

JSF

Olivier Cailloux

LAMSADE, Université Paris-Dauphine

Version du 14 mai 2017

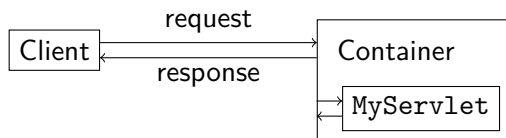
# Objectif de JSF

Objectif : apporter une solution aux difficultés du développement web

## Difficultés web

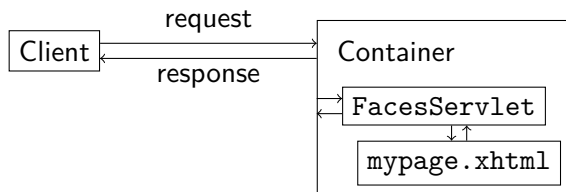
- canevas pages
- synchronisation état
- validation
- contexte
- requêtes partielles...

## Requête Servlet normale



- Problème : envoyer une page HTML à l'aide d'un servlet (page non statique)
- Construire le code HTML via du code Java ?
- `out.print("<html><head>"); ...`
- Illisible, inélégant, difficile à tester et à valider...
- Souhait : langage approprié pour générer HTML
- Solution JSF : langage basé sur XML

# Requête JSF



- Router la requête vers le servlet JSF : FacesServlet
- Déjà fourni par Java EE
- FacesServlet cherche une page correspondant à la requête
- Exemple : `contextPath/faces/mypage.xhtml`  $\Rightarrow$  JSF cherche page `mypage.xhtml`
- Page à décrire dans un langage de définition de vue (VDL)
- Requête JSF  $\neq$  Requête Faces d'après spec JSF

# Composition de JSF

## View Definition Languages

- Facelets (autre ?)

# Composition de JSF

## View Definition Languages

- Facelets (autre ? JSP, déprécié)

# Composition de JSF

## View Definition Languages

- Facelets (autre ? JSP, déprécié)
  - bibliothèque de tags
- 
- API : état composants ; événements ; validation ; conversions ; navigation ; i18n
  - Utilise beans gérés
  - Composants UI ; modèle de rendering ; de conversion ; d'écoute ; de validation

# Facelets

- Page décrite en langage Facelets : page XHTML
- Utilisant des éléments HTML
- Et des éléments Facelets



## Facelet my-static-page.xhtml

Page souhaitée

Name:

my-static-page.xhtml :

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict ...>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
  <head>
    <meta http-equiv="Content-Type" content = ... />
    <title>Title</title>
  </head>
  <body>
    <h:outputText value="Name:" />
    <h:inputText value="Mon nom" />
  </body>
</html>
```

# Cycle de traitement

- Cycle de traitement de requête (*Request Processing Lifecycle*) : requête du client traitée par JSF jusqu'à réponse du serveur
- JSF implémente IOC (?)

# Cycle de traitement

- Cycle de traitement de requête (*Request Processing Lifecycle*) : requête du client traitée par JSF jusqu'à réponse du serveur
- JSF implémente IOC (? Inversion of Control)

# Cycle de traitement

- Cycle de traitement de requête (*Request Processing Lifecycle*) : requête du client traitée par JSF jusqu'à réponse du serveur
- JSF implémente IOC (? Inversion of Control) : code du développeur appelé à des moments spécifiés dans un cycle de vie pré-défini
- (Autre IOC déjà vu ?)

# Cycle de traitement

- Cycle de traitement de requête (*Request Processing Lifecycle*) : requête du client traitée par JSF jusqu'à réponse du serveur
- JSF implémente IOC (? Inversion of Control) : code du développeur appelé à des moments spécifiés dans un cycle de vie pré-défini
- (Autre IOC déjà vu ? Servlets !)

# Phases du cycle de traitement

## Exécution

- 1 *Restore View*
- 2 *Apply Request Values* ; PE
- 3 *Process Validations* and convert ; PE
- 4 *Update Model Values* ; PE
- 5 *Invoke Application* ; PE

Lors des Process Events (PE) :

- Response complete → arrêt
- Render Response ou erreurs Validation ou Conversion → court-circuit

## Rendu (rendering)

- 6 *Render Response* : typiquement, génération d'HTML

## Exemple statique

- Requête vers `faces/my-static-page.xhtml`
- Requête JSF : exécution cycle de traitement
- *Restore view* : création composants `HtmlOutputText`, `HtmlInputText`
- Retiennent paramètres value correspondants

### Composants générés (`my-static-page.xhtml`)

- Composant `HTMLOutputText` : `value = Name:`
- Composant `HTMLInputText` : `value = Mon Nom`

## Exemple statique (2)

- Phases *Apply Request Values, Process Validations, Update Model Values, Invoke Application* : rien à faire
- *Render Response* : HTML en entrée renvoyé tel quel ; composants invoqués pour génération HTML

HTML généré (`my-static-page.xhtml`)

⇒ Name:<input type="text" ... value="Mon Nom" />

Name:




## Lecture dynamique

- Souhait : afficher le nom enregistré dans un bean géré
- `MyBean.uname` (`@Named`) contient le nom de l'utilisateur

Page souhaitée

Name:




- En Facelet : `<h:inputText value="#{myBean.uname}">`
- Lors exécution du cycle de traitement ?

## Lecture dynamique

- Souhait : afficher le nom enregistré dans un bean géré
- `MyBean.username` (`@Named`) contient le nom de l'utilisateur

Page souhaitée

Name:



- En Facelet : `<h:inputText value="#{myBean.username}">`
- Lors exécution du cycle de traitement ?
- *Restore view* : création composants, dont `HtmlInputText` avec `value` qui référence propriété `username` dans `myBean`
- *Render Response* : lors génération HTML par composant `HtmlInputText`, utilisation de la valeur de `username`

# Formulaire POST

## Page souhaitée dans formulaire HTML

Name:

Submit

Souhaité : envoi formulaire en POST lors soumission

Code Facelet:

```
<body>
  <h:form>
    <h:outputText value="Name:" />
    <h:inputText value="#{myBean.uname}" />
    <h:commandButton value="Submit"
      action="#{myBean.submitName}" />
  </h:form>
</body>
```

⇒ création de trois composants + formulaire

⇒ encapsulation dans un élément `<form method="post">` HTML

# Postback

- Utilisateur remplit formulaire
- Utilisateur clique Submit
- Navigateur envoie données du formulaire en POST à la même page
- *Postback* : nom de la requête POST quand envoyée à la page d'origine
- Dans l'exemple : postback vers `.../faces/mypage.xhtml`
- Requête JSF  $\Rightarrow$  cycle de traitement démarre

# Cycle postback

- *Restore View* : restauration arbre de composants depuis mémoire
- *Apply Request Values* :
  - Place valeur nom entrée dans composant `HTMLInputText`
  - Enregistre que bouton a été cliqué dans composant `HTMLCommandButton`
- *Process Validations* : rien à faire
- *Update Model Values* : place valeur de `HTMLInputText` dans propriété `uname` de `myBean`
- *Invoke Application* :
  - active événement lié au `HTMLCommandButton`
  - ⇒ exécute méthode `submitName` de `myBean`
  - conseillé : envoie redirection ou navigue et recommence cycle, ou passe à phase Rendering

## Redirection suite à postback

Conseillé : utiliser le patron de conception PRG (?)

## Redirection suite à postback

Conseillé : utiliser le patron de conception PRG (?  
POST-redirect-GET)

## Redirection suite à postback

Conseillé : utiliser le patron de conception PRG (?  
POST-redirect-GET)

```
public String submitName() {  
    // do stuff  
    // if success:  
    return "my-page-when-successful?faces-redirect=true"  
    // otherwise:  
    return "my-page-when-failure?faces-redirect=true";  
}
```

- ?faces-redirect=true : fait savoir à JSF qu'il doit renvoyer une réponse de redirection
- JSF renvoie HTTP REDIRECT vers .../faces/my-page-...
- Client demande .../faces/my-page-...
- Requête traitée par FacesServlet



# Facelets

- Un des VDL défini par JSF
- ViewId pointe vers une page Facelet (XHTML 1.0)
- Éléments XML non préfixés : seront simplement restitués au client tels quels
- Éléments XML préfixés référant une bibliothèque de tags JSF : agissent sur la vue construite par JSF
- JSF définit des bibliothèques de tags et leur sémantique
- Associés à des préfixes d'une lettre (usage courant, mais peuvent être changés)
- Utilisation de tags tierce-partie possible
- Tag également appelé custom action ou custom tag

## Standard HTML RenderKit Tag Library

- `http://xmlns.jcp.org/jsf/html`, préfixe `h:`
- Chaque tag associé à un composant et un renderer
- `h:commandButton` ; `h:commandLink` ; `h:inputFile` ;  
`h:inputTextarea` ; `h:selectBooleanCheckbox` ; `h:outputLink`...
- Ces tags ont pour propriétés l'union des attributs du Composant et du Renderer sur lesquels ils sont basés
- Exemple : propriété `value` pour tags créant des composants `ValueHolder` (ou `UICommand`)
- Tags créant composant `ActionSource2` a propriété `action`
- Valeur sera transférée à `actionExpression` du composant

# Facelet Core Tag Library

- `http://xmlns.jcp.org/jsf/core`, préfixe `f:`
- Agissent *généralement* sur le composant parent le plus proche
- Indépendant de HTML
- Exemple, `f:convertNumber` : enregistre un convertisseur sur un composant

# Recommandations pages Facelet

## Objectif

- Objectif : servir une page HTML 5 (conforme à spec HTML 5, sérialisation HTML 5)
- ... Tout en écrivant une page valide XHTML 1.0

Pour ce faire :

- PAS d'en-tête XML `c-à-d <?xml version="1.0" ... ?>`
- `<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">`
- `<html xmlns="http://www.w3.org/1999/xhtml" xmlns:h="http://xmlns.jcp.org/jsf/html" ...>`

# Expression Language

- Unified Expression Language
- Permet de référencer des propriétés ou des méthodes
- ... d'objets *instanciés*
- Value expression, r-value : propriété (lecture seule)
- Value expression, l-value : propriété (lecture et écriture)
- Pourquoi {l / r}-value ?
- Method expression : méthode

Exemple :

```
<h:inputText rendered = "#{day.odd}"  
    value = "#{employee.name}"/>
```

# Expression Language

- Unified Expression Language
- Permet de référencer des propriétés ou des méthodes
- ... d'objets *instanciés*
- Value expression, r-value : propriété (lecture seule)
- Value expression, l-value : propriété (lecture et écriture)
- Pourquoi {l / r}-value ? Similaire à  $x := 3$  (x est une l-value)
- Method expression : méthode

Exemple :

```
<h:inputText rendered = "#{day.odd}"  
    value = "#{employee.name}"/>
```

# Instanciación

- Exemple de value expression : `#{employee.name}`
- Référence managed bean nommé Employee
- Instanciación automática d'après son scope
- Utilise méthode `getName()` pour lecture
- Utilise méthode `setName(String)` pour écriture
- Scopes **CDI** sur managed beans! et pas `javax.faces.bean...`
- Pour classe accessible via EL : l'annoter en plus `@Named`

Exemple :

`@Named`

`@SessionScoped`

```
public class Employee {
```

## Combinaison de EL et Facelets

- EL : spécification indépendante de Facelets
- Mais usage souvent combiné
- Cf. fonctionnement normal du cycle de vie Facelets !



## Combinaison de EL et Facelets : exemple

### Exemple

```
<h:inputText value = "#{employee.name}" />  
<h:commandButton action = "#{myCtrl.submitName}" />
```

Quel traitement après envoi via bouton Submit ?

## Combinaison de EL et Facelets : exemple

### Exemple

```
<h:inputText value = "#{employee.name}" />  
<h:commandButton action = "#{myCtrl.submitName}" />
```

Quel traitement après envoi via bouton Submit ?

- Lors *Process Request Values* : "Mon nom" enregistré dans `HtmlInputText`

## Combinaison de EL et Facelets : exemple

### Exemple

```
<h:inputText value = "#{employee.name}" />  
<h:commandButton action = "#{myCtrl.submitName}" />
```

Quel traitement après envoi via bouton Submit ?

- Lors *Process Request Values* : "Mon nom" enregistré dans `HtmlInputText`
- ... `ActionEvent` lié à `HtmlCommandButton` placé dans queue

## Combinaison de EL et Facelets : exemple

### Exemple

```
<h:inputText value = "#{employee.name}" />  
<h:commandButton action = "#{myCtrl.submitName}" />
```

Quel traitement après envoi via bouton Submit ?

- Lors *Process Request Values* : "Mon nom" enregistré dans `HtmlInputText`
- ... `ActionEvent` lié à `HtmlCommandButton` placé dans queue
- Lors *Update Model Values* : `employee.setName(Mon nom)`

## Combinaison de EL et Facelets : exemple

### Exemple

```
<h:inputText value = "#{employee.name}" />  
<h:commandButton action = "#{myCtrl.submitName}" />
```

Quel traitement après envoi via bouton Submit ?

- Lors *Process Request Values* : "Mon nom" enregistré dans `HtmlInputText`
- ... `ActionEvent` lié à `HtmlCommandButton` placé dans queue
- Lors *Update Model Values* : `employee.setName(Mon nom)`
- Lors *Invoke Application* : `myCtrl.submitName()`

# Utilité des Facets

- Composants organisés en arbre (parent – enfant)
- Parfois nécessaire d'associer à un composant des sous-composants non enfants
- Ou des sous-composants jouant un rôle particulier
- Possible grâce aux « Facets »

## Exemple : tableau de données

- Enfants : colonnes de données
- Sous-composant de rôle « header »
- Sous-composant de rôle « footer »

# Facets : mécanisme

- Chaque composant a des Facets : `public Map<String, UIComponent> getFacets();`
- Facet : une association d'un rôle (String) et d'un composant

## En Facelets

```
<h:someComponent ... >  
  <f:facet name="someRole">  
    <h:someSubComponent />  
  </f:facet>  
</h:someComponent>
```

## f:metadata

- Nous allons avoir besoin d'une Facet sur UIViewRoot
- Problème (avec Facelets) ?



## f:metadata

- Nous allons avoir besoin d'une Facet sur UIViewRoot
- Problème (avec Facelets) ? Pas d'élément représentant UIViewRoot

## f:metadata

- Nous allons avoir besoin d'une Facet sur UIViewRoot
- Problème (avec Facelets) ? Pas d'élément représentant UIViewRoot
- Mais on peut utiliser le tag f:metadata
- Typiquement juste dessous le tag html
- Enregistre une Facet sur UIViewRoot (avec rôle métadonnée)  
en fait un UIPanel
- Permet d'inclure du comportement qui ne rentre pas ailleurs sur la page
- En l'occurrence : pour récupérer les paramètres requêtes lors d'un GET

# Requête GET

- GET HTTP : information requête contenue entièrement dans l'URL
- Représente requête idempotente
- Peut donc être « bookmarquée »
- Ou référencée de n'importe où
- Paramètres envoyés via composante « query » de URL
- Exemple : `http://.../context/faces/user.xhtml?idUser=25161&details=3`

## Récupération paramètres GET en Facelets

- En Facelet : paramètre capturé à l'aide de `f:viewParam`
- Doit être dans élément `f:metadata`
- Fonctionne comme un `UIInput`

### Capture de paramètres `idUser` et `details`

```
<html ...>
  <f:metadata>
    <f:viewParam name=idUser
      value="#{userBean.id}" />
    <f:viewParam name=details
      value="#{displayBean.detailLevel}" />
  </f:metadata>
```

## Événements du cycle de vie

- On peut aussi intervenir à des moments donnés du cycle de traitement
- Par exemple : charger contenu de la page en fonction de paramètres requêtes
- Utiliser `f:viewAction`

### Intervention pré-Render Response

```
<f:metadata>
  <f:viewAction action="#{ctrl.loadAnswer}">
  </f:viewAction>
</f:metadata>
```

# GET Vs POST

- Le plus souvent : méthode GET
- Utiliser GET en première approximation **ssi requête « safe »** (lecture seule)
- Formulaire de recherche : doit générer un GET
- Concevoir d'abord les pages de résultat
- Doivent fonctionner même lors d'une requête extérieure
- Attention : beaucoup de tutoriaux sur le net se concentrent sur les requêtes POST (plus intéressant, raison historique)

# Recherche via GET

## Formulaire de recherche par nom

Name pattern:

- Concevoir d'abord page de résultats !
- Répond à un GET : `.../search.xhtml?pattern=...`
- Tester indépendamment du formulaire de recherche
- Ensuite, il n'y a plus qu'à s'arranger pour que le formulaire de recherche renvoie à la bonne URL

# Formulaires GET et Facelets

- Composants `ActionSource2` génèrent des requêtes POST
- Dans cas simples, un formulaire HTML statique suffit !
- Si composants facelets nécessaires : pattern PRG (voir plus loin)



# Formulaires HTML

## Rappel formulaires HTML ([spec](#))

- Utiliser élément `form`
- Indiquer méthode GET ou POST : `form method="get"` (GET par défaut)
- Placer éléments `input` ou autres contrôles dans formulaire
- Données formulaire (*form data set*) : noms contrôles et valeurs associées
- GET : Le client envoie un GET avec données dans l'URL
- POST : le client envoie un POST avec données dans le corps

## Exemple de formulaire HTML (GET)

### Un formulaire HTML

```
<form action="http://example.com.../getuser.xhtml"
      method="get">
  First name: <input type="text" name="fname" />
  Last name: <input type="text" name="lname" />
  <br />
  <input type="submit" value="Send" />
</form>
```

First name:

Last name:

- Utilisateur entre les données et clique sur Send
- Le client envoie un GET à `http://example.com.../getuser.xhtml?fname=First&lname=Last`

# Requêtes POST

- Modifications données ou autres requêtes non idempotentes : utiliser POST
- Afficher les résultats : utiliser un GET ! permet traitement correct du refresh côté utilisateur, affichage url correcte...
- Pour combiner les deux : pattern POST-redirect-GET

# Pattern POST-redirect-GET

- ➊ Affichage initial de la page contenant formulaire POST
- ➋ Utilisateur clique élément correspondant à composant `ActionSource2`
- ➌ Client envoie *postback* : requête sous forme POST avec données de l'utilisateur
- ➍ Activation du composant `ActionSource2` côté serveur
- ➎ Réponse du serveur : redirect vers page-réponse
- ➏ Client s'exécute en effectuant un GET

# POST et Facelets

- Sur `initial.xhtml` : utiliser composant `ActionSource2` (t.q. `h:commandButton`)
- Rendu page initiale : indique au client d'utiliser POST
- Utilisateur soumet formulaire : action composant invoquée
- Action renvoie outcome sous forme `response.xhtml?faces-redirect=true&...`
- Serveur répond HTTP REDIRECT vers `response.xhtml?...`
- Client demande `response.xhtml?...`
- Serveur sert `response.xhtml?...`

En résumé : fait savoir au client qu'il est renvoyé de `initial.xhtml` à `response.xhtml`

# Mémorisation de l'état

- Rappel : HTTP sans état
  - Mais en réalité il faut souvent conserver un état !
  - Exemple : utilisateur enregistre un nouvel item
  - Il faut retenir l'id item pour le montrer à l'utilisateur en réponse
  - Recommandé (quand possible) : le *client* conserve l'état
  - Car conforme à philosophie HTTP
  - Permet bookmarks, etc.
- ⇒ Favoriser managed beans de scope Request
- Mais : conserver un état sur le serveur peut s'imposer si état compliqué, ...

# Mémorisation de l'état : deux façons

## Client conserve l'état

- Client charge page `newitem.xhtml`
- Client soumet formulaire avec détails item
- App. renvoie `response.xhtml?faces-redirect=true&itemId=13`
- Client demande `response.xhtml?itemId=13`

## Serveur conserve l'état

- Client charge page `newitem.xhtml`, soumet formulaire
- Serveur retient item id dans bean `SessionScoped`
- App. renvoie `response.xhtml?faces-redirect=true`
- Client demande `response.xhtml`
- Serveur sait quel item montrer au client grâce au bean

## Ressources web

- Parfois nécessaire d'accéder (via Facelets par exemple) à des *ressources web*
- Ressource web : fichier image, script, ...
- À placer dans `web-root/resources/rid`
- Ou dans classpath : `META-INF/resources/rid`
- Rid : `[locale-prefix/][library-name/][library-version/]resource-name[/resource-version]`
- Exemple : `<h:outputStylesheet library="css" name="default.css" />` ⇒  
`web-root/resources/css/default.css`



# PROJECT\_STAGE

- Options de configuration via descripteur web
- Recommandé : mettre PROJECT\_STAGE à Development (par défaut : Production), valeurs permises : voir [ProjectStage](#), cf. [Application](#)
- Plus d'informations de débogage lors Exception

## Modification PROJECT\_STAGE

```
<web-app ...>
  <context-param>
    <param-name>javax.faces.PROJECT_STAGE
    </param-name>
    <param-value>Development</param-value>
  </context-param>
```

## Voir aussi

- Utiliser l'attribut "immediate" pour traiter certains composants avant les autres (utile pour passer outre erreurs de validation par exemple)
- Pour utiliser des templates, voir Facelets tag library (<http://xmlns.jcp.org/jsf/facelets>, préfixe ui:)
- Facilités de localisation
- Interventions avancées possibles via fichier `WEB-INF/faces-config.xml`
- Résultat action :  
`faces-redirect=true&includeViewParams=true`  
ajoute paramètres de la cible à requête de redirection
- `h:link` : lien sans action ( $\neq$  `h:commandLink`), génère élément a
- `h:button` : bouton sans action ( $\neq$  `h:commandButton`)
- Cible connue lors rendu initial et non lors du postback

# Descripteur web

- Préciser dans le descripteur quelles requêtes (quelles urls) sont adressées à JSF (à FacesServlet)
- Dans module web (.war)

Dans descripteur web WEB-INF/web.xml :

```
<web-app ...>
  <servlet>
    <servlet-name>my-faces-servlet-name</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet
    </servlet-class>
  </servlet>
  <servlet-mapping>
    ...
  </servlet-mapping>
</web-app>
```

## Descripteur web : association préfixe

Association préfixe recommandée (existe aussi association suffixe) :

```
<web-app ...>
  <servlet-mapping>
    <servlet-name>my-faces-servlet-name</servlet-name>
    <url-pattern>/faces/*</url-pattern>
  </servlet-mapping>
</web-app>
```

- Distingue requêtes *faces* (exemple `contextPath/faces/view.xhtml`) de requêtes non-faces (exemple `contextPath/view.xhtml`)

# Références

- [Facelets tags](#)
- Exemples du tutorial Java EE 7 : voir par exemple `web/jsf/guessnumber-jsf`
- [Livre](#) [JavaServer Faces 2.0](#)
- The Java EE Tutorial: [JavaServer Faces Technology](#)
- [Try it](#) [HTML Editor](#) ([formulaires](#))
- [JSR 344](#) (JavaServer Faces 2.2) ([zip complet](#), [PDF principal](#))
- [HTML\\_BASIC RenderKit](#) (information redondante, voir plutôt [Facelets tags](#))
- [JSR 245](#) (JavaServer Pages 2.3, contient Unified Expression Language) ([direct](#))

# Exercices I

- Créer un projet JSF (ou ajouter facette JSF à un projet web existant)
- Configurer descripteur web pour qu'il serve les requêtes Faces
- Créer une page Facelet `index.xhtml` conforme à la syntaxe Facelet (suivre recommandations pour avoir l'en-tête correct) mais sans composants actifs (sans tags Facelets). La page affiche simplement "Hello".
- Publier le projet sur le serveur
- Naviguer vers cette page pour qu'elle soit considérée comme une requête Faces (comment ?)
- Vérifier qu'elle a bien été servie par le servlet Faces en regardant le code source de la page reçue par votre navigateur : le doctype résultant doit avoir été transformé en `<!DOCTYPE html>`

## Exercices II

- Modifier la page `index.xhtml` : elle doit contenir un tag `h:outputText` qui est chargé de générer le texte "Hello". Pour le moment ce texte est toujours inscrit en dur dans la page.
- Observer le résultat (vous savez quoi faire à ce stade...)
- Reprendre le cycle de traitement. Décrire ce qui se passe à chaque étape sur cet exemple.
- Créer un managed bean `Greeter` `@RequestScoped` et une méthode `getGreeting()` qui renvoie "Hello".
- Ajouter ce qu'il faut pour que votre `Greeter` soit accessible par EL
- Modifier la page (vous le voyiez venir ?) : au lieu d'écrire "Hello" en dur, le tag doit invoquer la méthode `getGreeting()` pour générer le texte
- Vérifier que ça fonctionne

## Exercices III

- Décrire ce qui se passe à chaque étape du cycle de vie après ces modifications
- Faire en sorte que la page affiche "Hello, " suivi de `myname` quand on appelle `index.xhtml?name=myname` (astuce : enregistrer la valeur envoyée dans une propriété de `Greeter`, vérifier que ceci fonctionne puis la réutiliser)
- En supposant maintenant qu'on appelle `index.xhtml?name=myname&feelgood=true`, faire en sorte que l'état de bien-être binaire passé en paramètre s'enregistre dans une propriété de type `Boolean` de `Greeter`. Utiliser pour ce faire les mécanismes de conversion automatique de JSF (astuce : faites au plus simple).
- Modifier la page pour que JSF affiche une erreur quand le nom manque, en utilisant le mécanisme de validation de JSF. (Astuce : jeter un œil au tag `h:messages`.)



## Exercices IV

- Logger les types des composants composant l'arbre de composants (en commençant par sa racine, `UIViewRoot`) juste avant le rendu de la page (astuce : faire en sorte qu'une méthode de `Greeter` — ou d'un autre bean créé à cet effet — soit exécutée juste avant le rendu)
- Créer un managed bean `ZeList` qui contient une liste `fakeDb` de strings inscrite en dur dans votre application, dans un champ immutable (par exemple `private static List<String> fakeDb = Arrays.asList("string 1", "blah", "hey", ...);`) et dont la responsabilité est simplement se servir cette liste. Quel est le scope adéquat pour ce bean ?

## Exercices V

- Créer une page `search.xhtml` contenant un seul contrôle texte et un bouton Search. Lorsque l'utilisateur clique Search il voit tous les strings de `fakeDb` contenant la chaîne qu'il a introduit dans le contrôle texte. Créer un bean à cet effet et y injecter `ZeList`.

## Phase *Restore View*

- Associe l'url à un `viewId`
- ⇒ Requête `contextPath/faces/view.xhtml` associée à `viewId`  
`view.xhtml`
- Si requête initiale : lit la page `viewId`
- Construit un arbre de *composants JSF*
- ⇒ Construit composant `HtmlInputText`, ...
- Si arbre existe déjà dû à requête précédente : le restaure

## Phase *Apply Request Values*

Name:

- Place valeurs envoyées dans composants JSF implémentant

`EditableValueHolder`

⇒ Place "Mon nom" dans composant `HtmlInputText`

- Retient quel composant ou composants a été activé implémentant

`ActionSource2`

⇒ Utilisateur a cliqué sur « Submit » : événement associé à composant `HtmlCommandButton`

⇒ `ActionEvent` de source `HtmlCommandButton` placé dans queue

## Phase *Process Validations*

- Convertit et valide les valeurs des composants
- Selon convertisseurs et validateurs enregistrés
- Messages d'erreurs enregistrés dans le `FacesContext` cf.  
`addMessage()`
- Si erreur de conversion ou validation : passe directement à *Render Response*

# Contexte

- Un `FacesContext` associé à chaque requête Faces
- Modifié au cours du cycle de traitement de la requête
- Stocke la vue (arbre de composants)
- Contient les messages d'erreur durant validation, conversion...
- Accès via `FacesContext.getCurrentInstance()`

## Phase *Update Model Values*

- À ce stade, requête valide, valeurs composants ok
  - Composants pouvant être édités implémentant `EditableValueHolder` : appelle la méthode liée à propriété `value`
- ⇒ Appelle `"#{myBean.text}"`
- Désigne la méthode `setText` d'une instance de la classe `MyBean`

## Phase *Invoke Application*

Dans cette phase : traitement des actions et navigation

Name:

- Si un composant a été activé, on l'a retenu en phase *Apply Request Values* ([ActionEvent](#) issu de composants implémentant [ActionSource2](#))
  - Si on vient d'arriver sur la page : rien à faire à cette phase
  - Si composant activé : méthode liée à action invoquée
- ⇒ Développeur a enregistré `#myActor.processSubmit()` comme action du `HtmlCommandButton`
- ⇒ `processSubmit()` : String enregistre le nom dans la BD
- ⇒ ... et renvoie "success" (ou "failure" si échec)
- Conteneur récupère String de l'action ou invoque `toString()` et déclenche navigation vers cette `viewId`
  - Donc retour à phase 1 !



## Actions : Exemple Facelets

Page index.xhtml (extrait) :

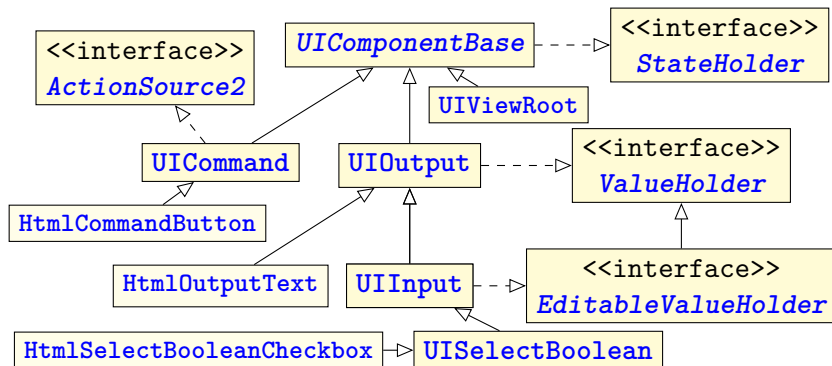
```
<body>  
  <h:commandButton value="Submit" action="response" />  
</body>
```

- L'action peut aussi être un simple string
- ⇒ Cliquer sur Submit envoie à la page `response.xhtml`

## Phase *Render Response*

- Chaque composant de l'arbre contribue à encodage réponse
- Délégation aux renderers
- Renderer et composant collaborent pour afficher le texte désiré
- Exemple : `Renderer HTML Output-Text` affiche dans le cas le plus simple l'attribut `value` du composant

# Composants



- Composants (ici, uniquement composants UI) : les briques des vues
- **UIComponentBase** : état et comportement par défaut
- **UIOutput** : affiche typiquement une sortie sur la page
- **UIInput** : prend une entrée utilisateur

## Composants et actions

- `StateHolder` : a un état à sauver
- `ActionSource2` peut entraîner navigation vers nouvelle vue
- `ValueHolder` maintient une valeur et fournit accès à des données dans le niveau modèle
- `EditableValueHolder` : validation, événements de changement de valeur
- On peut enregistrer un convertisseur sur `ValueHolder`
- JSF propose des convertisseurs par défaut
- Composant `ValueHolder` associé à un objet côté serveur et affiché sur client à l'aide du convertisseur
- On peut enregistrer un validateur sur `EditableValueHolder` (validation avant mise à jour du modèle)
- JSF fournit des `Validator`

# Hiérarchie de composants

- Racine : `UIViewRoot`
- Représente la vue décrite en VDL par le développeur
- Composants associés à listeners, etc.
- Seront utilisés par les phases ultérieures
- Composants responsables de l'invocation du code développeur
- Pas d'interaction explicite avec ces composants pour usage habituel

# Rendu

- Un composant UI est associé à un **Renderer** si delegated implementation
- Renderers contenus dans une bibliothèque fournie par conteneur ou tierce partie
- Fournisseur donne éventuellement des composants UI supplémentaires
- Fournisseur implémente **RenderKit** associant les composants et les renderers
- Une application peut déclarer (dans descripteur) un **RenderKit** différent du standard
- Les composants et les renderers ont des attributs configurables
- Exemple, composant `UIInput` : attributs `id`, `value`...
- Exemple, renderer hypothétique `TextBox` : attribut `width`
- Rendu standard ?

# Rendu HTML

- JSF fournit Standard HTML RenderKit et composants HTML
- Composants HTML particularisent les composants UI de base
- Composants base associés à un ou plusieurs modes de rendu
- JSF fournit un Renderer et un composant HTML pour chaque association [composant de base, mode de rendu]
- Exemple : UICommand ?

# Rendu HTML

- JSF fournit Standard HTML RenderKit et composants HTML
- Composants HTML particularisent les composants UI de base
- Composants base associés à un ou plusieurs modes de rendu
- JSF fournit un Renderer et un composant HTML pour chaque association [composant de base, mode de rendu]
- Exemple : UICommand ?
  - Renderer `Button` + Composant `HtmlCommandButton`
  - Renderer `Link` + Composant `HtmlCommandLink`



# Rendu HTML

- JSF fournit Standard HTML RenderKit et composants HTML
- Composants HTML particularisent les composants UI de base
- Composants base associés à un ou plusieurs modes de rendu
- JSF fournit un Renderer et un composant HTML pour chaque association [composant de base, mode de rendu]
- Exemple : UICommand ?
  - Renderer `Button` + Composant `HtmlCommandButton`
  - Renderer `Link` + Composant `HtmlCommandLink`
- `UIInput`  $\Rightarrow$  Renderers `File`, `Hidden`, `Secret`, `Text`, `Textarea`; Composants `HtmlInputFile`, ...
- `UISelectBoolean`  $\Rightarrow$  Renderer `Checkbox` + Composant `HtmlSelectBooleanCheckbox`

## Phase *Restore View*

La spécification JSF définit des classes et leurs comportements.  
Examinons le démarrage d'une requête JSF.

- input déterminé depuis Servlet
- Détermination de la vue via `ViewHandler` par défaut ou autre
- `deriveViewId(FacesCon. c, String input) ⇒ viewId`
- `viewId` : chemin (relatif contexte) page XHTML décrivant vue (cas Facelet)
- `restoreView(FacesC. c, String viewId): UIViewRoot`
- si null : `createView(F c, String viewId): UIViewRoot`
- `createView` utilise `ViewDeclarationLanguageFactory` → `vdl` puis `vdl.createView(FacesCon. c, String viewId)`
- À l'issue de la phase *Restore View* : hiérarchie de composants

## Phase *Apply Request Values*

- `UIViewRoot.processDecodes()`
- Appelle normalement les `processDecodes` de tout l'arbre
- Délègue à `render.decode()`
- `EditableValueHolder`  $\Rightarrow$  récupère valeur dans requête (non convertie sauf immediate)
- `ActionSource2` activé  $\Rightarrow$  `ActionEvent` dans queue
- Événements `ActionEvent` seront délivrés à la fin de *Invoke Application* ou de *Apply Request Values* si immediate

## Phase *Process Validations*

- À ce stade, composants `UIInput` connaissent leur valeur soumise
- `processValidators()` sur (normalement) chaque composant (propagés par `UIViewRoot`)
- `UIInput` : appelle `validate()` qui appelle `getConvertedValue` qui appelle `Renderer.getConvertedValue` puis `validateValue`
- `validateValue` appelle `validate()` sur les `Validator` enregistrés
- `validateValue` appelle la méthode liée à propriété `validatorBinding` (deprecated)
- Messages d'erreurs enregistrés dans le `FacesContext` cf. `addMessage()`

## Phase *Update Model Values*

- À ce stade, requête valide, valeurs composants ok
- `processUpdates()` sur (normalement) chaque composant (propagés par `UIComponent`)
- `UIInput` : appelle `updateModel()` qui appelle la méthode liée à propriété `value`

## Phase *Invoke Application*

Dans cette phase : traitement des `ActionEvent` et navigation

- Événement `ActionEvent` issu de composant « source »  
(implémentant `ActionSource2`)
- Méthode `fromAction` liée à `actionExpression` de « source »
- `ActionEvent` traités par `ActionListener` par défaut sauf  
remplacement maison
- `Invoke action`  $\Rightarrow$  `toString()`  $\Rightarrow$  `String`: `logicalOutcome`

## Phase *Invoke Application* : Navigation

- Récupère instance de `NavigationHandler` par défaut sauf remplacement maison
- Invoque `handleNavigation(FacesContext c, String fromAction, String logicalOutcome)`
- `navigationHandler` indique (facultativement) une nouvelle vue à rendre ou flow node à sélectionner
- `logicalOutcome = null` ou `""` : ré-affiche vue actuelle

## Navigation (détails)

- `outcome (blah?t=1) ⇒ queryString (?t=1) / out (blah)`
- `isRedirect := faces-redirect=true ∈ queryString`
- `includeViewParams := includeViewParams=true` ou `faces-include-view-params=true ∈ queryString`
- Si sans extension : `out += extension` du `viewId` actuel
- Si `out` relatif : `out := chemin viewId actuel (jusqu. dernier /) + out`
- Récupérer `viewHandler` par défaut ou remplacement maison
- `viewHandler.deriveViewId(FacesContext c, String out) ⇒ implicitViewId` supposé non `null`
- Si `isRedirect` : envoi HTTP redirect (vers `implicitViewId + queryString`) au client via `viewHandler.getRedirectUrl` ; `ResponseComplete`
- Sinon : `viewHandler ⇒ crée vue implicitViewId`



## Spécifications pages Facelet

- Une page Facelet doit se conformer au DTD [XHTML-1.0-Transitional](#)
- Éléments dans namespace XHTML (<http://www.w3.org/1999/xhtml>) rendus tels quels
- Modes de traitement : HTML 5 (`<process-as>html5</process-as>` dans `faces-config`), Facelets XHTML (`xhtml`), XML View (`xml`), et Facelets JSPX (`jspx`)
- En mode html 5, déclaration XML et DOCTYPE rendues ssi présentes dans le fichier VDL spécification ambiguë
- En mode html 5, XML Doctype simplifié en `<!DOCTYPE html>`

# Rappels XHTML 1.x

- **XHTML 1.0** : reformulation de HTML 4 compatible XML 1.0
- 3 DTDs correspondant aux DTDs HTML 4
- Sémantique : cf. HTML 4
- Déclaration XML : jamais obligatoire, mais fortement encouragée pour tous documents XHTML
- DOCTYPE obligatoire avant la racine : `<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">`  
(ou Transitional ou Frameset)
- Racine : `<html xmlns="http://www.w3.org/1999/xhtml" ...>`
- Media type : `text/html` ou `application/xhtml+xml`
- XHTML 1.1 : relâche compatibilité HTML 4, plus strict  
(`<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">`)

# HTML 5

Spec HTML 5 :

- Langage, API interaction avec représentation mémoire (DOM)
- Deux syntaxes concrètes pour transmission dans ce langage

## Syntaxe HTML version 5.0 ("HTML 5")

- Recommandée
- Media type `text/html`
- Pas d'en-tête `xml`
- `<!DOCTYPE html>` puis élément racine `html` ou DOCTYPE obsolètes

## Syntaxe XHTML version 5.0 ("XHTML 5")

- Media type XML tel que `application/xhtml+xml`
- DOCTYPE non obligatoire pour syntaxe XHTML 5

# Licence

Cette présentation, et le code LaTeX associé, sont sous [licence MIT](#). Vous êtes libres de réutiliser des éléments de cette présentation, sous réserve de citer l'auteur.

Le travail réutilisé est à attribuer à [Olivier Cailloux](#), Université Paris-Dauphine.