JSF
(*Jakarta Faces*)

ITU - Université Côte d'Azur
Richard Grin
Version 2.78 – 4/4/24

1

Plan de ce cours

- Présentation de JSF
- Architecture des applications JSF
- Exemple de page JSF
- Backing bean et portée CDI
- Cycle de vie JSF
- Configuration
- Navigation
- PRG
- Messages d'erreur ou information
- Événements
- Ajax
- JSF et HTML5
- Ressources
- Flots
- Bibliographie

2

Présentation de JSF

3

JSF

- Framework Java server-side pour développer des interfaces utilisateur pour les applications Web
- Utilise Facelets, langage de déclaration de page (format XHTML) pour écrire des pages JSF
- Une page JSF décrit une vue JSF qui est un arbre de composants Java
- Les pages et vues JSF sont enregistrées sur le serveur HTTP et sont transformées en pages HTML pour être envoyées sur le client HTTT

4

Expression Language

- EL (*Expression Language*) : spécification Jakarta qui définit un langage utilisé par Facelets pour désigner des propriétés ou des méthodes Java
- Dans les pages JSF les expressions EL sont encadrées par **#{ }**

5

EL et backing bean

- Une expression EL désigne le plus souvent les propriétés ou méthodes d'un *backing bean*, classe Java liée à la page JSF :
  - une propriété du backing bean représente une donnée dynamique
  - une méthode du backing bean représente un traitement à exécuter

6

Exemple - page JSF `index.xhtml`

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="jakarta.faces.html">
  <h:head> <title>Présentation</title> </h:head>
  <h:body>
    <h3>Présentation</h3>
    <h:form>
      Nom : <h:inputText value="#{utilisateur.nom}"/>
      <h:commandButton value="Enregistrer"
        action="#{utilisateur.direHello()}"/>
    </h:form>
  </h:body>
</html>
```

Espace de noms par défaut

Espace de noms XML pour JSF

Vous devinez comment cette page sera utilisée ?

7

Exemple - page JSF `index.xhtml`

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="jakarta.faces.html">
  <h:head> <title>Présentation</title> </h:head>
  <h:body>
    <h3>Présentation</h3>
    <h:form>
      Nom : <h:inputText value="#{utilisateur.nom}"/>
      <h:commandButton value="Enregistrer"
        action="#{utilisateur.direHello()}"/>
    </h:form>
  </h:body>
</html>
```

Formulaire

utilisateur : nom backing bean

Propriété nom du backing bean

Bouton soumet le formulaire par une requête POST envoyée au serveur

Méthode `direHello` du backing bean exécutée pour le traitement du formulaire

8

Exemple - Backing bean

```
@Named
@RequestScoped
public class Utilisateur {
  private String nom;
  public String getNom() {
    return nom;
  }
  public void setNom(String nom) {
    this.nom = nom;
  }
  public String direHello() {
    return "hello";
  }
  ...
}
```

Nom par défaut : « utilisateur »

`@Named` permet au backing bean d'être utilisé dans une expression EL

`#{utilisateur.xxx}` dans page JSF provoque la création par CDI d'une instance du bean, s'il n'y en a pas déjà une

9

Exemple - Backing bean

```
@Named
@RequestScoped
public class Utilisateur {
  private String nom;
  public String getNom() {
    return nom;
  }
  public void setNom(String nom) {
    this.nom = nom;
  }
  public String direHello() {
    return "hello";
  }
  ...
}
```

Portée du bean ; il sera supprimé à la fin de la requête

```
<h:inputText
  value="#{utilisateur.nom}"/>
```

Propriété nom

```
<h:commandButton value="Enregistrer"
   action="#{utilisateur.direHello()}"/>
```

Nom prochaine page à afficher

10

Exemple - page JSF `hello.xhtml`

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="jakarta.faces.html">
  <h:head> <title>Hello</title> </h:head>
 <h:body>
    <h3>Hello</h3>
    Hello #{utilisateur.nom}
  </h:body>
</html>
```

Que fait cette page ?

Propriété nom du backing bean

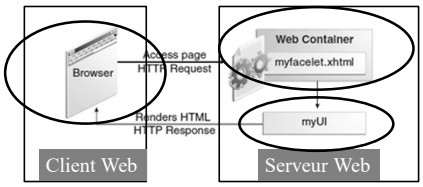
Est-ce que `nom` contiendra le nom tapé par l'utilisateur ?

Quand la page HTML est créée sur le serveur, pour être retournée à l'utilisateur, la requête POST est toujours en cours de traitement. C'est donc la même instance du backing bean que celle utilisée par `index.xhtml`.
Donc `nom` contient le nom saisi par l'utilisateur.

11

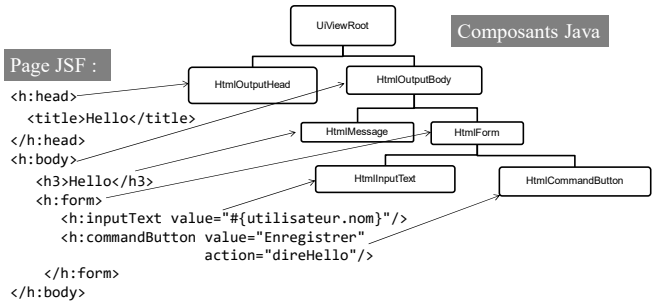
Page JSF, composants Java, HTML

- Page JSF qui contient des balises (`<h:inputText>`, …)
- Ces balises décrivent des composants Java
- La page JSF est traduite en une page HTML retournée au client HTTP



12

Page JSF – « Vue » Java



13

Principaux services rendus par JSF

- Conversion entre les ***textes*** saisis par l'utilisateur et les ***valeurs Java*** du serveur
- Validation des données saisies par l'utilisateur
- Automatisation de l'affichage des messages d'erreur si problèmes de conversion ou de validation
- Internationalisation
- Support d'Ajax sans programmation
- Templates graphiques
- Librairies de composants

14

Les traitements

- Grâce aux expressions EL, séparation nette entre
  - la structure et les composants des pages JSF
  - les données de l'application
  - les traitements à exécuter
- Même les traitements directement liés à l'interface utilisateur sont effectués *en dehors de la page JSF*, par du code Java (*backing bean*)

15

Architecture des applications JSF

16

Rappel : répartition des tâches

- Pages JSF pour l'interface avec l'utilisateur ; ne contiennent aucun traitement
- Backing beans pour les traitements liés directement à l'interface utilisateur
- Backing beans font appel à d'autres classes Java (souvent beans CDI) pour les autres traitements (traitements métier, accès aux sources de données,…)
- Accès aux bases de données utilisent souvent JPA et donc des classes entités

17

Backing bean

- Souvent, mais pas obligatoirement, un backing bean par page JSF
- Géré par le container **CDI**

Qu'est-ce que ça signifie ?

Le container crée le backing bean et le supprime

A quel moment le backing bean est créé par le container ?

La 1ère fois qu'il y a une référence au backing bean

A quel moment le container supprime le bean ?

A la fin de sa portée

18

Code d'un backing bean

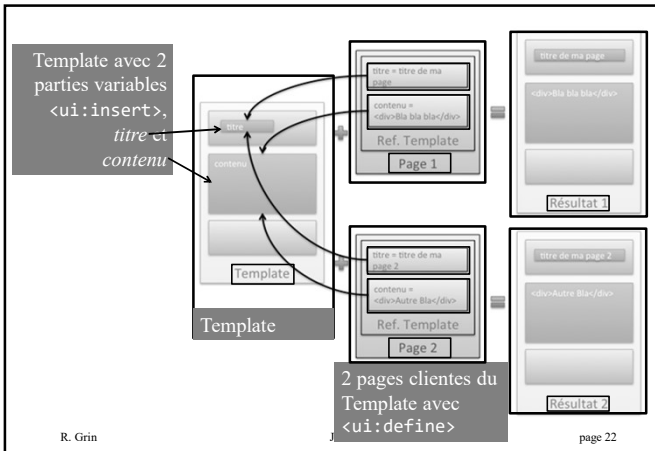
- Propriétés pour valeurs à afficher ou saisies par l'utilisateur :
```
<h:inputText value="#{utilisateur.nom}"/>
```
setter et *getter* utilisés en interne
- Traitements lancés par l'utilisateur, par un clic sur un bouton par exemple :
```
<h:commandButton value="Enregistrer"
   action="#{utilisateur.direHello()}"/>
```
- Expressions qui indiquent si un composant doit être affiché :
```
<h:inputText
   value="#{bean.client.pourcentageRemise}"
   rendered="#{bean.client.bonClient}" />
```
code semblable avec `disabled`

Servlet « Faces »

- Le servlet JSF gère le cycle de vie JSF, fondamental pour JSF
- Ce servlet est fourni par JSF ; il est caché au développeur
- Un mapping indique les requêtes qui seront traitées par ce servlet ; par défaut celles avec les URLs `/faces/*` ou `*.jsf` ou `*.faces` ou `*.xhtml` ; correspond toujours à un fichier `*.xhtml` (par exemple, `page.jsf` correspond à `page.xhtml`) ; mapping modifiable dans web.xml

Template

- Les pages d'un site Web respectent le plus souvent une charte graphique
- L'aspect commun aux pages est défini dans un fichier XHTML à part, appelé *template*
- Un template est une page JSF avec des parties « variables » identifiées par un nom :
```
<ui:insert name="header"/>
```
- Une page JSF qui utilise ce template (cliente du template) a l'aspect défini par le template ; elle définit seulement les parties variables :
```
<ui:define name="header">...</ui:define>
```
Contenu de *header*



Conversion et validation

- Les données saisies dans les formulaires HTML arrivent sur le serveur sous la forme de chaînes de caractères (paramètres des requêtes HTTP)
- Il faut les convertir dans le bon type Java (celui de la propriété du backing bean) :
```
<h:inputText id="age" value="#{bean.age}" >
```
- Elles doivent ensuite être validées (18 < age < 70, par exemple) **avant** d'exécuter les traitements qui les utilisent

Convertisseurs et validateurs

- La conversion peut être implicite ; par exemple la `String` "12" convertie en nombre entier 12
- La conversion ou la validation peut être effectuée par un convertisseur ou validateur standard fourni par JSF
- Le développeur peut aussi écrire son propre code pour convertir ou valider

Exemple de convertisseur standard

```
<h:inputText id="dateNaissance"
             value="#{employe.dateNaissance}">
    <f:convertDateTime type="localDateTime"
                       pattern="dd/MM/yyyy"/>
</h:inputText>
<h:message for="dateNaissance" />
```

Si erreur de conversion, le message est affiché ici

Utilisable aussi avec `<h:outputText>`

Selon le type Java de la propriété `dateNaissance`

Convertisseur non standard

- Classe annotée par `@FacesConverter` dont le paramètre `value` identifie le convertisseur : `@FacesConverter("fr.unice.Pays")`
- Implémente l'interface `jakarta.faces.convert.Converter<T>`
- Cette interface contient 2 méthodes
  - `T getAsObject(FacesContext ctx, UIComponent composant, String valeur)`
  - `String getAsString(FacesContext ctx, UIComponent composant, T valeur)`
- Elles doivent lancer une `ConverterException` si la conversion n'est pas possible (jakarta.faces.convert)

Validateur non standard

- Classe annotée par `@FacesValidator` dont le paramètre `value` identifie le validateur
- Implémente l'interface `jakarta.faces.convert.Validator<T>`
- Cette interface contient la méthode
  - `void validate(FacesContext context, UIComponent composant, Object valeur)`
    La méthode lance une `ValidatorException` si la valeur n'est pas validée
- Comme pour un convertisseur, le message d'erreur de l'exception sera affiché (si `<h:message>`)

Convertisseur / validateur avec injection CDI

- Si on veut injecter des beans CDI dans une classe convertisseur (ou validateur), il faut ajouter l'attribut `managed` à l'annotation `@FacesConverter` (ou `@FacesValidator`) avec la valeur `true`

Exemple convertisseur

id convertisseur

```
@FacesConverter(value = "discountConverter", managed = true)
public class DiscountConverter implements Converter<Discount> {
  @Inject
  private DiscountManager discountManager;
  @Override
  public Discount getAsObject(FacesContext context,
      UIComponent component, String value) {
    if (value == null) { return null; }
    return discountManager.findById(value);
  }
  ...
}
```

Javadoc :
- `getAsObject` retourne `null` si valeur à convertir est `null`
- `getAsString` retourne chaîne vide si valeur est `null`

`getAsObject` transforme quel type en quel type ?

Comment ?

Désigner convertisseur et validateur

- L'identificateur peut être utilisé dans
  - un attribut `converter` ou `validator`
  - une balise `<f:converter>` ou `<f:validator>`

Exemples désigner convertisseur

```
<h:selectOneMenu id="discount"
    value="#{clientDetailsBean.client.discount}">
  <f:converter converterId="discountConverter"/>
  <f:selectItems value="#{clientDetailsBean.discounts}"
      var="discount"
      itemValue="#{discount}"
      itemLabel="#{discount.id} - #{discount.taux} %"/>
</h:selectOneMenu>
```

Qui peut expliquer ?

```
<h:selectOneMenu id="discount"
      value="#{clientDetailsBean.client.discount}"
      converter="discountConverter">
  <f:selectItems
     ...
```


31

Validation des beans

- La validation peut aussi se faire en annotant le champ d'un backing bean ou d'une classe entité JPA, qui va recevoir la valeur

```
@Min(18) @Max(70)
private int age;
```


32

Composants des pages JSF

- De nombreux composants standards correspondent aux balises HTML :
  - saisie des données `<h:inputText>`
  - listes déroulantes `<h:selectOneMenu>`
  - boutons `<h:button>` ou `<h:commandButton>`
  - tables `<h:dataTable>`
  - …
- Pour améliorer ces composants ou pour en avoir de nouveaux le développeur peut
  - utiliser des bibliothèques de composants
  - créer ses propres composants

33

Exemple plus complet de page JSF

34

En-tête

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="jakarta.faces.html"
      xmlns:f="jakarta.faces.core">
<h:head>
  <title>Inscription</title>
  <h:outputStylesheet name="css/styles.css"/>
</h:head>
```

Fichier CSS ; chemin par rapport au répertoire `/resources`

35

Panel et liste déroulante

```
<h:body>
  <h3>Inscription</h3>
  <h:form>
    <h:panelGrid columns="3"
        columnClasses="rightalign,leftalign,leftalign">
C1  <h:outputLabel value="Titre : " for="titre"/>
C2  <h:selectOneMenu id="titre" label="Titre"
                     value="#{inscriptionBean.titre}" >
      <f:selectItem itemLabel="" itemValue="#{null}"/>
      <f:selectItem itemLabel="M." itemValue="M"/>
      <f:selectItem itemLabel="Mme" itemValue="Mme"/>
    </h:selectOneMenu>
C3  <h:message for="titre"/>
```

3 colonnes : intitulé, zone de saisie, message d'erreur éventuel

Classes définies dans fichier CSS

Choix un item

Inscription

Titre :

36

Liste déroulante

```
<h:body>
  <h3>Inscription</h3>
  <h:form>
    <h:panelGrid columns="3"
         columnClasses="rightalign,leftalign,leftalign">
      <h:outputLabel value="Titre : " for="titre"/>
      <h:selectOneMenu id="titre" label="Titre"
                       value="#{inscriptionBean.titre}" >
        <f:selectItem itemLabel="" itemValue="#{null}"/>
        <f:selectItem itemLabel="M." itemValue="M"/>
        <f:selectItem itemLabel="Mme" itemValue="Mme"/>
      </h:selectOneMenu>
      <h:message for="titre"/>
```

Choix utilisateur rangé dans propriété backing bean

Items

37

Nom et email (avec validateur)

```
<h:outputLabel value="Nom :" for="nom"/>
<h:inputText id="nom" label="Nom"
      required="true"
      value="#{inscriptionBean.nom}" />
<h:message for="nom" />
<h:outputLabel value="Email :" for="email"/>
<h:inputText id="email" label="Email"
      required="true"
      value="#{inscriptionBean.email}">
  <f:validator validatorId="validateurEmail"/>
</h:inputText>
<h:message for="email" />
```

Saisie du nom

Message si nom pas saisi

Saisie email

Validateur spécial pour email, écrit dans l'application

38

Bouton de commande

```
      <h:panelGroup/>
      <h:commandButton id="enregistrement"
          value="Enregistrer"
          action="#{inscriptionBean.confirmer()}" />
    </h:panelGrid>
  </h:form>
 </h:body>
</html>
```

Bouton pour soumettre formulaire

Méthode du bean lancée après le traitement des données saisies

Quelle page sera retournée en réponse à la soumission du formulaire ?

39

CSS et JSF – fichiers des styles

- Dans la balise `<h:head>`, utiliser `<h:outputStylesheet>` à la place de `<link rel="stylesheet">` de HTML :
  `<h:outputStylesheet name="css/style.css" />`
- JSF va chercher le fichier dans le répertoire `resources` placé sous la racine des pages Web ; dans `resources/css/style.css` pour l'exemple

40

CSS et JSF – style d'un composant

- Pour donner du style à un composant JSF, on peut aussi utiliser l'attribut `styleClass` (ou `style` pour du CSS intégré) ou un des attributs particuliers à un composant (comme `columnClasses` de `<h:panelGrid>`)
- Attention si vous utilisez `#id` dans le fichier CSS : l'id d'un composant JSF ne correspond pas à l'id du composant HTML envoyé au client

41

Autres ressources

- Fichiers JavaScript (dans `<h:head>`) : `<h:outputScript>`
- Images : `<h:graphicImage>`
- Désignent des fichiers sous `/resources`

42

Backing bean
et portée CDI (*scope*)

43

Backing bean

- Bean CDI utilisé dans une page JSF
- Doit être annoté par `@Named`

44

`@Named`

- `@Named` donne un nom « EL » (*Expression Language*) au bean pour l'utiliser dans une page JSF

```
@Named // ou bien @Named("autreNom")
@RequestScoped
public class Bean { … } // contient propriété nom
```

- Une page JSF peut désigner le bean :

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

Valeur de la *propriété* `nom` du bean (utilisation de `getNom` et `setNom`)

45

Bean CDI dans une page JSF

- Quand une page JSF désigne un bean, par exemple `#{inscriptionBean.nom}` le container CDI regarde s'il existe un bean du même type dans la portée
- Si c'est le cas, le bean existant est utilisé par la page JSF
- Sinon un nouveau bean est créé par le container et utilisé par la page JSF

46

Précaution pour getter

- Quand une page JSF utilise une propriété d'un backing bean, le getter de la propriété peut être appelé plusieurs fois
- Un getter doit donc éviter autant que possible d'exécuter à chaque appel un code coûteux, tel qu'un accès à une base de données

47

Exemple

Qui peut expliquer ?

```
@Named
@ViewScoped
public class CustomerBean implements Serializable {
  private List<Customer> customerList;
  @Inject
  private CustomerManager customerManager;

  public List<Customer> getCustomers() {
    if (customerList == null) {
      customerList = customerManager.findAll();
    }
    return customerList;
  }
}
```


48

Utilité de la portée

- Détermine la durée de vie du backing bean
- Si la portée est la session, il va durer jusqu'à la fin de la session de travail de l'utilisateur ; si une autre expression EL y fait référence pendant la même session, c'est le même bean qui sera utilisé
- On peut ainsi partager des informations entre plusieurs requêtes, pages ou traitements
- Il faut choisir la bonne portée, pour ne pas encombrer la mémoire du serveur, tout en permettant le partage d'information

49

Rappel des portées CDI

- `Dependent` : (portée par défaut) nouvel objet créé à chaque référence ; l'objet est supprimé quand l'objet dans lequel il est injecté est supprimé
- `RequestScoped` : requête HTTP
- `ViewScoped` : associé à une vue (page JSF)
- `FlowScoped` : associé à un ensemble de vues
- `ConversationScoped` : début et fin par code Java
- `SessionScoped` : session de travail de l'utilisateur
- `ApplicationScoped` : durée de vie de l'application ; partagé entre tous les utilisateurs

50

`Serializable`

- Un backing bean qui a une autre portée que `RequestScoped` et `Dependent` doit implémenter `Serializable` car il peut être retiré temporairement de la mémoire centrale par le container

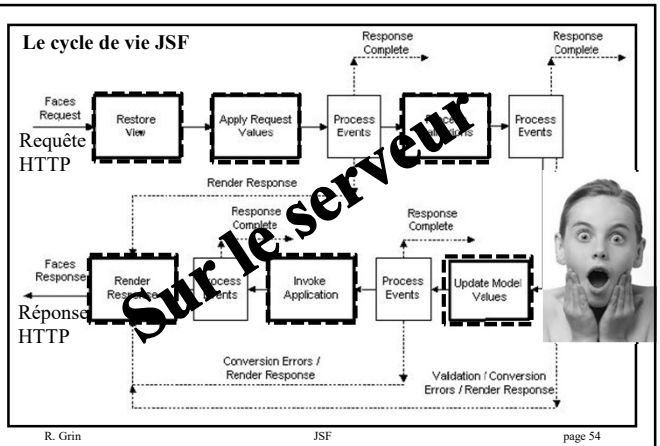
51

Le cycle de vie JSF

52

- Indispensable de bien comprendre tout le processus qui se déroule entre
  - la soumission d'un formulaire par l'utilisateur
  - la réponse du serveur sous la forme d'une page HTML

53

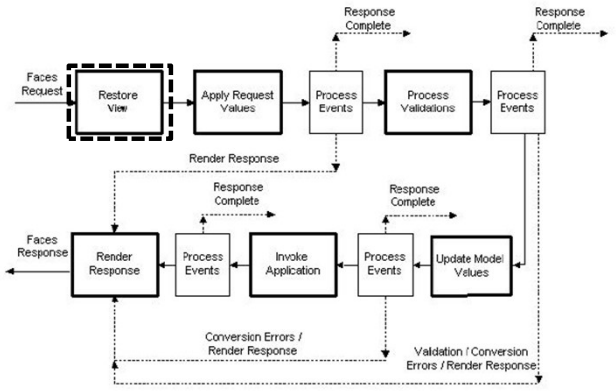


54

Soumission d'un formulaire

- Sur le poste client HTTP (navigateur), l'utilisateur a saisi des valeurs dans ce formulaire
- Ces valeurs sont mises en paramètres de la requête POST envoyée au serveur Web

55

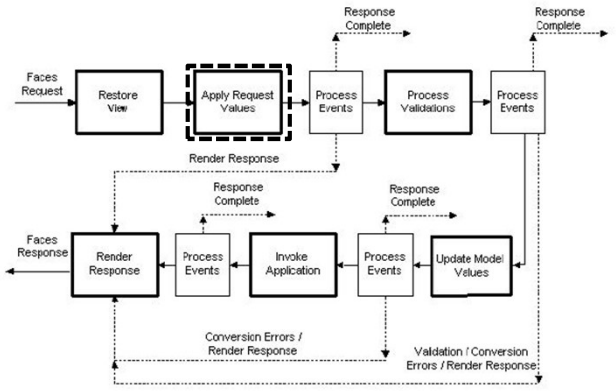


56

Phase de restauration de la vue

- La vue qui correspond à ***la page qui contient le formulaire*** est restaurée (phase « Restore View »)

57



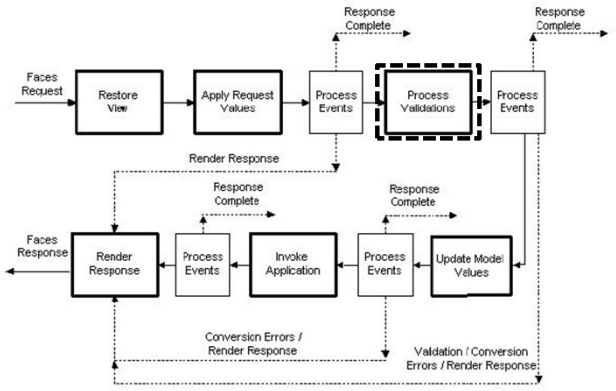
58

Phase d'application des paramètres

- Les valeurs des paramètres de la requête HTTP (saisies par l'utilisateur dans le formulaire) sont attribuées aux composants Java de la vue : phase « Apply Request Values »
- Par exemple, si l'utilisateur a saisi un âge dans un composant `<h:inputText>` du formulaire HTML, cet âge est mis dans une variable interne de l'instance Java qui représente le composant dans la vue

De quel type ?

59

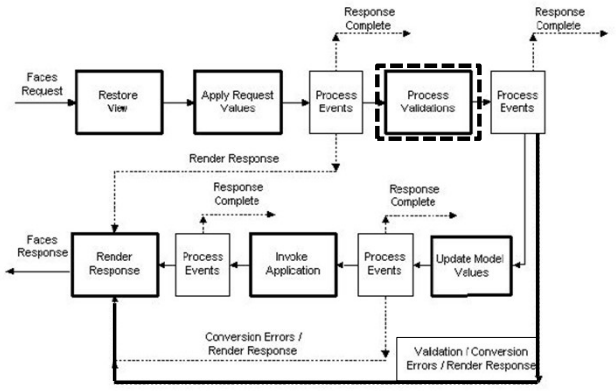


60

Phase de validation

- Les expressions EL sont utilisées pour savoir dans quelles propriétés Java mettre les valeurs récupérées à l'étape précédente :
  `<h:inputText value="#{bean.age}" ... />`
- Les données sont converties dans le type Java et ensuite validées pour voir si elles respectent les contraintes indiquées dans l'application (validateurs)
- Si une conversion ou validation échoue, la phase suivante sera la phase « Rendu de la réponse » qui retourne le formulaire avec tous les messages d'erreur

61

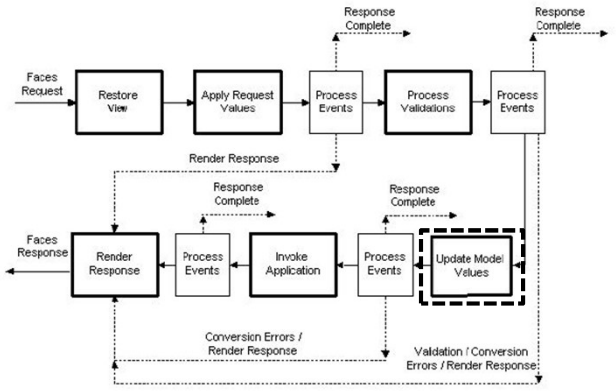


62

Validation sur le client HTTP

- Possible de valider avec une fonction JavaScript par une balise JSF personnalisée Intérêt ?
- Attention, ne remplace pas une validation sur le serveur Pourquoi ?

63



64

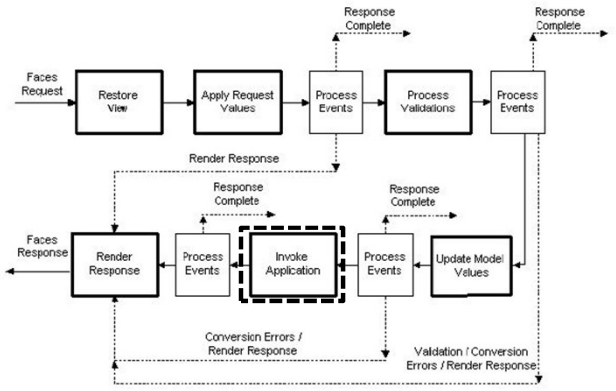
Phase de mise à jour du modèle

- Toutes les données ont été validées ; elles peuvent être mises dans les propriétés des *backing beans* associées (expressions EL) :

```
<h:inputText id="nom"
  value="#{bean.utilisateur.nom}" />
```

Quel code Java pour cette expression EL ?

65



66

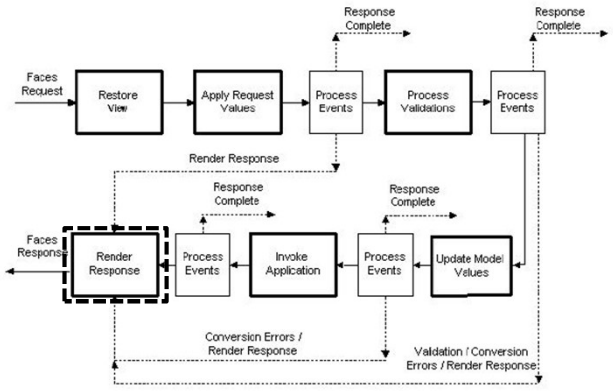
Phase d'invocation de l'application

- Maintenant que les données sont enregistrées dans les backing beans, l'action associée au bouton ou au lien cliqué par l'utilisateur peut être lancée ; une méthode des backing beans est exécutée :

```
<h:commandButton ...
  action="#{inscriptionBean.confirmer}" />
```

- La valeur de retour de cette méthode détermine la prochaine page à afficher (navigation)

67



68

Phase de rendu de la réponse

- La page déterminée par la valeur retour de la méthode « action » est codée en HTML et envoyée vers le client HTTP, en réponse à la requête POST

69

Cycle de vie pour une requête GET avec paramètres

- Si l'URL de la requête contient des paramètres `http://serveur/appli/adresse?p1=v1&p2=v2` et si la page JSF demandée contient des paramètres de vue (étudiés plus loin) pour ranger ces valeurs dans un backing bean, ce même cycle de vie est exécuté (conversion, validation des paramètres,…)
- Comme si les valeurs des paramètres de la requête avaient été saisies par un utilisateur dans un formulaire soumis avec une méthode POST

70

Configuration

71

Exemple de web.xml (1/2)

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="6.0"
   xmlns="https://jakarta.ee/xml/ns/jakartaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
https://jakarta.ee/xml/ns/jakartaee/web-app_6_0.xsd">
  <context-param>
    <param-name>jakarta.faces.PROJECT_STAGE</param-name>
    <param-value>Development</param-value>
  </context-param>
```

Production
pour livrer au client

72

Exemple de web.xml (2/2)

```
  <session-config>
    <session-timeout>30</session-timeout>
  </session-config>
  <welcome-file-list>
    <welcome-file>index.xhtml</welcome-file>
  </welcome-file-list>
</web-app>
```


73

Phase de projet

- Attention, la phase de projet `Development` ajoute automatiquement les messages d'erreurs, même s'il n'y a aucun composant pour les messages dans la page (`<h:message>` ou `<h:messages>`)
- Ne pas oublier de mettre ces composants dans la page pour que l'utilisateur voie les messages quand l'application sera en production

74

`beans.xml`

- Configuration pour CDI, par exemple pour indiquer le mode de découverte des beans
- Sous `WEB-INF` si fichier WAR ou sous `META-INF` sinon

75

`WEB-INF/faces-config.xml`

- Configuration pour JSF
- Par exemple pour indiquer des fichiers de textes pour l'internationalisation (`<resource-bundle>`) ou pour donner des règles de navigation

76

Configuration JSF

- Pour activer Jakarta Faces pour l'application, le plus simple est d'annoter un bean CDI avec `@FacesConfig` :

```
@ApplicationScoped
@FacesConfig
public class Config {
}
```

- D'autres configurations pourront être ajoutées à cette classe, par exemple pour la sécurité

77

Navigation

78

Navigation dans JSF

- Navigation = affichage d'une nouvelle page
- Composants JSF standards pour la navigation : `<h:commandButton>`, `<h:button>`, `<h:commandLink>`, `<h:link>`, `<h:outputLink>`
- `<h:commandButton>` et `<h:commandLink>` doivent être dans une balise `<h:form>` ; ils soumettent le formulaire
- Les autres composants peuvent ne pas être dans un formulaire et, s'ils le sont, ils ne le soumettent pas

POST ou GET

- `<h:commandButton>` et `<h:commandLink>` soumettent le formulaire par une requête POST et affichent la page indiquée par `action` :

```
<h:commandButton value="Voir liste"
                 action="liste" />
```

- `<h:button>` et `<h:link>` lancent une requête GET et affichent la page indiquée par `outcome` :

```
<h:link value="Voir les notes"
        outcome="/notes/voirNotes" />
```

- Pour ces 2 cas, JSF complète l'adresse de la page par `.xhtml` si elle n'a pas d'extension (`liste.xhtml` par exemple)
- `<h:outputLink>` lance une requête GET et affiche la page indiquée par `value` ; page peut être en dehors de l'application

Recommandation

- Requête GET si on veut juste afficher une page, par exemple afficher la liste de tous les clients
- Requête POST si la requête modifie des données sur le serveur ; par exemple si l'utilisateur a modifié le nom d'un client dans un formulaire

Adresse affichée par le navigateur

- Après une requête POST (`<h:commandButton>` et `<h:commandLink>`), le navigateur affiche l'URL du formulaire qui vient d'être soumis, alors que la nouvelle page est affichée
- Si on veut voir le bon URL, il faut ajouter une redirection
- Après une requête GET (`<h:button>` et `<h:link>`), le navigateur affiche le bon URL
- Vu dans le TP 2

Navigation statique et dynamique

- La valeur de l'attribut `action` qui détermine la nouvelle page peut être
  - définie « en dur » au moment de l'écriture de l'application ; la navigation est statique
  - définie par la valeur retournée par une méthode Java ; la navigation est dynamique

Exemple de navigation statique

- Dans page1.xhtml :

```
<h:commandButton
    action="page2"
    value="Aller à page2" />
```

Page suivante si clic sur bouton ?

Exemple de navigation dynamique

- Dans page1.xhtml :

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

- Dans le backing bean :

```
@Named
@RequestScoped
public class Bean {
  public String faire(){
    ...
    if (...) return "page2";
    else return "autrePage";
  }
```

Page suivante si clic sur bouton ?

85

Redirection

Qu'est-ce qu'une redirection HTTP ?

- Dans le cas d'une navigation statique, ajouter `?faces-redirect=true` à l'URL de la page à afficher :
  `action="page2?faces-redirect=true"`
- Dans le cas d'une navigation dynamique, ajouter à la valeur retournée par la méthode :
  `return "page2?faces-redirect=true";`
- En ce cas, la réponse à la requête ne retournera pas page2 mais plutôt un ordre de redirection vers page2

86

- Exercices 1 et 2 du TP 2 (modèle PRG)

87

PRG

88

Problèmes avec POST

- Un formulaire est dans une page **P1** ; la soumission du formulaire par une requête POST affiche une page **P2** en réponse
- 2 problèmes :
  - quand **P2** est affichée en réponse à la requête POST, le navigateur affiche l'adresse de **P1**
  - Si l'utilisateur fait alors un refresh de **P2**, le navigateur soumet à nouveau le formulaire

89

La raison du problème d'URL

- C'est JSF qui dirige vers la nouvelle page (à la fin de la phase « Invoke Application »)
- Le navigateur n'en a pas connaissance (navigation dynamique) ; il continue à afficher l'adresse du formulaire

90

Refresh ou *reload* d'une page

- Si l'utilisateur demande un *refresh* de la page **P2**, c'est qu'il veut la dernière version de l'information affichée dans la page
- Le navigateur envoie donc à nouveau la requête qui a permis d'obtenir la page
- Si la page a été affichée en réponse à une requête POST, le navigateur relance donc la requête POST avec les mêmes paramètres, ce qui correspond à une nouvelle soumission du formulaire

91

Les conséquences des 2 problèmes

- Impossible de garder un marque-page pour la page affichée après un POST
- Commande ou paiement involontaire dû à une soumission multiple d'un formulaire

Qu'est-ce qui est le plus grave ?

92

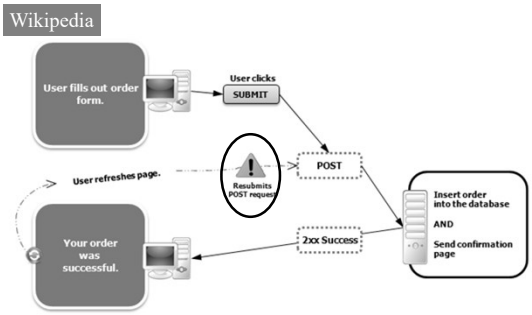
La solution : Post – Redirect - Get (PRG)

- Le modèle Post – Redirect - Get :
  - 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

93

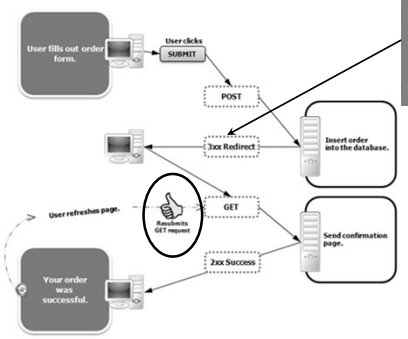
Sans PRG

Wikipedia



94

Avec PRG



Redirection : le navigateur doit charger la page spécifiée avec une requête GET

95

Problèmes de PRG

- La redirection peut poser un problème :
  - 2 requêtes à la place d'une seule. Si un backing bean est de portée requête, une nouvelle instance est utilisée pour la 2ème requête ; les données de la 1ère instance sont perdues
  - Les messages de succès ou d'information, générés par programmation Java, ne sont conservés que le temps d'une requête et donc n'apparaissent plus après une redirection
- Solutions dans transparents suivants

96

Portée session ?

- Une des solutions serait de ranger les informations dans la session plutôt que dans la requête
- Mais cette solution est **mauvaise** :
  - Elle peut conduire à une session trop encombrée
  - D'autres problèmes peuvent survenir ; par exemple, si l'utilisateur ouvre plusieurs onglets, et si ses actions dans les onglets utilisent le même bean de portée session pour des requêtes différentes

Solutions

- JSF offre des possibilités pour conserver les informations plus longtemps que la requête, sans passer en portée session
- Le plus intéressant est d'utiliser les paramètres de vue (permet de rester en portée « requête »)

Paramètres de vue

- Une requête GET avec 2 paramètres nom et age : http://serveur.fr/appli/page2.xhtml?nom=Bob&age=23
- Avec 2 paramètres de vue, les valeurs « Bob » et « 23 » peuvent être récupérées dans des propriétés d'un backing bean
- Un paramètre de vue est créé par une balise `<f:viewParam>` placée à l'intérieur d'une balise `<f:metadata>` dans la page demandée par GET

Exemple

Requête GET
http://serveur.fr/appli/page2.xhtml?nom=Bob

- Dans page2.xhtml (juste avant `<h:head>`) :

```
<f:metadata>
  <f:viewParam name="nom"
               value="#{bean.p1}" />
</f:metadata>
```

- « Bob » sera mis dans la propriété `p1` de `bean`, avant l'affichage de page2.xhtml
- Dans `<f:viewParam>`, comme dans `<h:inputText>`, on peut ajouter un convertisseur ou un validateur

Utilisation d'un paramètre de vue

- Permet de passer l'id d'un client de la page customerList.xhtml à la page details.xhtml :
  - Dans la table des clients de customerList.xhtml, lien GET :

```
<h:link outcome="details?idClient=#{item.id}"
        value="#{item.id}"/>
```

  - details.xhtml a un paramètre de vue pour récupérer l'id du client dans sa propriété `idClient` :

```
<f:metadata>
  <f:viewParam name="idClient"
               value="#{bean.idClient}"
</f:metadata>
```

Code extrait du TP 1

`<f:viewAction>`

- Balise d'une page demandée par GET ; doit être placée à l'intérieur de `<f:metadata>`
- Exécute une méthode d'un backing bean
- Exemple :
  `<f:viewAction action="#{bean.m()}"/>`
- Cette méthode est exécutée
  - après l'enregistrement par `<f:viewParam>` des valeurs des paramètres de la requête GET dans des backing beans
  - avant l'affichage de la page

Exemple d'utilisation

- Soit une requête GET http://…/employe.xhtml?id=12
- Un paramètre de vue de la page employe.xhtml récupère la valeur du paramètre id de la requête (12) dans la propriété `id` d'un backing bean
- L'action de `<f:viewAction>` récupère dans la base de données l'employé qui a l'id 12 et le met dans une propriété `employe` (type `Employe`) du backing bean
- Les informations sur l'employé sont ensuite affichées dans la page employe.xhtml, par exemple par `#{bean.employe.nom}`

GET http://…/employe.xhtml?id=12

Code de l'exemple

- Dans la page employe.xhtml :

```
<f:metadata>
  <f:viewParam name="id" value="#{bean.id}"/>
  <f:viewAction action="#{bean.findEmp()}"/>
</f:metadata>
...
#{bean.employe.nom}
```

- Dans le backing bean (`id` et `employe` sont des propriétés du bean) :

```
public void findEmp() {
  this.employe = empManager.findEmp(this.id);
}
```

Qui veut expliquer ?

Une autre façon de faire

- Utiliser un paramètre de vue, sans viewAction
- Un convertisseur dans `<f:viewParam>` transforme l'id en `Employe` (remplace l'utilisation de `<f:viewAction>`)
- Vous vous souvenez quand on a utilisé un convertisseur dans le TP 1 ?

Donner une valeur à un paramètre (1/2)

- Plusieurs façons de donner une valeur à un paramètre de requête GET :
  - `<h:link outcome="page?p1=4&p2='bibi'" … >`
    `<h:link outcome="page?p1=#{bean.prop + 2}" … >`
  - `<h:link outcome="page" …>`
    `  <f:param name="p1" value="4"/>`
    `</h:link>`

Donner une valeur à un paramètre (2/2)

- Dans le cas d'une requête POST avec redirection, écrire une méthode action dont la valeur de retour contient la valeur du paramètre :

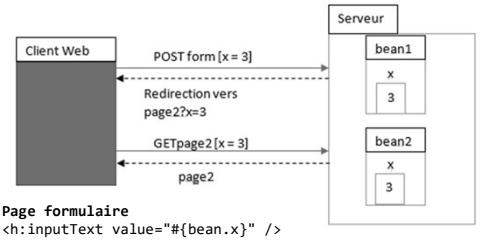
```
public String faire() {
  return "page2?nb=" + nombre
         + "&amp;faces-redirect=true";
}
```

Section « Messages d'erreur ou de succès »

PRG avec JSF sur un exemple

- L'utilisateur a saisi la valeur 3 dans un formulaire contenu dans `form.xhtml`
- On veut afficher cette valeur dans `page2.xhtml`, après redirection, avec un backing bean de portée requête
  (alors qu'il y a 2 requêtes à cause de la redirection)

Vue d'ensemble



Page formulaire

```
<h:inputText value="#{bean.x}" />
```

Code du backing bean

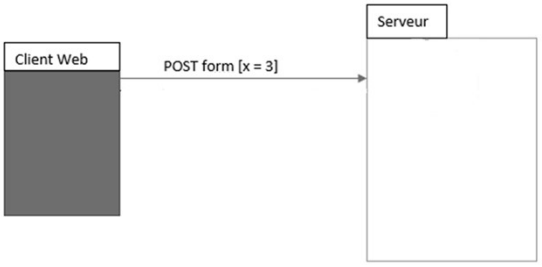
```
@Name @RequestScope
Bean {
 private x ; // et getter et setter pour x
 public String action() {
  ...
  return "page2?faces-redirect=true&x=" + x;
 }
}
```

Code page2.xhtml

```
<f:metadata>
  <f:viewParam
    name="x"  value="#{bean.x}"/>
</f:metadata>
...
#{bean.x}
```


109

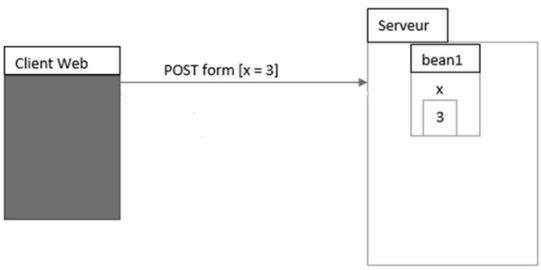
1ère requête



110

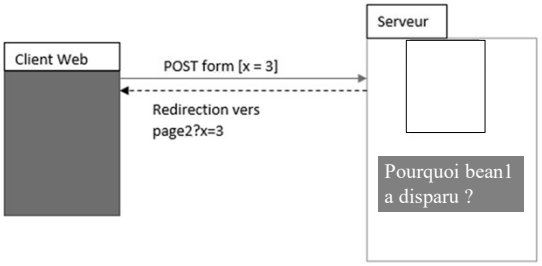
1ère requête

Pourquoi ?



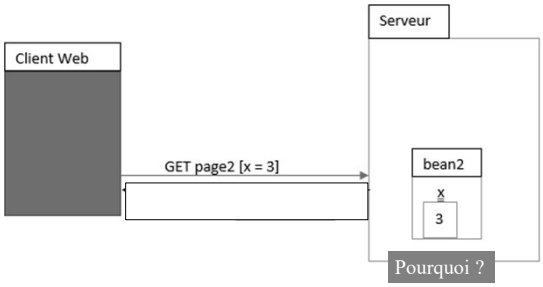
111

1ère requête



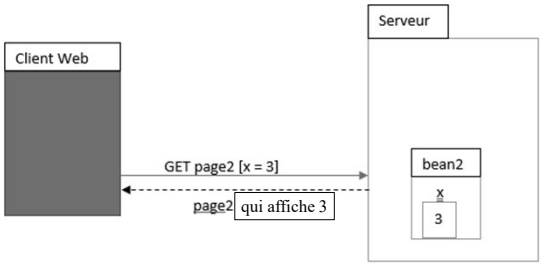
112

2ème requête (à cause redirection)



113

Réponse 2ème requête



114

Choix de la portée

- L'exemple suivant montre que le choix de la bonne portée est important
- Pour comprendre ce qui se passe, il faut avoir bien compris une grande partie des notions importantes déjà vues sur JSF

115

Exemple (1/3) – Le problème

- Dans le TP 1, le backing bean de la page qui affiche les détails sur un client est de portée View
- La portée Request est suffisante pour afficher les détails sur un client
- Mais elle ne suffit pas si on veut pouvoir modifier des informations sur le client affiché :
  Au moment de la soumission du formulaire, on a une erreur « `/CustomerDetails.xhtml @25,104 value="#{customerDetailsBean.customer.name}": Target Unreachable, 'null' returned null` »

116

Exemple (2/3) – Explication

- Dans la table, l'utilisateur clique sur un id, pour l'affichage des informations sur le client choisi
- Une requête GET est envoyée au serveur pour faire afficher la page des informations sur le client
- A l'arrivée sur le serveur de la requête GET,
  - une instance de `CustomerDetailsBean` est créée par CDI
  - et les informations sur le client sont mises dans la propriété `customer` de cette instance Comment ?
- Avec ces informations, la page des détails sur le client est retournée en réponse à la requête GET et l'instance du bean est supprimée Pourquoi ?

117

Exemple (3/3) – Explication

- L'utilisateur modifie des informations et soumet le formulaire ; à l'arrivée de la requête POST sur le serveur la page des détails est restaurée
- La requête POST est une **nouvelle** requête ; si la portée du backing bean est Request, à l'arrivée sur le serveur de la requête POST une nouvelle instance du backing bean est créée et cette instance a la valeur `null` pour sa propriété `customer`
- D'où le message d'erreur pour `#{customerDetailsMBean.customer.name}` quand JSF veut restaurer la page

118

Messages d'erreur ou d'information / de succès

119

Cas d'utilisation

- Les conversions et validations peuvent conduire à l'affichage de messages d'erreur
- Lorsqu'une action s'est bien déroulée il faut aussi le signaler à l'utilisateur par un message d'information / de succès

120

Affichage des messages

- `<h:messages>` indique un emplacement dans la page où sont affichés tous les messages
- `<h:message>` indique un emplacement pour afficher un message pour un seul composant identifié par l'attribut `for`
- Ces 2 balises ont de nombreux attributs

121

Exemples

```
<h:message for="montant"
           errorStyle="color: red"/>

<h:messages globalOnly="true"/>
```


122

Messages standards

- Les convertisseurs et validateurs standards produisent des messages d'erreur standards
- Les attributs des composants standards `requiredMessage`, `converterMessage` ou `validatorMessage` permettent de modifier ces messages
- Exemple :

```
<h:inputText id="nom"
             required="true"
             requiredMessage="Nom requis"/>
```


123

Messages personnalisés

- Les convertisseurs et validateurs personnalisés permettent d'afficher facilement des messages d'erreur
- S'ils ne peuvent pas être utilisés (par exemple, validation qui englobe plusieurs composants) ou pour afficher un message de succès, le développeur peut ajouter des messages dans les pages JSF par programmation Java
- Les transparents suivants étudient les classes et méthodes Java utilisées pour afficher ces messages

124

Classe `FacesMessage`

- La classe `FacesMessage` (`jakarta.faces.application`) représente un message affiché par `<h:messages>` ou `<h:message>`
- Propriétés d'un message :
  - sévérité (`SEVERITY_FATAL`, `SEVERITY_ERROR`, `SEVERITY_WARN`, `SEVERITY_INFO` de la classe interne `Severity`)
  - message détaillé
  - message résumé

125

Faire afficher son propre message

- Il suffit de passer une instance de `FacesMessage` à la méthode `addMessage` de `FacesContext` pour que le message soit affiché par les `<h:messages>` ou `<h:message>`
- `addMessage` a 2 paramètres :
  - id client du composant avec lequel le message est associé (de type `String` ; `null` si le message n'est pas attaché à un client particulier)
  - Le message (classe `FacesMessage`)

126

Exemple (pas lié à un composant)

```
String msgResume = "...";
String msgDetail = "...";
FacesMessage.Severity severite = FacesMessage.SEVERITY_ERROR;
FacesMessage fm =
    new FacesMessage(severite, msgResume, msgDetail);
FacesContext.getCurrentInstance().addMessage(null, fm);
```

Message pas lié à un composant

Une instance de `FacesContext` est associée au traitement de la requête en cours

127

Exemple (lié à un composant)

```
context.addMessage(
    "insertion:login",
    new FacesMessage(FacesMessage.SEVERITY_ERROR,
                     "Le message résumé",
                     "Le message détaillé"));
```

id client du composant

128

Message après redirection

- Par défaut un `FacesMessage` ne dure qu'une seule requête, il ne s'affiche pas après une redirection
- Si on veut qu'il s'affiche après une redirection, il faut un code un peu plus complexe qui utilise la classe `Flash`

```
public void addFlashMessage(FacesMessage message) {
  FacesContext facesContext =
      FacesContext.getCurrentInstance();
  Flash flash =
      facesContext.getExternalContext().getFlash();
  flash.setKeepMessages(true);
  facesContext.addMessage(null, message);
}
```


129

Message de validation ou conversion

- Quand il y a une erreur dans un convertisseur ou un validateur, le code doit lancer une exception (`ConverterException` ou `ValidatorException`) ; c'est le message passé au constructeur de l'exception qui est affiché
- Pas besoin de préciser le composant associé au message puisque c'est celui qui est validé

130

Exemple de message dans validateur

```
public void validateId(FacesContext context,
    UIComponent composant, Object id) {
  Compte compte = compteManager.getCompte((Long) id);
  if (compte == null) {
    FacesMessage message =
      new FacesMessage("Pas de compte avec id " + id);
    throw new ValidatorException(message);
  }
}
```


131

- Finir TP 2
- TP 3 (templates)

132

Id client d'un composant

- Si on veut associer un message à un composant et qu'on n'est pas dans un convertisseur ou un validateur il faut trouver l'id client du 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` »

Identificateur « client »

- Pas toujours facile de trouver les id « client »
- Regarder le code HTML généré aide
- Identificateur client composé de plusieurs niveaux, avec le séparateur « : » ; par exemple, « formulaire:j_idt41 »
- Pour faciliter, donner des ids aux composants (y compris composants englobants) et/ou utiliser l'attribut `prependId="false"` pour les formulaires qui les contiennent
- Identificateur doit exister dans la page HTML générée (attention aux `rendered="false"`)

Messages internationalisés

- L'internationalisation d'une application n'est pas difficile mais pas étudiée dans ce cours d'introduction
- Juste une petite difficulté pour faire afficher des messages créés par programmation, qui dépendent de la langue il faut récupérer le `ResourceBundle` qui contient les textes en différentes langues :

```
ResourceBundle textes =
  ResourceBundle.getBundle(CHEMIN_BUNDLE);
...
texte = textes.getString(nomPropTexte);
```


Événements

Types d'événements

- 2 grands types d'événements :
  - Générés par une action de l'utilisateur (de type « *application* »)
  - Générés à un moment du cycle de vie (de type « *lifecycle* »)

Événements « application »

- ValueChangeEvent : valeur d'un composant modifiée par l'utilisateur (champ de saisie de texte, menu, liste déroulante,…) ; liés aux composants tels que `<h:selectOneMenu>` ou `<h:inputText>`
- ActionEvent : clic sur un bouton ou un lien ; liés aux composants tels que `<h:commandButton>` ou `<h:commandLink>`

Écouteur

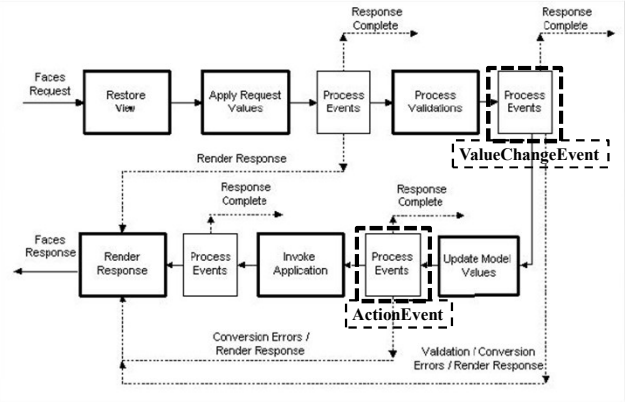
- Les événements sont traités par des écouteurs (*listeners*) durant le cycle de vie JSF par,
  - des méthodes (attribut xxxListener)
  - ou des classes (sous-balise <f:xxxListener>)
- Remarque : les actionListeners sont exécutés avant les méthodes « action », durant la phase « Invoke Application » du cycle de vie

139

`Action` ou `ActionListener` ?

- `Action` pour les traitements métier
- `ActionListener` pour les événements liés à l'interface utilisateur
- `Action` n'a pas accès aux composants de l'interface utilisateur alors que `ActionListener` y a accès (par le paramètre `ActionEvent` de la méthode qui est lancée)

140



141

Exemples

- Classe écouteur :

```
<h:selectOneMenu value="#{bean.codePays}"
                 onchange="submit()">
  <f:valueChangeListener
     type="fr.unice.jsf.PaysListener" />
  <f:selectItems value="#{bean.listePays}" />
</h:selectOneMenu>
```

- Méthode écouteur :

```
<h:selectOneMenu value="#{bean.codePays}"
                 onchange="submit()"
    valueChangeListener="#{bean.modifPays}" />
  <f:selectItems value="#{bean.listePays}" />
</h:selectOneMenu>
```


142

Méthode valueChangeListener

- `ValueChangeEvent` en paramètre et `void` en retour
- Exemple :

```
public void modifPays(ValueChangeEvent evt) {
  for (Locale locale : pays) {
    if (locale.getCountry().equals(evt.getNewValue())) {
      FacesContext.getCurrentInstance()
        .getViewRoot().setLocale(locale);
  }
}
```

Que fait cette méthode ?

143

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'événements de ce type) : provoqué par divers « sous-phases » du cycle de vie ; par exemple, `PreRenderComponentEvent` ou `PostValidateEvent`

144

Ajax

145

Ajax

- *Asynchronous Javascript and XML*
- Une requête Ajax envoyée par le client est lancée « en arrière-plan » ; l'utilisateur peut continuer à travailler avec la page en cours
- La réponse du serveur à la requête ne modifie qu'une partie de la page en cours

146

`<f:ajax>`

- Ajoute des fonctionnalités « Ajax » au composant JSF dans lequel il est inclus
- Un événement déclencheur sur ce composant provoque l'envoi d'une requête Ajax au serveur sans attendre la soumission du formulaire
- Par exemple, une liste déroulante peut envoyer une requête Ajax quand l'utilisateur fait un nouveau choix ; la réponse du serveur ne mettra à jour qu'une partie de la page, par exemple une 2ème liste déroulante dont les valeurs dépendent du choix fait par l'utilisateur dans la 1ère liste

147

`<f:ajax>` - les attributs

- `event` : événement déclencheur de la requête (doit être un événement compatible avec le composant) ; valeur par défaut dépend du composant, par exemple, `valueChange` pour les `<h:inputText>` ou `action` pour les `<h:commandButton>`
- `execute` : composants pris en compte pour l'envoi de la requête (espace séparateur) ; valeur par défaut `@this`, le composant qui contient `<f:ajax>`
- `render` : composants mis à jour au moment de la réponse du serveur (espace séparateur) ; valeur par défaut `@none`

148

Exemple 1

```
<h:form prependId="false">
  <h:inputText value=... />
  <h:commandButton value="Enregistrer" ...>
    <f:ajax execute="@form"
            render="out1 out2" />
  </h:commandButton>
  <h:outputText id="out1" .../>
  <h:outputText id="out2" .../>
</h:form>
```

Formulaire englobant

149

Exemple 2

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


150

Identificateurs « client »

- Utilisés dans les attributs `execute` et surtout `render`
- Revoir section « Messages d'erreur ou d'information »

151

Mélange d'Ajax et de requête normale

- Quand un champ de saisie lance une requête Ajax, et que l'utilisateur appuie sur la touche [Entrée] après avoir saisi la valeur, un message d'avertissement indique qu'une requête Ajax a été mélangée avec une requête « normale » (l'appui sur la touche [Entrée] déclenche la soumission du formulaire en même temps que l'action « Ajax »)
- Pour ne plus avoir cet avertissement, il suffit de passer le bouton de soumission en Ajax :

```
<h:commandButton …><f:ajax execute="@form"
render="@form"/></h:commandButton>
```


152

Listener

- L'attribut `listener` de `<f:ajax>` désigne une méthode qui est appelée quand l'événement déclencheur de l'appel Ajax survient :

```
<f:ajax event="keyup" render="table"
        listener="#{bb.m}"
```

- Le plus souvent la méthode modifie le backing bean utilisé par la page, ce qui modifie la page

153

Exemple – valeur composant

```
public void m(AjaxBehaviorEvent event) {
  UIInput input = (UIInput) event.getComponent();
  UnType valeur = (UnType) input.getValue();
  ...
}
```

La méthode « listener » doit avoir un paramètre de type `AjaxBehaviorEvent`

154

Exemple d'utilisation

- Un écouteur peut récupérer
  - un client à partir d'un id saisi par l'utilisateur
  - tous les noms qui commencent par les caractères tapés par l'utilisateur sont affichés comme suggestion (complétion de nom)

155

JSF et HTML5

156

Pages HTML5

- Les pages HTML générées par JSF peuvent être déclarées comme des documents HTML5 avec `<!DOCTYPE html>`

157

Composants HTML5

- Avant JSF 2.2, pour profiter d'un nouveau composant HTML dans JSF il fallait écrire un composant JSF
- JSF 2.2 permet de profiter des services offerts par JSF (validation de données, affichage automatique des messages d'erreur, utilisation simple d'Ajax, …) tout en utilisant un attribut ou un composant HTML5

158

2 possibilités

- On peut utiliser un attribut HTML5 (attribut « *pass-through* ») dans un composant JSF
- Exemple : attribut `placeholder` de `<input>` alors qu'il n'est pas un attribut de `<h:inputText>`
- L'attribut sera passé tel quel à la balise HTML générée à la fin du cycle de vie JSF
- On peut aussi utiliser une balise HTML5, tout en profitant des facilités de JSF avec des attributs traités par JSF

159

Attributs HTML5 en JSF

- Etudions d'abord le cas d'un attribut HTML5 dans un composant JSF

160

Passer un attribut HTML5

Remarque : donner un autre alias que « p » si PrimeFaces est utilisé

1. Avec l'espace de noms « *passthrough* » :

```
<html ... xmlns:pt="jakarta.faces.passthrough">
  <h:form>
    <h:inputText value="#{bean.nom}"
        pt:placeholder="Votre nom"/>
```

2. Avec une balise `<f:passThroughAttribute>` :

```
<h:inputText value="#{bean.nom}" >
  <f:passThroughAttribute
    name="placeholder" value="Votre nom" />
</h:inputText>
```


161

Balise HTML5 en JSF

- Etudions maintenant comment insérer une balise HTML5 dans une page JSF, tout en profitant des avantages de JSF (expressions EL comme valeurs, conversions, validations, messages d'erreur,…)

162

Composants HTML5 *pass-through*

- Une page JSF peut contenir des balises HTML5
- Pour qu'un élément HTML5 soit traité par JSF il suffit qu'un de ses attributs ait l'espace de noms `jakarta.faces` :

```
<html xmlns:faces="jakarta.faces">
...
<input type="text" id="nom"
      placeholder="Votre nom"
      faces:value="#{bean.name}"/>
```


163

Exemple

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:faces="jakarta.faces">
<form jsf:id="form">
  <input name="nom" type="text"
         faces:id="nom" value="#{bean.nom}"/>
  <input type="submit" value="Soumettre" />
  <p>Texte saisi : #{bean.nom}</p>
</form>
</html>
```

la valeur de `value` sera associée à la propriété `nom` du backing bean

164

Composant HTML5 traité par JSF

- Le composant HTML5 est transformé en un composant JSF suivant un mapping qui tient compte du nom de la balise et éventuellement de certains attributs de la balise
- Exemple : `<input type="file">` est transformé en `<h:inputFile>`

165

Composant HTML5 sans équivalent JSF

- Si le composant HTML5 n'a pas de correspondance dans JSF, JSF crée un composant spécial de balise `<jsf:element>`
- On peut ajouter des capacités Ajax à un tel composant (voir exemple à suivre), mais seulement quelques attributs JSF liés aux événements JavaScript
- Les valeurs des attributs HTML5 peuvent contenir des expressions EL comme pour les composants JSF

166

Exemple

- Ajouter un comportement Ajax à la balise `div` :

```
<div jsf:id="compteurClics">
  Clics: #{bean.compterClicsSouris}
  <f:ajax event="click" render="@this"
          listener="#{bean.incrementer}"/>
</div>
```


167

Exemple avec `<input type="date">`

```
<h:form>
   <h:outputLabel value="Date : " for="date" />
   <input type="date" jsf:id="date" name="date"
          value="#{reservationBean.date}">
      <f:convertDateTime pattern="yyyy-MM-dd"/>
   </input>
   <br/>
   <h:commandButton value="Enregistrer la date"
                    action="resultat"/>
</h:form>
```

Le type de **`date`** est `java.util.Date`

168

Ressources

169

Ressources des pages Web

- Les pages HTML font le plus souvent appel à des ressources externes : images, fichiers CSS, code Javascript,…
- Il faut utiliser des balises JSF dédiées aux ressources à la place des balises HTML ; en ce cas JSF va chercher les ressources dans le répertoire `resources` placé sous la racine des pages Web

170

CSS

- Le plus souvent on met les fichiers CSS dans un sous-répertoire `css` du répertoire `resources`

```
<h:outputStylesheet name="css/style.css"/>
```


171

JavaScript

- Le plus souvent on met les fichiers JavaScript dans un sous-répertoire `js` du répertoire `resources`

```
<h:outputScript name="js/jsf.js" />
```

- Par défaut, le script sera placé dans la page HTML à l'endroit où on a mis la balise ; on peut ajouter un attribut `target` pour indiquer où le mettre (par exemple `target="head"` pour le mettre à la fin de la section head de la page)

172

Image

```
<h:graphicImage name="images/photo.jpg" />
```


173

Thèmes

- On peut avoir des thèmes qui réunissent les ressources en utilisant l'attribut `library` des balises que l'on vient de voir
- En ce cas, chaque librairie correspondra alors à un thème différent, par exemple on aura un thème « defaut » et un thème « dark »
- Sous chacun des répertoires defaut et dark (placés sous le répertoire `resources`), on retrouvera les répertoire `css`, `js` et `image` qu'on vient de voir
- Il suffit alors de changer de librairie pour changer à la fois le CSS, le JavaScript et les images

174

Flots (*faces flow*)

175

Présentation

- A la manière des « *wizards* », les flots de pages (en fait, flots de nœuds, comme on le verra plus loin) sont des ensembles de pages qui partagent des informations communes (entreposées dans un backing bean de portée Flow)

176

Exemple

- Les pages qui permettent à un utilisateur de créer un compte avec ses informations personnelles, l'adresse de livraison, l'adresse de facturation et les informations sur sa carte de crédit
- Chaque type d'information est donné sur une page différente
- Souvent la dernière page demande confirmation de toutes les données saisies et on peut lancer leur enregistrement ou un traitement avec les informations recueillies sur toutes les pages

177

Utilité

- L'utilisation d'un enchaînement de pages pour obtenir de l'utilisateur un ensemble d'informations est fréquent
- Avant JSF 2.2, un backing bean de portée conversation était le plus souvent utilisé, avec les difficultés occasionnées par cette portée
- Un flot de pages
  - simplifie la programmation d'une telle situation
  - facilite la modularité
  - peut être réutilisé dans des applications différentes

178

Similaire à une méthode Java

- Un seul point d'entrée, plusieurs points de sortie possibles
- Paramètres d'entrée et valeur de retour
- Peut être appelé d'un autre point de l'application
- Interface bien définie mais les détails de l'implémentation sont cachés
- Un flot peut en appeler un autre et les portées s'empilent et se dépilent comme dans une pile d'exécution de méthodes

179

Navigation pour entrer et sortir du flot

- On ne peut entrer dans le flot que par la page de démarrage (utiliser une requête GET pour aller vers cette page) ; les autres pages du flot ne sont pas atteignables de l'extérieur du flot
- Quand on est dans le flot, on ne peut aller que vers des pages du flot ou vers une page de sortie
- Pour sortir du flot il suffit de naviguer depuis une page du flot vers une page de sortie ou vers une page qui n'est pas dans le flot (il faut y aller avec une requête POST, éventuellement avec une redirection ; une requête GET ne marche pas)

180

Suppression des backing beans

- Dès que la navigation amène à la page de sortie, les backing beans de portée flow sont supprimés automatiquement par CDI

181

Navigation par requête POST

- La navigation ne marche pas toujours bien entre pages du flot si on utilise des requêtes GET
- Il est donc conseillé de n'utiliser que des requêtes POST, avec éventuellement des redirections pour avoir des URL affichés qui correspondent aux pages affichées

182

Plusieurs flots

- Les flots peuvent être enchaînés séquentiellement ou bien emboités les uns dans les autres

183

Portée

- Comme la portée « conversation » la portée « flot » (annotation des backing beans par `@FlowScoped`) est entre la portée requête et la portée session
- La portée flot est une portée CDI qui commence quand l'utilisateur navigue dans la page d'entrée du flot et se termine quand l'utilisateur quitte le flot en navigant vers une page de sortie du flot
- Au contraire de la portée session, la portée flot n'a pas de problème si l'utilisateur ouvre plusieurs onglets du navigateur sur la même application

184

Définition d'un flot

- Suit le principe « Convention plutôt que configuration » : on peut définir un flot en créant un répertoire dans la racine des pages JSF de l'application
- Si on n'utilise pas la « convention », le flot peut être configuré dans le fichier `<nom-flot>-flow.xml` placé dans le répertoire qui contient les pages du flot, dans le fichier `WEB-INF/faces-config.xml` ou bien par programmation avec un constructeur de Flow CDI (pas étudié dans ce support)

185

Définition d'un flot par convention

- Répertoire dans la racine des pages, du même nom (identificateur du flot) que le flot
- Dans le répertoire,
  - Une page du même nom que le répertoire (et suffixe .xhtml) comme nœud de démarrage du flot
  - Les autres pages qui sont dans le flot
  - Un fichier de configuration obligatoire nommé <nom-flot>-flow.xhtml (vide si on n'a rien à configurer)
- En dehors du répertoire une seule page de sortie de nom `<nom-flot>-return.xhtml`

186

id d'un flot

- Si le flot a été défini en suivant les conventions (transparent précédent), l'id du flot est le nom du répertoire qui contient la définition et les pages du flot
- Sinon, l'id du flot est donné par l'attribut `id` de la balise `<flow-definition>` dans la définition du flot

187

Code pour entrer et sortir

- Les exemples suivants supposent que la convention a été suivie (pas de configuration)
- Exemple de code pour entrer dans le flot nommé flot1 (« flot1 » est l'id du flot) :

```
<h:button id="start1"
  value="Entrer dans Flot1"
  outcome="flot1"/>
```

- Exemple de code pour sortir du flot nommé :

```
<h:commandButton id="home" value="home"
  action="/flot1-return" />
```

Fichier flot1-return.xhtml situé en dehors du répertoire flot1

188

Code du backing bean

```
@Named
@FlowScoped("flot1")
public class Flot1Bean implements Serializable {
  private String val;

  public String getVal() {
    return val;
  }

  public void setVal(String val) {
    this.val = val;
  }
}
```

Propriété utilisée par le flot

189

Configuration avec fichier XML (1/2)

- Si la convention ne convient pas, on peut définir la configuration dans un fichier de configuration non vide (fichier placé dans le répertoire du flot ou fichier global `WEB-INF/faces-config.xml`)
- Il doit alors contenir au moins la définition de l'id du flot et une déclaration de page de sortie du flot (il peut y avoir plusieurs balises `<flow-return>` pour plusieurs pages de sorties) :

```
<flow-return id="first-return-id">
  <from-outcome>/retour</from-outcome>
</flow-return>
```


190

Configuration avec fichier XML (2/2)

- Le fichier peut aussi configurer
  - le nœud de démarrage
  - d'autres nœuds du flot
  - des paramètres à passer au flot (`<inbound-parameter>`)
  - un initialiseur, méthode exécutée avant l'entrée dans le flot
  - un « finalizer », méthode appelée juste avant la sortie du flot

191

Exemple de fichier XML minimal

```
<faces-config version="4.0"
    xmlns="https://jakarta.ee/xml/ns/jakartaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
https://jakarta.ee/xml/ns/jakartaee/web-facesconfig_4_0.xsd">
  <flow-definition id="flot1">
    <flow-return id="return1">
      <from-outcome>#{bean.fin()}</from-outcome>
    </flow-return>
  </flow-definition>
</faces-config>
```

En valeur retour, l'id de la vue ou bien, si la vue n'est pas définie, le chemin par rapport à la racine ; par exemple /index (sans le suffixe)

192

Types de nœuds

- Les types de nœud sont décrits dans la javadoc de la classe `jakarta.faces.flow.FlowHandler`) :
  - View, page JSF « normale »
  - Switch, liste d'expressions EL de type boolean ; la 1ère expression qui est évaluée à true détermine le prochain nœud
  - Return, arrête le flot en cours et définit la page affichée à la sortie du flot (page qui n'est pas dans le flot)
  - Method Call, appel d'une méthode (avec paramètres) dont la valeur de retour détermine le prochain nœud
  - Faces Flow Call, appel d'un autre flot ; au retour de cet autre flot, le flot reprend la main (pile de flots)

193

En savoir plus sur les flots…

- Les informations détaillées sur ces nœuds et la configuration d'un flot par programmation ne seront pas étudiées dans cette section d'introduction aux flots
- Un exemple du tutoriel Jakarta EE : https://eclipse-ee4j.github.io/jakartaee-tutorial/#the-checkout-module-example-application

194

Bibliographie

- The Definitive Guide to Jakarta Faces in Jakarta EE 10. Bauke Scholtz et Arjan Tijns, Apress. Le plus complet, et à jour sur JSF 4.0
- Core JSF, 3ème édition mise à jour pour JSF 2.0. David Geary et Cay Horstmann. Prentice Hall. Le meilleur livre pour débuter.

195