JSF

1-Introduction

-Les pages statiques sont composées de HTML pur contenant éventuellement des graphiques eux aussi statiques (JPG, PNG, par exemple).

- Les pages dynamiques sont en revanche composées en temps réel (à la volée) à partir de données calculées à partir d'informations fournies par l'utilisateur

-Pour créer un contenu dynamique, il faut analyser les requêtes HTTP, comprendre leur signification et créer des réponses dans un format que le navigateur saura traiter. Les servlets simplifie le processus en fournissant une vue orientée objet du monde HTTP (HttpRequest, HttpResponse, etc.).

-Cependant, le modèle des servlets était de trop bas niveau et c'est la raison pour laquelle, les pages JSP (JavaServer Pages) ont ensuite pris le relais pour simplifier la création des pages web dynamiques. En coulisse, une JSP est une servlet, sauf qu'elle est écrite essentiellement en HTML - avec un peu de Java pour effectuer les traitements.

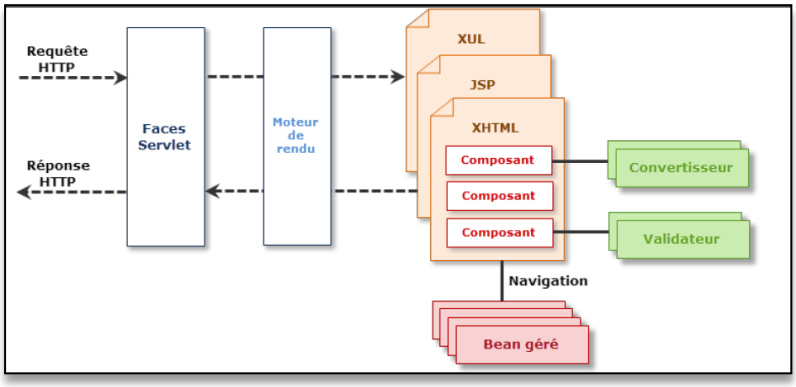
-JSF (JavaServer Faces, ou simplement Faces) a été créé en réponse à certaines limitations de JSP et utilise un autre modèle consistant à porter des composants graphiques vers le Web. Inspiré par le modèle Swing, JSF permet aux développeurs de penser en termes de composants, d'événements, de beans gérés et de leur interactions plutôt qu'en termes de requêtes, de réponses et de langages à marqueurs. Son but est de faciliter et d'accélérer le développement des applications web en fournissant des composants graphiques (comme des zones de texte, les listes, les onglets, les grilles, etc.) afin d'adopter une approche RAD (Rapid Application Development).

-Les applications JSF sont des applications web classiques qui interceptent HTTP via la servlet Faces et produisent du HTML.

En coulisse, cette architecture permet de greffer n'importe quel langage de déclaration de page (PDL), de l'afficher sur des dispositifs différents (navigateur web, terminaux mobiles, etc.) et de créer des pages au moyen d'événements, d'écouteurs et de composants, comme en Swing.

2-Architecture de JSF

La figure ci-dessous représente les parties les plus importantes de l'architecture JSF qui la rendent aussi riche et aussi souple :

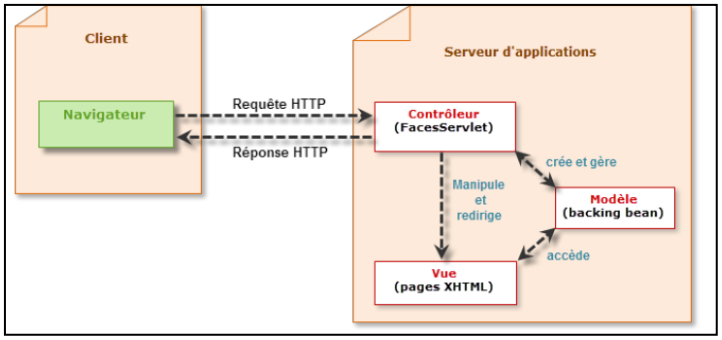


1. FacesServlet : est la servlet principale de l'application qui sert de contrôleur. JSF utilise le patron de conception MVC2 (Modèle-Vue-Contrôleur).

-MVC2 permet de découpler la vue (la page) et le modèle (les données affichées dans la vue). Le contrôleur prend en charge les actions de l'utilisateur qui pourraient impliquer des modifications dans le modèle et dans les vues. Avec JSF, ce contrôleur est la servlet FacesServlet.

-Toutes les requêtes de l'utilisateur passent systématiquement par elle, qui les examine et appelle les différentes actions correspondantes.

-En utilisant les beans gérés. Il est possible de configurer le comportement de cette servlet au travers d'annotations spécifiques (sur les beans gérés, les convertisseurs, les composants, les moteurs de rendu et les validateurs).



2. Pages et composants : JSF doit envoyer une page sur le dispositif de sortie du client (un navigateur, pas exemple) et exige donc une technologie d'affichage appelée PDL. Dans sa version la plus récente, JSF utilise plutôt les Facelets. Facelets est formée d'une arborescence de composants (également appelés widgets ou contrôles) fournissant des fonctionnalités spécifiques pour interagir avec l'utilisateur (champ de saisie, boutons, liste, etc.).

3 Moteurs de rendu : JSF reconnait deux modèles de programmation pour afficher les composants : l'implémentation directe et l'implémentation déléguée. Avec le modèle direct, les composants doivent eux-même s'encoder vers une représentation graphique et réciproquement. Avec le mode délégué, ces opérations sont confiées à un moteur de rendu, ce qui permet aux composants d'être indépendants de la technologie d'affichage (navigateur, terminal mobile, etc.) et donc d'avoir plusieurs représentations graphiques possibles

4 Convertisseurs et validateurs : Lorsque la page est affichée, l'utilisateur peut s'en servir pour entrer des données. Comme il n'y a pas de contraintes sur les types, un moteur de rendu ne peut pas prévoir l'affichage de l'objet. Voilà pourquoi les convertisseurs existent : ils traduisent un objet (Integer, Date, Enum, Boolean, etc.) en chaîne de caractères afin qu'il puisse s'afficher (protocole HTTP est un protocole uniquement textuel) et, inversement, construisent un objet à partir d'une chaîne qui a été saisie. JSF fournit un ensemble de convertisseurs pour les types classiques dans la paquetage javax.faces.convert, mais vous pouvez développer les vôtres ou ajouter des types provenant de tierces parties.

Parfois, les données doivent également être validées avant d'être traitées par le back-end : c'est le rôle des validateurs ; nous pouvons ainsi associer un ou plusieurs validateurs à un composant unique afin de garantir que les données saisies sont correctes. JSF fournit quelques validateurs (LengthValidator, RegexValidator, etc.) et vous permet d'en créer d'autres en utilisant vos propres classes annotées. En cas d'erreur de conversion ou de validation, un message est envoyé dans la réponse à afficher

5. Beans gérés et navigation : Tous les concepts que nous venons de présenter - qu'est-ce qu'une page, qu'est-ce qu'un composant, comment sont-il affichés convertis et validés - sont liés à une page unique, mais les applications web sont généralement formées de plusieurs pages et doivent réaliser un traitement métier (en appelant une couche EJB, par exemple). Le passage d'une page à une autre, l'invocation d'EJB et la synchronisation des données avec les composants sont pris en charge par les beans gérés.

3-Eléments de base de JSF

FacesServlet:

-FacesServlet est une implémentation de **javax.servlet.Servlet** qui sert de contrôleur central par lequel passent toutes les requêtes.

-La survenue d'un élément (lorsque l'utilisateur clique sur un bouton, par exemple) provoque l'envoi d'une notification au serveur *via* HTTP ; celle-ci est interceptée

par **javax.faces.webapp.FacesServlet**, qui examine la requête et exécute différentes actions sur le modèle a l'aide de beans gères.

-Les applications JSF ont besoin d'une servlet nommée **FacesServlet** qui agit comme un contrôleur frontal pour toute l'application. Cette servlet et son association doivent être définies dans le fichier **web.xml**

-Le descripteur de déploiement associe à la servlet les requêtes d'URL se terminant par une suffixe de votre choix par exemple ***.jsf**, ce qui signifie que toute demande d'une page se terminant par **.jsf** sera traitée par **FacesServlet.**

-La FacesServlet est interne aux implémentations de JSF ; bien que vous n'ayez pas accès a son code, vous pouvez la configurer avec des métadonnées

FacesContext

-JSF définit la classe abstraite **javax.faces.context.FacesContext** pour représenter les informations contextuelles associées au traitement d'une requête et à la production de la réponse correspondante. Cette classe permet d'interagir avec l'interface utilisateur et le reste de l'environnement JSF.

-Pour y accéder, vous devez soit utiliser l'objet implicite **facesContext** dans vos pages, soit obtenir une référence dans vos beans gères a l'aide de la méthode statique **getCurrentInstance()** : celle-ci renverra l'instance de **FacesContext** pour le thread courant

Beans gérés

-Comme on l'a indiqué plus haut, le modèle MVC encourage la séparation entre le modèle, la vue et le contrôleur. Avec Java EE, les pages JSF forment la vue et la FacesServlet est le contrôleur. Les beans gérés, quant a eux, sont une passerelle vers le modèle.

-Les beans gérés sont des classes Java annotées. Ils constituent le cœur des applications web car ils exécutent la logique métier (ou la délèguent aux EJB, par exemple), gèrent la navigation entre les pages et stockent les données. Une application JSF typique contient un ou plusieurs beans gérés qui peuvent être partages par plusieurs pages.

-Les données sont stockées dans les attributs du bean géré, qui, en ce cas, est également appelé "backing bean". Un backing bean définit les données auxquelles est lie un composant de l'interface utilisateur (la cible d'un formulaire, par exemple). Pour établir cette liaison, on utilise **EL, le langage d'expressions**.

-Les beans gérés sont des classes Java prises en charge par la FacesServlet. Les composants de l'interface utilisateur sont lies aux propriétés du bean (backing bean) et peuvent invoquer des méthodes d'action. Un bean géré doit respecter les contraintes suivantes :

-La classe doit être annotée par **@javax.faces.model.ManagedBean** ou son équivalent dans le descripteur de déploiement XML faces-config.xml.

-La classe doit avoir une portée (qui vaut par défaut @RequestScoped).

-La classe doit être publique et non finale ni abstraite.

-La classe doit fournir un constructeur public sans paramètre qui sera utilise par le conteneur pour créer les instances.

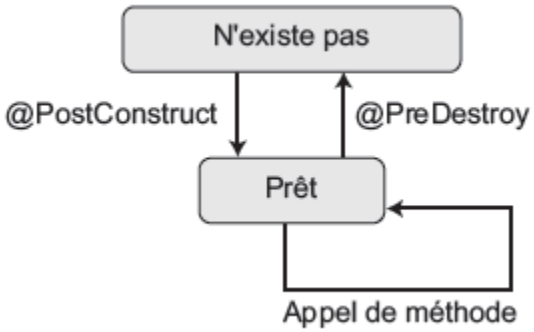
- La classe ne doit pas définir de méthode finalize().

-Pour être liés à un composant, les attributs doivent avoir des getters et des setters publics.

Bien qu'un bean gère puisse être un simple POJO annoté, sa configuration peut être personnalisée grâce aux éléments de @ManagedBean et @ManagedProperty (ou leurs équivalents XML).

@ManagedBean

La présence de l'annotation @javax.faces.model.ManagedBean sur une classe l'enregistre automatiquement comme un bean géré



-Cycle de vie

Les beans gères qui s'exécutent dans un conteneur de servlet peuvent utiliser les annotations @PostConstruct et @PreDestroy. Apres avoir crée une instance de bean géré, le conteneur appelle la méthode de rappel @PostConstruct s'il y en a une.

Puis le bean est lié a une portée et répond a toutes les requêtes de tous les utilisateurs.

Avant de supprimer le bean, le conteneur appelle la méthode @PreDestroy.

Ces méthodes permettent donc d'initialiser les attributs ou de créer et libérer les ressources externes

Portées

Les objets créés dans le cadre d'un bean géré ont une certaine durée de vie et peuvent ou non être accessibles aux composants de l'interface utilisateur ou aux objets de l'application. Cette durée de vie et cette accessibilité sont regroupées dans la notion de ***portée***. Cinq annotations permettent de définir la portée d'un bean géré :

@ApplicationScoped. Il s'agit de l'annotation la moins restrictive, avec la plus longue durée de vie. Les objets créés sont disponibles dans tous les cycles requête/ réponse de tous les clients utilisant l'application tant que celle-ci est active. Ces objets peuvent être appelés de façon concurrente et doivent donc être threadsafe (c'est-a-dire utiliser le mot-clé synchronized). Les objets ayant cette portée peuvent utiliser d'autres objets sans portée ou avec une portée d'application.

@SessionScoped. Ces objets sont disponibles pour tous les cycles requête/réponse de la session du client. Leur état persiste entre les requêtes et dure jusqu'a la fin de la session. Ils peuvent utiliser d'autres objets sans portée, avec une portée de session ou d'application.

@ViewScoped. Ces objets sont disponibles dans une vue donnée jusqu'a sa modification. Leur état persiste jusqu'a ce que l'utilisateur navigue vers une autre vue auquel cas il est supprime. Ils peuvent utiliser d'autres objets sans portée, avec une portée de vue, de session ou d'application.

@RequestScoped. Il s'agit de la portée par défaut. Ces objets sont disponibles du début d'une requête jusqu'au moment ou la réponse est envoyée au client.

@NoneScoped. Les beans gérés ayant cette portée ne sont visibles dans aucune page JSF ; ils définissent des objets utilises par d'autres beans gérés de l'application. Ils peuvent utiliser d'autres objets avec la même portée.

****

@ManagedProperty

Dans un bean géré, vous pouvez demander au système d'injecter une valeur dans une propriété (un attribut avec des getters et/ou des setters) en utilisant le fichier faces-config.xml ou l'annotation @javax.faces.model.ManagedProperty, dont l'attribut value peut recevoir une chaine ou une expression EL.

Expression de language

JSF EL, propose un mécanisme de communication entre les composants de la couche Vue et la logique applicative représentée par les Managed Beans.

Afin d'accéder en écrire comme en lecture aux propriétés du Managed Bean UserBean.java définit par l'annotaion @ManagedBean(name="userbean") on utilise le nom du bean "userbean" suivi du nom de la propriété ou du nom de la méthode du Managed bean **#{userbean.xxx}**

Les expressions EL peuvent utiliser la plupart des opérateurs Java habituels :

1. Arithmétiques : + - * / (division) % (modulo)

2. Relationnels : == (égalité) != (inégalité) < (inférieur) > (supérieur) <= (inférieur ou égal) >= (supérieur ou égal)

3. Logiques : && (et) || (ou) ! (non)

4. Autre : empty (vide) () (parenthèses) [ ] (crochets) . (séparateur) ?: (if arithmétique)

4-Facelets

-Comme **JSP**, ***Facelets*** est une technologie de présentation pour le développement d'applications web en *Java*.

-Une page *JSP* est transformée en une *Servlet* qui possède un cycle de vie différent de celui de *JSF*, ce qui peut être source de confusion et de problèmes. A l'inverse, *Facelets* est spécifiquement développé pour *JSF* et est plus performant et léger.

-*Facelets* introduit aussi des fonctionnalités au-delà de celles offertes par le *JSP*, comme par exemple un système de *templating* ou encore la possibilité de créer des composants personnalisés sans écrire la moindre ligne de code *Java*.

-*Facelets* est basé sur *xml*, c'est pour cette raison que les vues sous *facelets* sont des pages *xhtml* (ou encore *jspx*) et qu'elles doivent impérativement respecter la structure d'un document *xml*.

XHTML

XHTML a été crée peu de temps après HTML 4.01. Ses racines puisent dans HTML, mais avec une reformulation en XML strict. Ceci signifie qu'un document XHTML est un document XML qui respecte un certain schéma et peut être représenté graphiquement par les navigateurs – un fichier XHTML (qui porte l'extension .xhtml) peut être directement utilise comme du XML ou être affiche dans un navigateur

Bibliothèques de marqueurs autorisés avec le PDL Facelets

URI	***Préfixe classique***	***Description***
http://xmlns.jcp.org/jsf/html	h	Contient les composants et leurs rendus HTML (h:commandButton, h:commandLink, h:inputText, etc.).
http://xmlns.jcp.org/jsf/core	f	Contient les actions personnalisées

		independantes d'un rendu particulier (f:selectItem, f:validateLength, f:convertNumber, etc.).
http://xmlns.jcp.org/jsf/facelets	ui	Marqueurs pour le support des templates.

Gestion des ressources

La plupart des composants ont besoin de ressources externes pour s'afficher correctement <h:graphicImage> a besoin d'une image, <h:commandButton> peut également afficher une image pour représenter le bouton, <h:outputScript> référence un fichier JavaScript et les composants peuvent également appliquer des styles CSS.

Avec JSF, une ressource est un élément statique qui peut être transmis aux éléments afin d'être affiche (images) ou traite (JavaScript, CSS) par le navigateur.

Les versions précédentes de JSF ne fournissaient pas de mécanisme particulier pour servir les ressources : lorsque l'on voulait en fournir une, il fallait la placer dans le répertoire WEB-INF pour que le navigateur du client puisse y accéder. Pour la modifier, il fallait remplacer le fichier et, pour gérer les ressources localisées (une image avec un texte anglais et une autre avec un texte français, par exemple), il fallait utiliser des répertoires différents. JSF 2.0 permet désormais d'assembler directement les ressources dans un fichier jar séparé, avec un numéro de version et une locale, et de le placer à la racine de l'application web, sous le répertoire suivant :

resources/<identifiant_ressource>

ou :

META-INF/resources/<identifiant_ressource>

<**identifiant_ressource**> est forme de plusieurs sous-répertoires indiques sous la forme :

[locale/][nomBib/][versionBib/]nomRessource[/versionRessource]

Tous les éléments entre crochets sont facultatifs. La locale est le code du langage, suivi éventuellement d'un code de pays (en, en_US, pt, pt_BR). Comme l'indique cette syntaxe, vous pouvez ajouter un numéro de version à la bibliothèque ou à la ressource elle-même. Voici quelques exemples :

book.gif

en/book.gif

en_us/book.gif

en/myLibrary/book.gif

myLibrary/book.gif

myLibrary/1_0/book.gif

myLibrary/1_0/book.gif/2_3.gif

Vous pouvez ensuite utiliser une ressource – l'image **book.gif,** par exemple – directement dans un composant <h:graphicImage> ou en précisant le nom de la bibliothèque (library="myLibrary"). La ressource correspondant a la locale du client sera automatiquement choisie.

<h:graphicImage value="book.gif" />

<h:graphicImage value="book.gif" library="myLibrary" />

<h:graphicImage value="#{resource['book.gif']}" />

<h:graphicImage value="#{resource['myLibrary:book.gif']}" />

Templating

-Le *templating* consiste à factoriser la structure commune d'un ensemble de pages et à l'extraire dans une nouvelle page, appelée *template* ou modèle. Les autres pages utilisent alors le *template* comme structure et y injectent leur contenu spécifique.

Le templating offre plusieurs avantages :

- Uniformiser la structure des pages
- Simplifier la mise à jour : une modification dans le *template* se propage automatiquement dans toutes les pages qui l'utilisent.
- Gain en productivité : moins de code à écrire : une page ne contient que ce qui lui est propre.

-Dans *facelets*, un *template* est une **page *xhtml*** ordinaire qui définit la structure du document avec des emplacements spécifiques où les pages qui utilisent ce *template* (pages clientes) insèreront leur contenu. Pour définir un tel emplacement, on utilise le *tag* **<*ui :insert*>** avec comme paramètre le nom logique de cet emplacement : en-tête, menu, etc. Le tag **<*ui :insert*>** comme tous les autres *tags* préfixés par " ui " sont fournis par facelets.

-En général, le *tag* **<*ui:insert*>** a un corps vide puisqu'il sera remplacé par le contenu fourni par la page cliente. Si par contre ce corps n'est pas vide, alors son contenu sera affiché lorsque la page cliente ne spécifie pas son propre contenu.

-On commence par spécifier que l'on utilise un *template* avec la balise **<*ui:composition*>** qui prend comme paramètre le chemin vers le fichier contenant le *template*. Pour définir les

différents blocs qui seront injectés dans le *template*, on utilise la balise ***<ui:define>***. Cette balise prend comme paramètre le nom logique du bloc correspondant dans le *template* et contient dans son corps le contenu à injecter dans le *template*.

Marqueur	***Description***
<ui:composition>	Definit une composition utilisant eventuellement un template. Le meme template peut etre utilise par plusieurs compositions.
<ui:define>	Definit un contenu qui sera insere dans l'element <ui:insert> correspondant du template.
<ui:decorate>	Permet de décorer le contenu d'une page.
<ui:fragment>	Ajoute un fragment a une page.
<ui:insert>	Définit un point d'insertion dans un template dans lequel on pourra ensuite inserer un contenu place dans un marqueur <ui:define>.

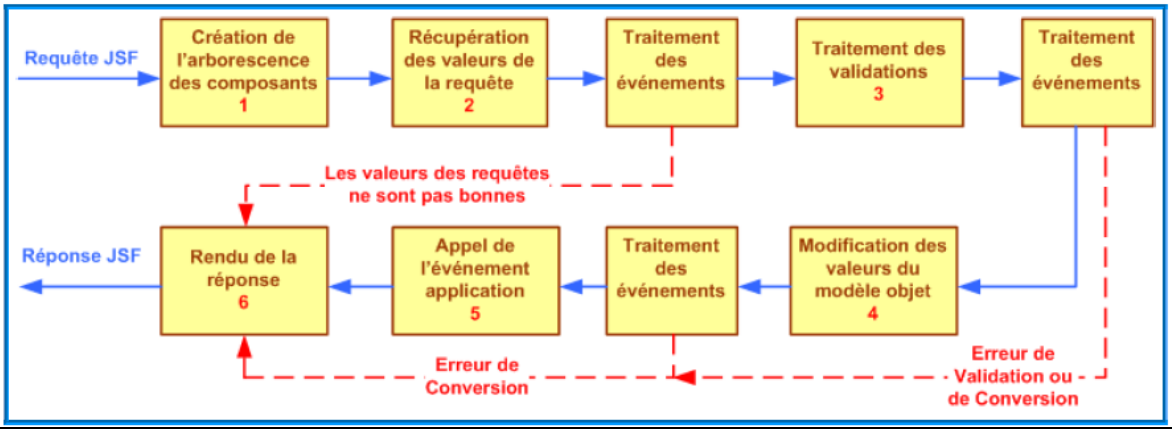
5-Cycle de vie

-Une page JSF est une arborescence de composants avec un cycle de vie spécifique qu'il faut bien avoir compris pour savoir à quel moment les composants sont validés ou quand le modèle est mis à jour.

- Un clic sur un bouton provoque l'envoi d'une requête du navigateur vers le serveur et cette requête est traduite en événement qui peut être traité par l'application sur le serveur.

Toutes les données saisie par l'utilisateur passent par une étape de validation avant que le modèle soit mis à jour et que du code métier soit appelé.

- JSF se charge alors de vérifier que chaque composant graphique (composants parent et fils) est correctement rendu par le navigateur.



Le cycle de vie de JSF se divise en six phases

1. Restauration de la vue : JSF trouve la vue cible et lui applique les entrées de l'utilisateur. S'il s'agit de la première visite, JSF crée la vue comme un composant UIViewRoot (racine de l'arborescence de composants, qui constitue une page particulière). Pour les requêtes suivantes, il récupère l'UIViewRoot précédemment sauvegardée pour traiter la requête HTTP courante.

2. Application des valeurs de la requête : Les valeurs fournies avec la requête (champ de saisie, d'un formulaire, valeurs des cookies ou à partir des entêtes HTTP) sont appliquées aux différents composants de la page. Seuls les composants UI (de la page) modifient leur état, non les objets qui forment le modèle.

3. Validations : Lorsque tous les composants UI ont reçu leurs valeurs, JSF traverse l'arborescence de composants et demande à chacun d'eux de s'assurer que la valeur qui leur a été soumise est correcte. Si la conversion et la validation réussissent pour tous les composants, le cycle de vie passe à la phase suivante. Sinon il passe à la phase de Rendu de la réponse avec les messages d'erreur de validation et de conversion appropriés.

4. Modification des valeurs du modèle : Lorsque toutes les valeurs des composants ont été affectées et validées, les beans gérés qui leur sont associés peuvent être mis à jour.

5. Appel de l'application : Nous pouvons maintenant exécuter la logique métier. Les actions qui ont été déclenchées seront exécutées sur le bean géré.

La navigation entre en jeu car c'est la valeur qu'elle renvoie qui déterminera la réponse.

6. Rendu de la réponse : Le but principal de cette phase consiste à renvoyer la réponse à l'utilisateur. Son but secondaire est de sauvegarder l'état de la vue pour pouvoir la restaurer dans la phase de restauration si l'utilisateur redemande la vue

6-Navigation:

Les applications web sont formées de plusieurs pages entre lesquelles vous devez naviguer. Selon le cas, il peut exister différents niveaux de navigation avec des flux de pages plus ou moins élaborés.

JSF dispose de plusieurs options de navigation et vous permet de contrôler le flux page par page ou pour toute l'application.

Navigation statique:

Les composants <h:commandButton> et <h:commandLink> permettent de passer simplement d'une page à une autre en cliquant sur un bouton ou sur un lien sans effectuer aucun traitement

intrinsèque. Il suffit d'initialiser leur attribut action avec le nom de la page vers laquelle vous voulez vous rendre

<h:commandButton value="Authentification" action="profil.xhtml" />

Navigation dynamique

Cependant, la plupart du temps, ceci ne suffira pas car vous aurez besoin d'accéder à une couche métier ou à une base de données pour récupérer ou traiter des données. En ce cas, vous aurez besoin d'un bean géré.

Il suffit juste de préciser la méthode du bean géré qui va traiter la logique métier Authentification) et s'occuper ensuite de la navigation.

<h:commandButton value="Authentification" action="#{loginBean.authentificate} " />

Les composants bouton et lien n'appellent pas directement la page vers laquelle ils doivent se rendre : ils appellent des méthodes du bean géré qui prennent en charge cette navigation et laissent le code décider de la page qui sera chargée ensuite. La navigation utilise un ensemble de règles qui définissent tous les chemins de navigation possibles de l'application. Dans la forme la plus simple de ces règles de navigation, chaque méthode du bean géré définit directement la page vers laquelle elle doit aller.

Différence entre Redirect et Foward

-Redirect redirige complètement le serveur vers la page appelé et produit donc une nouvelle requête complète. Du coup au niveau du navigateur ce n'est pas la même URL qui est affichée.

Pour la mettre en œuvre, il suffit de rajouter la chaîne suivante :

?faces-redirect=true

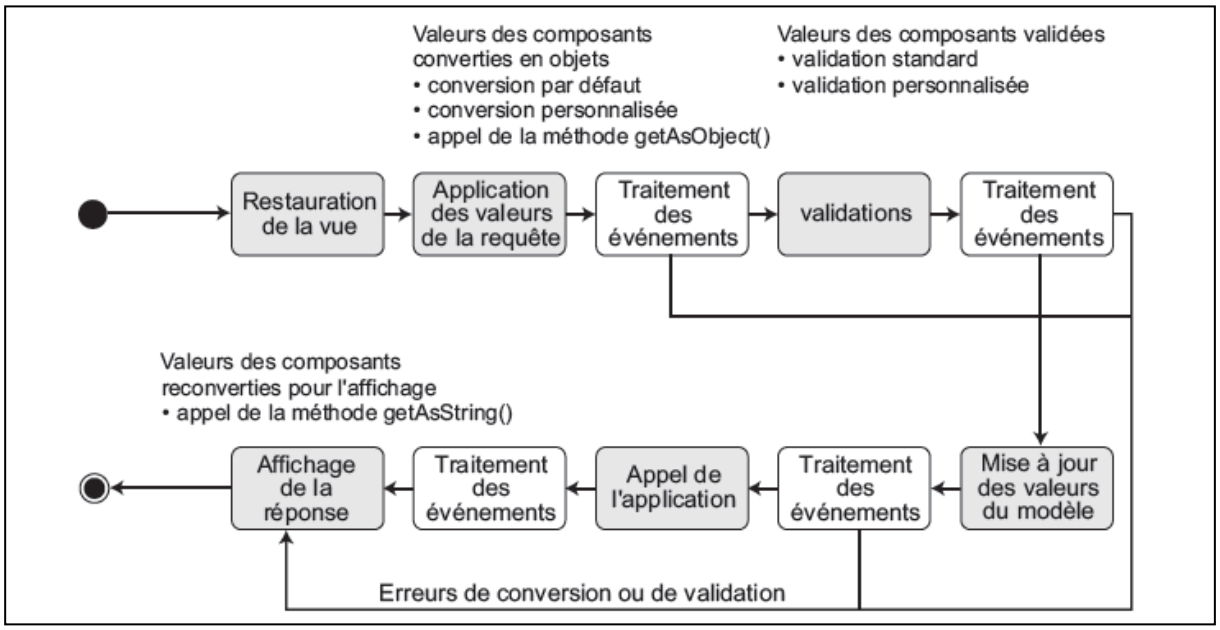
<h:commandButton value="Accueil" action="accueil?faces-redirect=true" />

-Foward change simplement l'action en cours pour une autre. On reste dans la même transaction HTTP. Pour le client, on est donc toujours dans la même URL. Ce qui veut dire que tu n'as pas besoin de repasser les paramètres à la nouvelle action si tu souhaites accéder aux paramètres envoyés par le client

7-La conversion et la validation des données:

JSF fournit un mécanisme standard de conversion et de validation permettant de traiter les saisies de utilisateurs afin d'assurer l'intégrité des données. Lorsque vous invoquez des méthodes métiers, vous pouvez donc vous fier à des données valides : la conversion et la validation permettent aux développeur de ce concentrer sur la logique métier au lieu de passer du temps à vérifier que les données saisies ne sont pas null, qu'elles appartiennent bien à un intervalle précis, etc.

-La conversion a lieu lorsque les données saisies par l'utilisateur doivent être transformées de String en un objet et vice versa (le protocole HTTP ne manipule que les chaînes de caractères). Elle garantit que les informations sont du bon type - en convertissant, par exemple, un String en java.util.Date, un String en Integer ou des dollars en euros. Comme pour la validation, elle garantit que les données contiennent ce qui est attendu (une date au format jj/mm/aaaa, un réel compris entre 3.14 et 3.15, etc.).



1. Au cours de la phase Application des valeurs de la requête, la valeur du composant de l'interface est convertie dans l'objet cible puis validée au cours de la phase de traitement des Validations.
2. Il est logique que la conversion et la validation interviennent avant que les données du composant ne soient liées au bean géré qui réalise les traitements métiers en coulisse (qui a lieu au cours de la phase Modification des valeurs du modèle).
3. En cas d'erreur, des messages d'alertes seront ajoutés et le cycle de vie sera écourté afin de passer directement à l'affichage de la Réponse (les messages seront alors affichés sur l'interface utilisateur avec <h:messages>).
4. Au cours de cette dernière phase, les propriétés du bean géré seront reconverties en chaînes de caractères afin d'être affichées.

-JSF fournit un ensemble de convertisseurs et de validateurs standard et vous permet également de créer les vôtres très facilement.

Les conversions standards:

Lorsqu'un formulaire est affiche par un navigateur, l'utilisateur remplit les champs et appuie sur un bouton ayant pour effet de transporter les données vers le serveur dans une requête HTTP formée de chaines. Avant de mettre a jour le modèle du bean géré, ces données textuelles doivent être converties dans les objets cibles (Float, Integer, BigDecimal, etc.). L'opération inverse aura lieu lorsque les données seront renvoyées au client dans la réponse pour être affichées par le navigateur. JSF fournit des convertisseurs pour les types classiques comme les dates et les nombres. Si une propriété du bean géré est d'un type primitif (Integer, int, Float, float, etc.), JSF convertira automatiquement la valeur du composant d'interface dans le type adéquat et inversement. Si elle est d'un autre type, vous devrez fournir votre propre convertisseur.

Classes enveloppes	Convertisseur
java.math.BigDecimal	javax.faces.convert.BigDecimalConverter
java.math.BigInteger	javax.faces.convert.BigIntegerConverter
java.lang.Boolean	javax.faces.convert.BooleanConverter
java.lang.Byte	javax.faces.convert.ByteConverter
java.lang.Character	javax.faces.convert.CharacterConverter
java.util.Date	javax.faces.convert.DateTimeConverter
java.lang.Double	javax.faces.convert.DoubleConverter
java.lang.enum	javax.faces.convert.EnumConverter
java.lang.Float	javax.faces.convert.FloatConverter
java.lang.Integer	javax.faces.convert.IntegerConverter
java.lang.Long	javax.faces.convert.LongConverter
java.lang.Number	javax.faces.convert.NumberConverter
java.lang.Short	javax.faces.convert.ShortConverter

Les Conversions personnalisés:

Parfois, la conversion de nombres, de dates, d'énumérations, etc. ne suffit pas et nécessite une conversion adaptée a la situation. Il suffit pour cela d'écrire une classe qui implémente l'interface javax.faces.convert.Converter et de lui associer des métadonnées. Cette interface expose deux méthodes :

Object **getAsObject**(FacesContext ctx, UIComponent component, String value)

String **getAsString**(FacesContext ctx, UIComponent component,Object value)

La méthode getAsObject() convertit la valeur chaine d'un composant d'interface utilisateur dans le type correspondant et renvoie la nouvelle instance ; elle lance une exception ConverterException si la conversion échoue. Inversement, getAsString() convertit l'objet en chaine afin qu'il puisse être affiche a l'aide d'un langage à marqueurs (comme XHTML).

-Pour utiliser ce convertisseur dans l'application web, il faut l'enregistrer : une méthode consiste à le déclarer dans le fichier faces-config.xml, l'autre, a utiliser

l'annotation @FacesConverter.

Les Validateurs personnalisés:

Les applications web doivent garantir que les données saisies par les utilisateurs sont appropriées. Cette vérification peut avoir lieu cote client avec JavaScript ou cote serveur avec des validateurs.

JSF simplifie la validation des données en fournissant des validateurs standard et en permettant d'en créer de nouveaux, adaptés à vos besoin. Les validateurs agissent comme des contrôles de premier niveau en validant les valeurs des composants de l'interface utilisateur avant qu'elles ne soient traitées par le bean géré.

Balises	Classe de validation	Attributs	Valeur
<f:validateDoubleRange>	DoubleRangeValidator	minimum maximum	Compare la valeur du composant aux valeurs minimales et maximales indiquées (de type double).
<f:validateLongRange>	LongRangeValidator	minimum maximum	Compare la valeur du composant aux valeurs minimales et maximales indiquées (de type long).
<f:validateLength>	LenghtValidato	minimum maximum	Teste le nombre de caractères de la valeur textuelle du composant
<f:validateRequired>	RequiredValidator		Vérife qu'une valeur est bien saisie dans le composant (donnée obligatoire).
<f:validateRegex>	RegexValidator	*pattern*	Vérifie qu'un groupe de validations est bien respecté au travers d'un composant spécifique dont les propriétés sont déjà préétablies.

Exemple:

```
<h:inputText value="#{employeeBean.employee.login}" required="true">
<f:validateLength minimum="4" maximum="8"/>
</h:inputText>
<h:inputText value="#{employeeBean.employee.age}">
<f:validateLongRange minimum="18" maximum="50"/>
</h:inputText>
```

Les Validateurs personnalisés:

Les validateurs standard de JSF peuvent ne pas convenir a vos besoins : vous avez peut-être des données qui respectent certains formats métier, comme un code postal ou une adresse de courrier électronique. En ce cas, vous devrez créer votre propre validateur. Comme les convertisseurs, un validateur est une classe qui doit implémenter une interface et redéfinir une méthode. Dans le cas des validateurs, cette interface est javax.faces.validator.Validator, qui n'expose que la méthode validate() :

void validate(FacesContext context, UIComponent component, Object value)

Le paramètre value est celui qui doit être vérifié en fonction d'une certaine logique métier. S'il passe le test de validation, vous pouvez simplement sortir de cette méthode et le cycle de la page se poursuivra. Dans le cas contraire, vous pouvez lancer une exception ValidatorException et inclure un FacesMessage avec des messages résumés et détaillés pour décrire l'erreur de validation. Ce validateur doit être enregistré dans le fichier faces-config.xml ou a l'aide de l'annotation @FacesValidator.