SessionScoped, page JSF login/password

```
<h:outputText rendered="#{!loginMBean.connected}"
              value="#{loginMBean.message}" />
<h:form id="loginForm"
        rendered="#{!loginMBean.connected}">
   Username: <h:inputText id="loginForm"
                  value="#{loginMBean.login}" />
   Username: <h:inputText id="password"
                  value="#{loginMBean.password}" />
   <h:commandButton value="Login"
                  action="#{loginMBean.connexion}" />
</h:form>
… suite transparents suivant !
```

```
<h:form id="deconnexionForm"
      rendered="#{loginMBean.connected}">
   <h:outputText
        value="#{loginMBean.message}" />
   <h:commandButton value="Deconnexion"
     action="#{loginMBean.deconnexion}" />
 </h:form>
```

- Note : l'attribut rendered remplace les if/then/else de JSTL !

@SessionScoped, page JSF login/password

```
@Named(value = "loginMBean")
@SessionScoped
public class LoginMBean implements Serializable {
    private String login;
    private String password;
    private boolean connected = false;
    private String message = "Veuillez vous identifier :";
    public boolean isConnected() {
        return connected;
    }
…
```

SessionScoped, page JSF login/password

```
public void deconnexion() {
   connected = false;
   message = "Veuillez vous identifier :";
}
public void connexion() {
   connected = (login.equals("michel")
             && password.equals("toto"));
   if (connected) {
       message = "Bienvenue, vous êtes connecté en tant que "
               + login + " ! ";
   } else {
       message = "Mauvais login/password, recommencez";
   }
}
```

@ViewScoped

- Prévu pour les pages faisant des appels Ajax,
  - Pas de nouvelle instance tant qu'on est dans la même page JSF
  - La navigation doit renvoyer null ; toute autre valeur, même désignant la même page, efface les données de la vue

@ConversationScoped

```
@Named(value = "customerMBean")
@ConversationScoped
public class CustomerMBean implements Serializable {
   @Inject
   private Conversation conversation;

   public String showDetails(Customer customer) {
       this.customer = customer;
       conversation.begin();
       return "CustomerDetails?id=" +
       customer.getCustomerId() +
       "&faces-redirect=true";
   }
   public String update() {
       customer = customerManager.update(customer);
       conversation.end();
       return "CustomerList?faces-redirect=true";
   }
```

@ConversationScoped

- Danger : à chaque begin() doit correspondre un end()
  - Attention avec des pages qui sont « bookmarkables » (correction tp1 par ex, le formulaire est bookmarkable) et appelables de plusieurs manières,
  - Piège avec les évènements preRenderView : la méthode appelée doit faire un begin et il faut faire un end quand on sort du formulaire, mais comme la page peut être aussi invoquée depuis une autre page, il ne faut pas faire un begin « avant » le preRenderView…
  - conversation.isTransient() peut aider

Modèle de navigation GET - PRG

Navigation par défaut

- Par défaut, JSF travaille avec des requêtes POST
- Depuis JSF 2.0 il est aussi devenu simple de travailler avec des requêtes GET, ce qui facilite l'utilisation des marque-pages (*bookmarks*) et de l'historique 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 (donc pas possible de faire réafficher cette page en utilisant l'historique du navigateur)

La raison du problème

- C'est le servlet JSF qui redirige vers la page désignée par le modèle de navigation JSF
- Le navigateur n'a pas connaissance de cette direction et pense être toujours dans la page qui contient le formulaire qui a été soumis

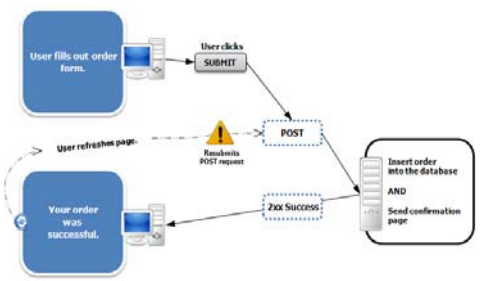
Les conséquences du problème

- 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 essaie de le soumettre à nouveau (il demande malgré tout une confirmation lors de la soumission multiple d'un formulaire)

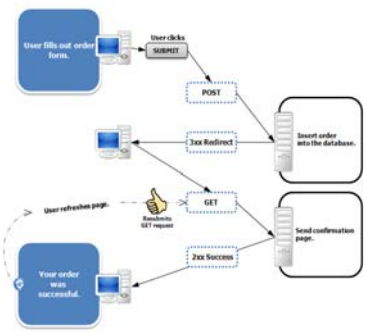
La solution : POST, REDIRECT, GET (PRG)

- 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.

Sans PRG



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
- Exemple : page qui confirme l'enregistrement dans une base de données des informations saisies par l'utilisateur dans le formulaire
- En effet, si les informations sont conservées dans un bean de portée **requête** de la page du formulaire, elles ne sont plus disponibles après la redirection

Solutions

- 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
- JSF 2.0 offre 3 nouvelles possibilités qui facilitent la tâche du développeur :
  - la mémoire flash,
  - les paramètres de vue,
  - La portée conversation de CDI

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 HTTP 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 : il est possible d'indiquer une conversion ou une validation à faire sur les paramètres, comme sur les valeurs des composants saisis par l'utilisateur.

<f:viewParam> (suite)

- Un URL vers une page qui contient des balises **`<f:viewParam>`** contiendra tous les paramètres indiqués par les **`<f:viewParam>`** s'il contient « **`includeViewParams=true`** »
- Exemple :

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

  - Dans le navigateur on verra l'URL avec les paramètres HTTP.

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`

- Cela 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, cela revient plus précisément à faire passer la valeur d'une propriété d'un bean dans un autre bean du même type

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" …>
    <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`**).

Bookmarquer des URL de page

- 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 depuis la vue
- 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>
...
Nom : #{bean.client.nom}
...
```

ANNEXES

Attributs communs aux composants

- **`id`** donne un identifiant au composant
- **`rendered`** indique si le composant doit être affiché (la valeur est le plus souvent le résultat booléen d'une méthode Java du backing bean, donné par une expression EL)
- **`styleClass`** donne le nom de la classe CSS pour la mise en forme du composant
- **`binding`** permet de lier le composant à une propriété d'un backing bean (classe Java)

Attribut `binding`

- Le composant peut ainsi être manipulé par le backing bean avec du code Java
- On n'utilise cet attribut que pour des traitements spéciaux, comme, par exemple, lier une dataTable pour pouvoir récupérer la ligne en cours par la méthode Java **`table.getRowData()`**

Messages d'information ou d'erreur

- Les messages d'erreur liés à JSF, générés par JSF ou par le code Java sont affichés
  - dans une zone de la page réservée aux messages
  - ou près du composant qui a généré l'erreur
- Exemples de messages :
  - indique que la saisie de l'utilisateur a provoqué une erreur de conversion ou de validation
  - message généré par le code Java pour une raison quelconque

<h:messages>

- Réserve un endroit de la page pour afficher tous les messages pour tous les composants et les messages qui ne sont pas liés à un composant particulier
- De nombreux attributs permettent de définir le style CSS et la mise en page et d'indiquer si on veut seulement un résumé ou les détails du message
- Exemple :

```
<h:messages errorClass="erreur" />
```


<h:message>

- Un attribut obligatoire for indique l'identificateur du composant qui est lié à ce message
- Habituellement placé près du composant dans la page JSF
- Exemple :

```
<h:message for="nom" />
```

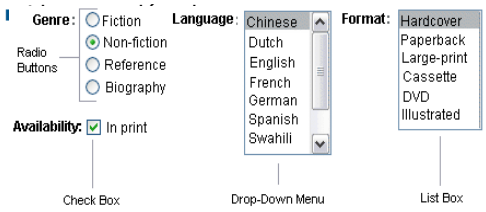

Annexe sur les listes déroulantes

Les listes

- Il y a plusieurs composants JSF standards qui permettent de proposer un choix (ou plusieurs) parmi une liste de valeurs
- Tous les composants sont bâtis sur des classes Java communes mais se présentent différemment à l'utilisateur

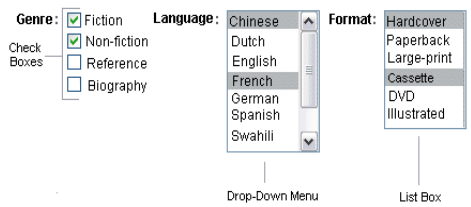
Listes à choix unique

- Boutons radios avec <h:selectOneRadio>
- Liste déroulante avec <h:selectOneMenu>



Listes à choix multiple

- Boîtes à cocher avec <h:selectManyCheckbox>
- Liste déroulante avec <h:selectManyMenu>



Les listes - Exemple

```
<h:selectOneListbox value="#{bean.choix}">
  <f:selectItem itemLabel="choix"
                itemValue="#{bean.option}" />
  <f:selectItem itemLabel="autre choix"
                itemValue="false" />
  <f:selectItems value="#{bean.options}"/>
</h:selectOneListbox>
```

Ce qui est désigné par **`f:selectItem`** et **`f:selectItems`** sera affiché dans la liste. Le choix fait par l'utilisateur parmi toutes ces valeurs, est rangé dans **`bean.choix`** (attribut **`value`** de **`selectOneListBox`**)

Les entrées de la liste qui correspondent à l'attribut **`value`** sont présélectionnées dans la liste

Classe `SelecItem`

- Paquetage **`javax.faces.model`**
- Donne un label, une valeur et d'autres propriétés d'une liste de GUI
- Utilisée pour la valeur (attribut value) des tags **`<f:selectItem>`** et **`<f:selectItems>`**

`<f:selecItems>`

- Son attribut **`value`** peut désigner un tableau, une collection ou une map
- Le plus souvent, on part d'une liste d'éléments et on la transforme en tableau ou collection de **`SelectItem`**
- Le transparent suivant donne un schéma de code pour une telle transformation (souvent fourni sous la forme d'une méthode **`static`** utilitaire)

Schéma de code pour `<f:selecItems>`

```
public static SelectItem[]
       getSelectItems(List<?> entities) {
  SelectItem[] items = new
  SelectItem[size];
  for (Object x : entities) {
    items[i++] =
      new SelectItem(x, x.toString());
  }
  return items;
}
```

Autre façon de faire (JSF 2.0)

- Pour éviter la complication de la transformation en **`SelectItem`**, on peut utiliser les attributs **`var`**, **`itemValue`** et **`itemLabel`**
- Exemple :

```
<f:selectItems
  value="#{ecoleController.personnes}"
  var="personne"
  itemValue="#{personne.id}"
  itemLabel="#{personne.nom}"/>
```

La variable « **`personne`** » contient un élément de la liste désignée par **`value`**

Chausse-trappes avec les listes

- Quelques chausse-trappes que l'on peut rencontrer avec les listes déroulantes sont exposées dans les transparents suivants
- En particulier avec les convertisseurs et les validateurs

Conversion pour les listes déroulantes

- Comment se fait la conversion entre les éléments d'une liste et ce qui est affiché ?
- Par exemple, une liste d'écoles et ce qui est affiché ?
- Il faut bien comprendre que le composant JSF va être transformé en élément(s) HTML et que toutes les valeurs vont être transformées en **`String`** ; en retour, une valeur choisie par l'utilisateur sera une **`String`** qu'il faudra éventuellement convertir en un autre type

Entrée « fictive » de la liste

- Si la liste contient des pays, de type **`Pays`**, il faut un convertisseur pour passer du type **`Pays`** à **`String`** (pour le passage en HTML) et vis-versa (pour le rangement du choix sous la forme d'une instance du type **`Pays`**)
- On aura un message d'erreur si on veut mettre une 1ère entrée du type « Choisissez un pays » car cette première entrée n'est pas du type **`Pays`**, même avec l'attribut noSelectionOption à true (voir transparent suivant), si on ne traite pas ce cas particulier dans le convertisseur

Désigner un convertisseur

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

- **`fooConverter`** désigne un convertisseur qui convertit des String dans le type de myBean.selectedItem et vis-versa

Convertisseurs standards

- Fournis avec JSF
- Exemple :

```
<inputText
value="#{employe.dateNaissance}">
    <f:convertDateTime
        pattern="dd/MM/yyyy"/>
</inputText>
```

Écrire un convertisseur

- Classe Java qui implémente l'interface **`Converter`** (paquetage **`javax.faces.convert`**) qui contient 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

Validation d'un choix dans une liste

- JSF vérifie que le choix qui est envoyé au serveur est bien un des choix proposés à l'utilisateur
- Ça implique que les choix proposés soient connus par JSF au moment de la validation
- Si les choix dépendent d'une propriété d'un backing bean (par exemple, un pays pour une liste de villes d'un certain pays), il faut donner la bonne portée au bean pour que les choix puissent être calculés correctement au moment de la validation (View ou Session par exemple)

Annexe sur Java beans

Bean basique et bean géré

- Rappel : un bean c'est une classe java qui suit certaines conventions
  - Constructeur vide,
  - Pas d'attributs publics, les attributs doivent avoir des getter et setters, on les appelle des propriétés.
- Une propriété n'est pas forcément un attribut
  - Si une classe possède une méthode getTitle() qui renvoie une String alors, on dit que la classe possède une propriété « title »,
  - Si book est une instance de cette classe, alors dans une page JSF #{book.title} correspondra à un appel à getTitle() sur l'objet book

Bean basique et bean géré

- Une propriété booléenne peut être définie par la méthode isValid(), au lieu de getValid()
- Ce sont bien les méthodes qui définissent une « propriété », pas un attribut. On peut avoir isValid() sans variable « valid ».

Bean basique et bean géré

- Règle pour transformer une méthode en propriété
  - Commencer par get, continuer par un nom capitalisé, ex getFirstName(),
  - Le nom de la propriété sera firstName
  - On y accèdera dans une page JSF par #{customer.firstName} où customer est l'instance du bean et firstName le nom de la propriété,
  - Cela revient à appeler la méthode getFirstName() sur l'objet customer.

Bean basique et bean géré

- Exception 1 : propriétés booléennes
  - getValid() ou isValid() (recommandé),
  - Nom de la propriété : valid
  - Accès par #{login.valid}
- Exception 2 : propriétés majuscules
  - Si deux majuscules suivent le get ou le set, la propriété est toute en majuscule,
  - Ex : getURL(),
  - Propriété : URL,
  - Accès par #{website.URL}

Exemples de propriétés

Nom de méthode	Nom de propriété	Utilisation dans une page JSF
getFirstName setFirstName	firstName	#{customer.firstName} <h:inputText value="#{customer.firstName}"/>
isValid setValid (booléen)	valid	#{login.valid} <h:selectBooleanCheckbox value="#{customer.valid}"/>
getValid setValid (booléen)	valid	#{login.valid} <h:selectBooleanCheckbox value="#{customer.valid}"/>
getZIP setZIP	ZIP	#{address.ZIP} <h:inputText value="#{address.ZIP}"/>

Annexe pour le codage des backing beans

Accéder aux objets Request et Response

- Pas d'accès automatique !
- Il faut « penser » différemment, il faut considérer les formulaires comme des objets.
- Si vous avez quand même besoin d'accéder à la requête et à la réponse, le code est un peu complexe…
  - Utile pour : manipuler la session, par exemple régler la durée.
  - Manipulation des cookies explicite, consulter le user-agent, regarder le host, etc.

Exemple

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

Faire afficher un message d'erreur

```
String msg = "Compte source n'existe pas";
FacesMessage facesMsg =
  new FacesMessage(
    FacesMessage.SEVERITY_ERROR,
    msgResume, msgDetails);
FacesContext.getCurrentInstance()
  .addMessage("transfert:source",
              facesMsg);
```