

FORMATION JAVA FRAMEWORK

JSF

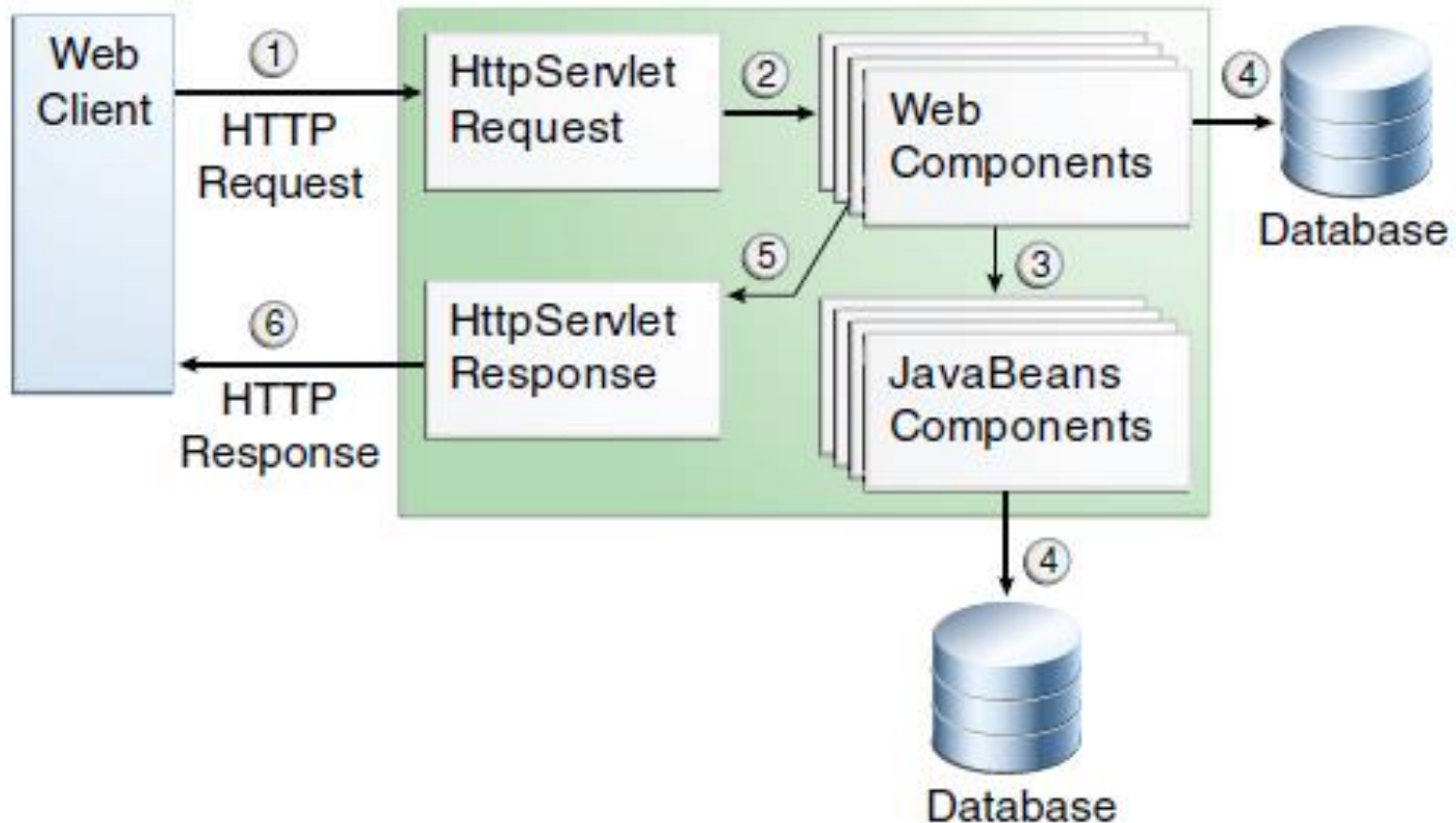
Objectifs

- Conception d'applications web en utilisant une architecture standard, protocoles, et composants
- Configurer JSF dans le conteneur Web
- Conception de vue avec JSF et EL
- Conception de composants avec les facelets
- Comprendre le cycle de vie d'une requête JSF2
- Utilisation des composants JSF2
- Utilisation de l'Ajex avec JSF2
- Implémentation et utilisation des composites component
- Utilisation des différentes librairies tierces (Richefaces, Primefaces)
- Tester une Application JSF 2

Présentation

- Java Server Faces (JSF) a changé la façon dont on écrit les applications Web Java. Conçu pour simplifier la création d'interfaces utilisateur (UI) des applications web Java performantes, JSF simplifie également le processus de développement. Il offre une solution élégante aux problèmes clés souvent associés au développement d'applications Web de qualité.

Architecture typique d'une application web



Utilité de JSF

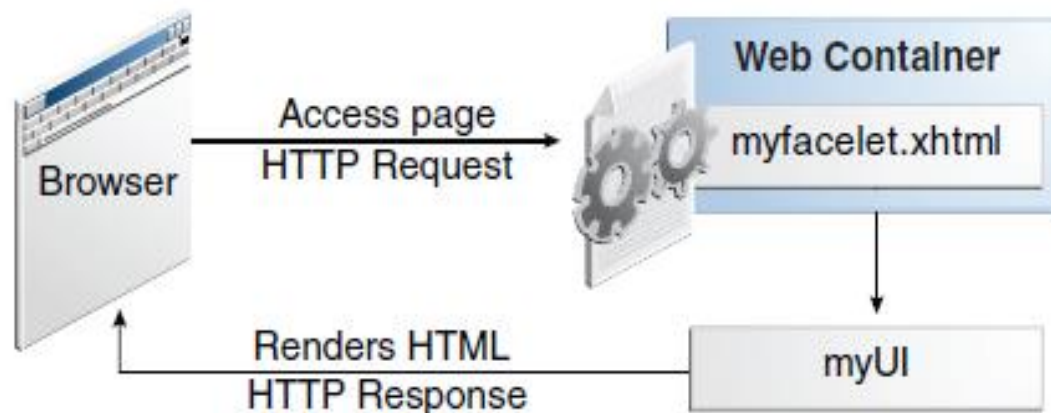
- Créer des pages Web dynamiques
- Par exemple, une page Web qui est construite à partir de données enregistrées dans une base de données
- Une grande partie de la programmation liée à la validation des données saisies par l'utilisateur et de leur passage au code de l'application est automatisée ou grandement facilitée
- Ajax sans programmation

JSF

- **Framework MVC** qui reprend le modèle des interfaces utilisateurs locales (comme Swing),
- **Le modèle** est représenté par des classes Java, les backing beans, sur le serveur,
- **Les vues** sont représentées par des composants Java sur le serveur, transformées en pages HTML sur le client,
- **Le contrôleur** est une servlet qui intercepte les requêtes HTTP liées aux applications JSF et qui organise les traitements JSF

Une page JSF

- Code XHTML,
- Contient des parties en EL : `#{...}`,
- Utilise souvent une (ou plusieurs) bibliothèque de composants,
- Sera traduite en XHTML « pur » pour être envoyée au client Web,



Répartition des tâches

- Les pages JSF ne contiennent pas de traitements (pas de code Java ou autre code comme dans les pages JSP, « ancien langage » de JSF pour créer les pages affichées à l'utilisateur,
- Les traitements liés directement à l'interface utilisateur sont écrits dans les backing beans,
- Ces backing beans font appels à des EJB ou des classes Java ordinaire pour effectuer les traitements qui ne sont pas liés directement à l'interface utilisateur

Répartition des tâches (2)

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

Beans

- 2 types principaux de beans :
 - EJB (essentiellement des beans sessions) liés aux processus métier,
 - « Backing beans » associés aux pages JSF et donc à l'interface utilisateur qui font la liaison entre les composants JSF de l'interface, les valeurs affichées ou saisies dans les pages JSF et le code Java qui effectue les traitements métier.

Backing bean

- Souvent, mais pas obligatoirement, un backing bean par page JSF,
- Conserve, par exemple, les valeurs saisies par l'utilisateur (attribut value), ou une expression qui indique si un composant ou un groupe de composant doit être affiché ; peut aussi conserver un lien vers un composant de l'interface utilisateur (attribut binding), ce qui permet de manipuler ce composant ou les attributs de ce composant par programmation.

Container pour les JSF

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


Composants JSF sur le serveur

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

Cycle de vie JSF 2

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

Le servlet « Faces »

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

# URL des pages JSF

- Les pages JSF sont traitées par le servlet parce que le fichier web.xml contient une configuration telle que celle-ci :

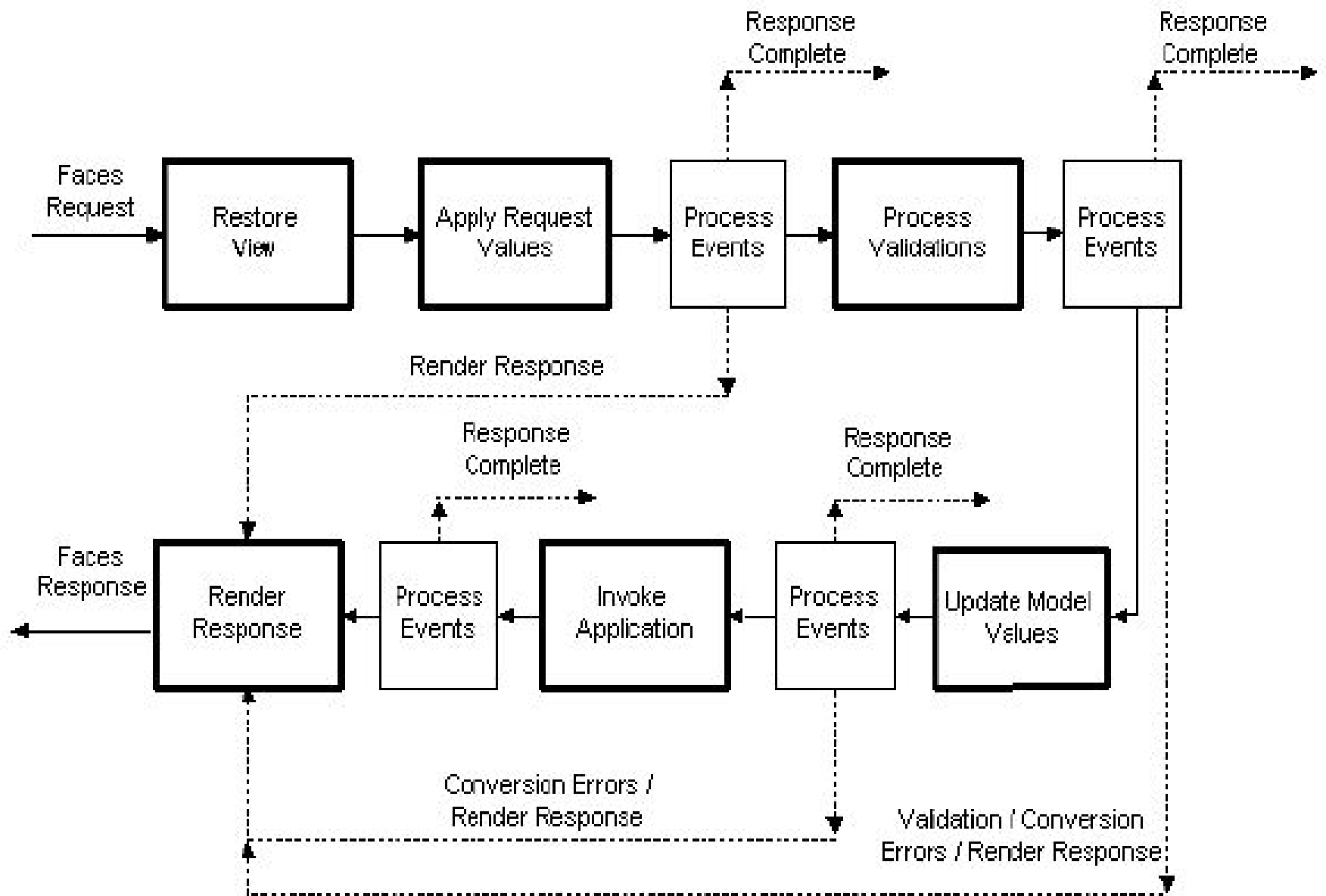
- `<servlet-mapping>`

```
 <servlet-name>Faces Servlet</servlet-name>
 <url-pattern>/faces/*</url-pattern>
</servlet-mapping>
```

Le pattern est souvent aussi de la forme

`*.faces` (les pages dont l'URL se termine par `.faces` sont considérées comme des pages JSF)  
ou `*.jsf`





# Sauter des phases

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

# immediate=true sur un UICommand

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

# immediate=true sur un EditableValueHolder

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

# Navigation statique et dynamique

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

# Dans les pages JSF

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

- Avec une valeur :

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

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

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

# Dans le fichier de configuration faces-config.xml

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

Valeur retournée

# Compléments

- S'il n'y a pas d'élément `<from-view-id>`, la règle de navigation s'applique à toutes les pages
- On peut mettre un joker à la fin de la valeur de `<from-view-id>` :  
`<from-view-id>/truc/*</from-view-id>`
- On peut ajouter `<redirect/>` à la fin de la règle de navigation pour que l'URL de la page soit modifié (redirection à la place de *forwarding*) ; intéressant si l'utilisateur peut avoir à enregistrer un signet (*bookmark*)



# Paramètres de vue

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

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

# <f:viewParam>

- L'attribut name désigne le nom d'un paramètre HTTP de requête GET,
- L'attribut value désigne (par une expression du langage EL) le nom d'une propriété d'un bean dans laquelle la valeur du paramètre est rangée,
- Important : il est possible d'indiquer une conversion ou une validation à faire sur les paramètres, comme sur les valeurs des composants saisis par l'utilisateur.

## <f:viewParam> (suite)

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

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

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

# Managed Beans

Classes pour représenter les données de formulaires

# Bean basique et bean managé

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

# Bean basique et bean managé

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

# Bean basique et bean managé

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

# Exemples de propriétés

Nom de méthode	Nom de propriété	Utilisation dans une page JSF
getFirstName setFirstName	firstName	<code>{customer.firstName}</code> <code>&lt;h:inputText value="{customer.firstName}"/&gt;</code>
isValid setValid (booléen)	valid	<code>{login.valid}</code> <code>&lt;h:selectBooleanCheckbox</code> <code>value="{customer.executive}"/&gt;</code>
getValid setValid (booléen)	valid	<code>{login.valid}</code> <code>&lt;h:selectBooleanCheckbox</code> <code>value="{customer.executive}"/&gt;</code>
getURL setURL	URL	<code>{address.ZIP}</code> <code>&lt;h:inputText value="{address.ZIP}"/&gt;</code>



# Pourquoi ne pas utiliser d'attributs publics

- 1) Règle d'or des beans

- Mauvais

```
public double vitesse;
```

- Bon

```
private double v;
```

```
public double getVitesse() {
 return v;
}
public void setVitesse(double v) {
 this.v = v;
}
```

- Utilisez les wizards de vos IDEs pour générer du code pour des propriétés (netbeans : clic droit/insert code)

# Pourquoi ne pas utiliser d'attributs publics

## ■ 2) On peut mettre des contraintes

```
public void setVitesse(double v) {
 if(v > 0) {
 this.v = v;
 } else {
 envoyerMessageErreur(); // ou exception...
 }
}
```

## ■ 3) On peut faire plus : ex notifier un changement de valeur

```
public void setVitesse(double v) {
 if(v > 0) {
 this.v = v;
 updateCompteurSurTableauDeBord(v);
 } else {
 envoyerMessageErreur();
 }
}
```

# Pourquoi ne pas utiliser d'attributs publics

- 4) on peut changer la représentation interne de la variable sans changer son interface d'utilisation
  - Par ex : stocker la vitesse int alors qu'on a des getters/setters qui fonctionnent en double...

# Les trois composantes des beans managés

- Des propriétés (donc définies par des get/set ou is...)
  - Une paire pour chaque élément input de formulaire,
  - Les setters sont automatiquement appelés par JSF lorsque le formulaire sera soumis. Appelé avant les « méthodes de contrôle »;
- Des méthodes « action controller »
  - Une par bouton de soumission dans le formulaires (un formulaire peut avoir plusieurs boutons de soumission),
  - La méthode sera appelée lors du clic sur le bouton par JSF

# Les trois composantes des beans managés

- Des propriétés pour les données résultat
  - Seront initialisées par les méthodes « action controller » après un traitement métier,
  - Il faut au moins une méthode get sur la propriété afin que les données puissent être affichées dans une page de résultat.

# Les « scopes »

- Ils indiquent la durée de vie des managed beans,
- Valeurs possibles : request, session, application, view, conversation, aucun ou custom
- RequestScope = valeur par défaut.
- On les spécifie dans faces-config.xml ou sous forme d'annotations de code (recommandé)

# Annotations pour les Scopes

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

# Annotations pour les Scopes

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



# Annotations pour les Scopes

- `@ApplicationScoped`
  - Met le bean dans « l'application », l'instance sera partagée par tous les utilisateurs de toutes les sessions. Le bean est en général stateless (sans attributs d'état).
  - Pour des méthodes utilitaires uniquement.

# Annotations pour les Scopes

- @ViewScoped
  - La même instance est utilisée aussi souvent que le même utilisateur reste sur la même page, même s'il fait un refresh (reload) de la page !
  - Le bean doit être sérializable,
  - A été conçu spécialement pour les pages JSF faisant des appels Ajax (une requête ajax = une requête HTTP -> une instance si on est en RequestScoped !)
  - On utilise souvent des event handlers dans les pages JSF avec ce type de bean (tp1)

# Annotations pour les Scopes

- `@ConversationScoped`
  - Utilise CDI, ne fait pas partie de JSF, `@Named` obligatoire (pas `@ManagedBean`)
  - Semblable aux session mais durée de vie gérée « par programme »,
  - Utile pour faire des wizards ou des formulaires remplis partiellement au travers de plusieurs pages;
  - On va confier à une variable injectée le démarrage et la fin de la durée de vie du bean,
- `@CustomScoped(value=« #{uneMap} »)`
  - Le bean est placé dans la HashMap et le développeur gère son cycle de vie.

# Annotations pour les Scopes

- `@NoneScoped`
  - Le bean est instancié mais pas placé dans un Scope.
  - Utile pour des beans qui sont utilisés par d'autres beans qui sont eux dans un autre Scope.

# Ou placer ces annotations

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

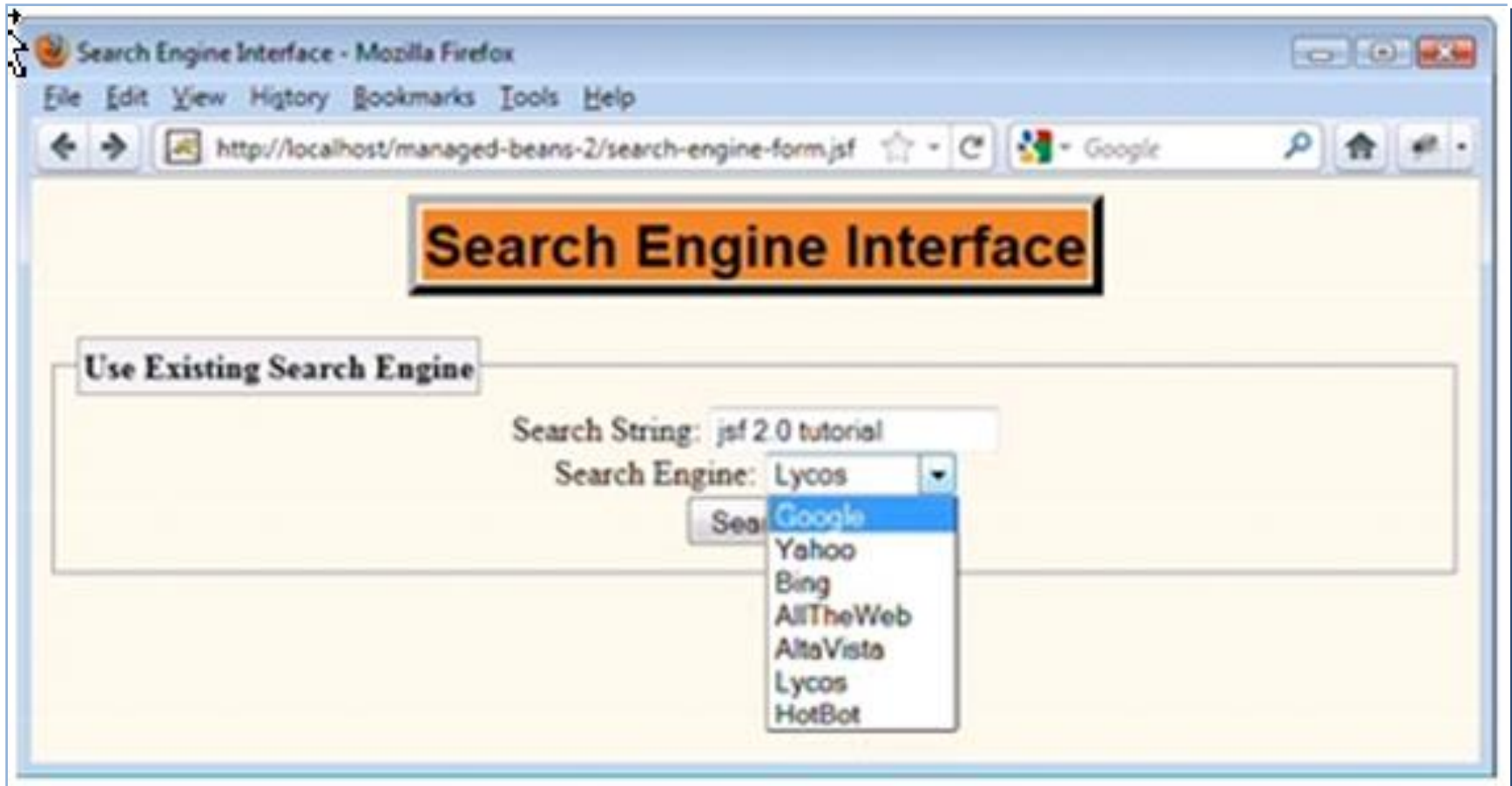
# Exemple ApplicationScoped

```
package coreservlets;

import javax.faces.bean.*;

@ManagedBean
@ApplicationScoped
public class Navigator {
 public String choosePage() {
 String[] results =
 { "page1", "page2", "page3" };
 return(RandomUtils.randomElement(results));
 }
}
```

# Exemple : choix d'un moteur de recherche



# Exemple : choix d'un moteur de recherche

```
<!DOCTYPE ...>
<html xmlns="http://www.w3.org/1999/xhtml"
 xmlns:f="http://java.sun.com/jsf/core"
 xmlns:h="http://java.sun.com/jsf/html">
<h:head>...</h:head>
<h:body>
...
<h:form>
 Search String:
 <h:inputText value="#{searchController.searchString}"/>

 Search Engine:
 <h:selectOneMenu value="#{searchController.searchEngine}">
 <f:selectItems value="#{searchController.searchEngineNames}"/>
 </h:selectOneMenu>

 <h:commandButton value="Search"
 action="#{searchController.doSearch}"/>
</h:form>
</h:body></html>
```



# Exemple : choix d'un moteur de

```
@ManagedBean
public class SearchController {
 private String searchString="", searchEngine;

 public String getSearchString() {
 return(searchString);
 }
 public void setSearchString(String searchString) {
 this.searchString = searchString.trim();
 }
 public String getSearchEngine() {
 return(searchEngine);
 }
 public void setSearchEngine(String searchEngine) {
 this.searchEngine = searchEngine;
 }
 public List<SelectItem> getSearchEngineNames() {
 return(SearchUtilities.searchEngineNames());
 }
}
```

# Exemple : choix d'un moteur de

```
public String doSearch() throws IOException {
 if (searchString.isEmpty()) {
 return("no-search-string");
 }
 searchString = URLEncoder.encode(searchString, "utf-8");
 String searchURL =
 SearchUtilities.makeURL(searchEngine, searchString);
 if (searchURL != null) {
 ExternalContext context =
 FacesContext.getCurrentInstance().getExternalContext();
 HttpServletResponse response =
 (HttpServletResponse)context.getResponse();
 response.sendRedirect(searchURL);
 return(null);
 } else {
 return("unknown-search-engine");
 }
}
```

## Et AJAX ?

- Les implémentations JSF2 supportent nativement AJAX.
- Les librairies supplémentaires proposent des compléments :

- MyFaces



- ICEfaces



- JBoss Richfaces

