**Créer une appli web avec JEE**

**Introduction au Java EE**

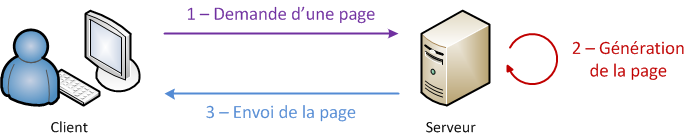
« Java EE » signifie Java Enterprise Edition

La plate-forme Java EE est construite sur le langage Java et la plate-forme Java SE (Java Standard Edition).

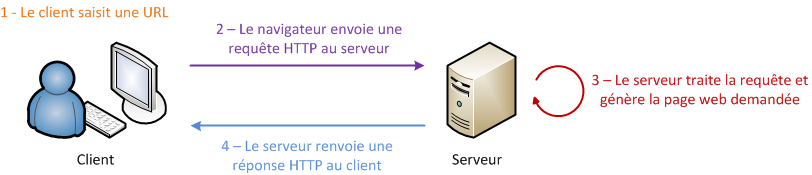
L'internet est le réseau, le support physique de l'information. Le web constitue une partie seulement du contenu accessible sur l'internet. Vous connaissez et utilisez d'autres contenus, comme le courrier électronique ou encore la messagerie instantanée. Un site web est un ensemble constitué de pages web. On distingue deux types de sites :

* **les sites internet statiques** : ce sont des sites dont le contenu est « fixe », il n'est modifiable que par le propriétaire du site. Ils sont réalisés à l'aide des technologies HTML, CSS et Javascript uniquement.
* **les sites internet dynamiques** : ce sont des sites dont le contenu est « dynamique », parce que le propriétaire n'est plus le seul à pouvoir le faire changer ! En plus des langages précédemment cités, ils font intervenir d'autres technologies : Java EE est l'une d'entre elles !

Consultation d’un site Web :



La communication qui s'effectue entre le client et le serveur est régie par des règles bien définies : le **protocole http** :



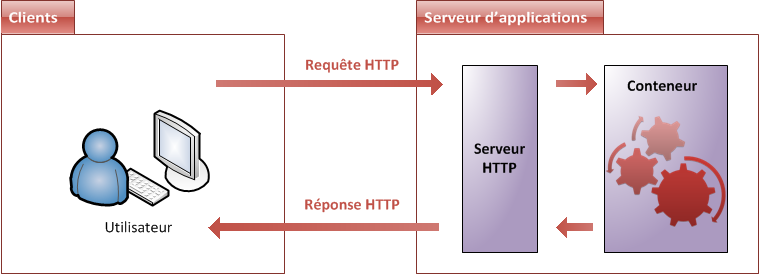
Rem : Le client ne comprend que les langages de présentation de l'information, en d'autres termes les technologies HTML, CSS et Javascript !

**Le Java EE mis à nu !**

Avec Java EE, l’application est découpée en couches, et le serveur sur lequel tourne l'application est lui-même découpé en plusieurs niveaux. Java EE fournit un ensemble d’extensions au Java standard afin de faciliter la création d’applications centralisées.

Le client et le serveur doivent se parler via HTTP. Côté client, le navigateur s'en occupe. Côté serveur, c’est le **serveur HTTP**. Il doit écouter tout ce qui arrive sur le port utilisé par le protocole HTTP, le port 80, et scruter chaque requête entrante. C'est en somme une interface de communication avec le protocole.

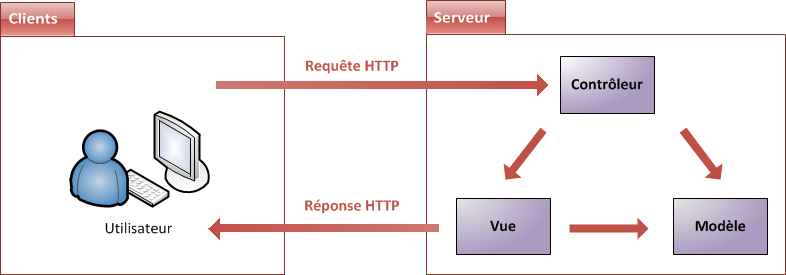
Le **serveur d'applications** va se charger d'exécuter votre code en plus de faire le travail du serveur http.



**Le modèle MVC : en théorie**

En anglais un design pattern, un modèle de conception (ou encore patron de conception) est une simple **bonne pratique**, qui répond à un problème de conception d'une application. C'est en quelque sorte une ligne de conduite qui permet de décrire les grandes lignes d'une solution. [Le modèle MVC (Modèle-Vue-Contrôleur)](http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller) en est un exemple. Voici ce qu’il impose :

* tout ce qui concerne le traitement, le stockage et la mise à jour des données de l'application doit être contenu dans la couche nommée "Modèle" (le M de MVC) .
* tout ce qui concerne l'interaction avec l'utilisateur et la présentation des données (mise en forme, affichage) doit être contenu dans la couche nommée "Vue" (le V de MVC) ;
* tout ce qui concerne le contrôle des actions de l'utilisateur et des données doit être contenu dans la couche nommée "Contrôle" (le C de MVC).



Un framework est un ensemble de composants qui servent à créer l'architecture et les grandes lignes d'une application. Vous pouvez le voir comme une boîte à outils géante, conçue par un ou plusieurs développeurs et mise à disposition d'autres développeurs, afin de faciliter leur travail.

**Modèle : des traitements et des données**

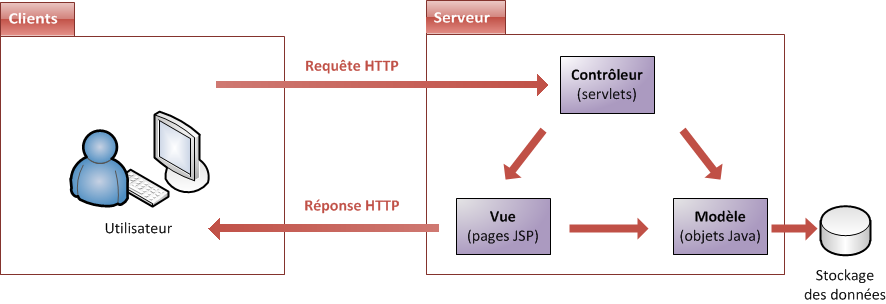
Dans le modèle, on trouve à la fois les données et les traitements à appliquer à ces données. Ce bloc contient donc des objets Java d'une part, et un système capable de stocker des données d'autre part.

**Vue : des pages JSP**

Une page JSP est destinée à la vue. Elle est exécutée côté serveur et permet l'écriture de gabarits (pages en langage "client" comme HTML, CSS, Javascript, XML, etc.). Elle permet au concepteur de la page d'appeler de manière transparente des portions de code Java, via des balises et expressions ressemblant fortement aux balises de présentation HTML.

**Contrôleur : des servlets**

Une servlet est un objet qui permet d'intercepter les requêtes faites par un client, et qui peut personnaliser une réponse en conséquence. Il fournit pour cela des méthodes permettant de scruter les requêtes [HTTP](http://www.w3.org/Protocols/rfc2616/rfc2616.html). **Cet objet n'agit jamais directement sur les données, il faut le voir comme un simple aiguilleur** : il intercepte une requête issue d'un client, appelle éventuellement des traitements effectués par le modèle, et ordonne en retour à la vue d'afficher le résultat au client.



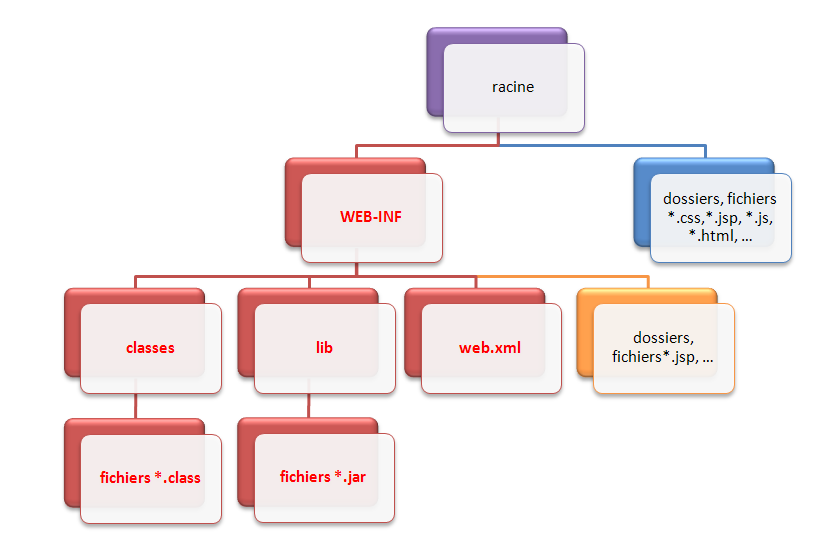
Dans une application Java EE sans frameworks la couche Modèle est constituée d'objets Java, la couche Vue est constituée de pages JSP et la couche Contrôle est constituée de servlets.

**Outils et environnement de développement**

IDE = Environnement de Développement Intégré.

**Structure d'une application Java EE**

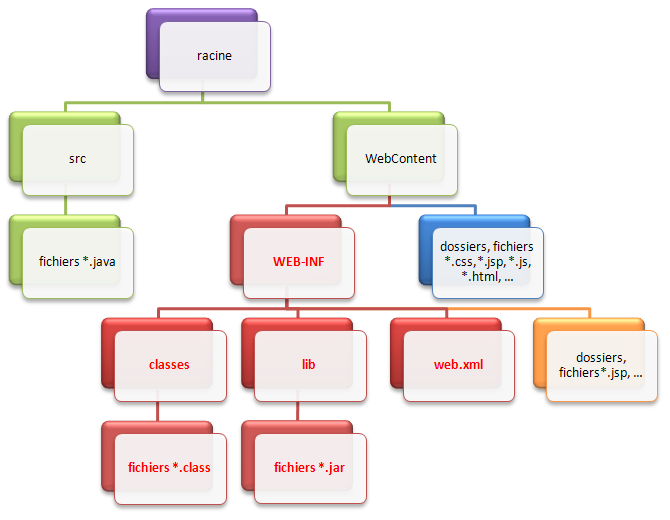
Structure de dossiers standard définie dans les spécifications de la plateforme :



* La racine de l'application est le dossier qui porte le nom du projet et qui contient l'intégralité des dossiers et fichiers de l'application.
* Le dossier nommé **WEB-INF** est un dossier spécial. Il doit obligatoirement exister et être placé juste sous la racine de l'application. Il doit à son tour obligatoirement contenir :
  + le fichier de configuration de l'application (web.xml)
  + un dossier nommé **classes**, qui contient à son tour les classes compilées (fichiers .class)
  + un dossier nommé **lib**, qui contient à son tour les bibliothèques nécessaires au projet (archives .jar)
* Les fichiers et dossiers placés directement sous la racine, en bleu sur le schéma, sont publics et donc accessibles directement par le client via leurs URL.
* Les fichiers et dossiers placés sous le répertoire **WEB-INF**, en orange sur le schéma, sont privés et ne sont donc pas accessibles directement par le client.

**Structure d'une application Java EE sous Eclipse**

Eclipse adapte souvent certaines spécificités à son mode de fonctionnement.



Eclipse déplace la structure standard de l'application vers un dossier nommé **WebContent**, et ajoute sous la racine un dossier **src** qui contiendra le code source de vos classes (les fichiers .java). Ceci implique donc deux choses très importantes :

* le dossier **WebContent** n'existe légitimement qu'au sein d'Eclipse. Si vous développez sans IDE, ce répertoire ne doit pas exister et votre application doit impérativement suivre la structure standard présentée précédemment.
* pour cette même raison, si on souhaite utiliser votre application en dehors de l'IDE, il faudra obligatoirement utiliser l'outil d'export proposé par Eclipse. Réaliser un simple copier-coller des dossiers ne fonctionnera pas en dehors d'Eclipse !

**La servlet**

**Méthode http**

##### **GET :** C'est la méthode utilisée par le client pour récupérer une ressource web du serveur via une URL. Lorsqu'il reçoit une telle demande, le serveur ne fait pas que retourner la ressource demandée, il en profite pour l'accompagner d'informations diverses à son sujet, dans ce qui s'appelle les en-têtes ou headers HTTP. Il est possible de transmettre des données au serveur lorsque l'on effectue une requête GET, au travers de paramètres directement placés après l'URL (paramètres nommés query strings) ou de cookies placés dans les en-têtes de la requête. La taille d'une URL est limitée, **on ne peut pas utiliser cette méthode pour envoyer des données volumineuses au serveur**, par exemple un fichier. Via cette méthode GET, il est uniquement possible de **récupérer ou de lire** des informations, sans que cela ait un quelconque impact sur la ressource demandée : ainsi, une requête GET est censée pouvoir être répétée indéfiniment sans risques pour la ressource concernée.

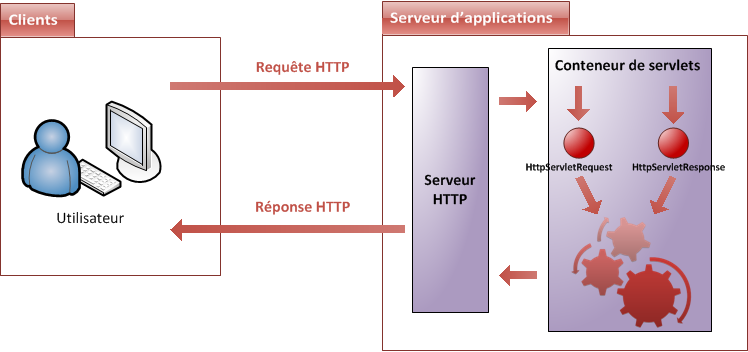
##### **POST :** La taille du corps du message d'une requête POST n'est pas limitée, c'est donc cette méthode qu'il faut utiliser pour soumettre au serveur des données de tailles variables, ou que l'on sait volumineuses. Cette méthode doit être utilisée pour réaliser les opérations qui ont un effet sur la ressource, et qui ne peuvent par conséquent pas être répétées sans l'autorisation explicite de l'utilisateur.

##### **HEAD :** Cette méthode est identique à la méthode GET, à ceci près que le serveur n'y répondra pas en renvoyant la ressource accompagnée des informations la concernant, mais **seulement ces informations**. En d'autres termes, il renvoie seulement les en-têtes HTTP ! Il est ainsi possible par exemple de vérifier la validité d'une URL ou de vérifier si le contenu d'une page a changé ou non sans avoir à récupérer la ressource elle-même.

**Sur le serveur**

Lorsque le **serveur HTTP** reçoit la requête du client, il la transmet au **conteneur de servlets** également nommé **conteneur web**. Celui-ci va alors créer deux nouveaux objets :

* [HttpServletRequest](http://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpServletRequest.html) : cet objet contient la requête HTTP, et donne accès à toutes ses informations, telles que les en-têtes (*headers*) et le corps de la requête.
* [HttpServletResponse](http://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpServletResponse.html) : cet objet initialise la réponse HTTP qui sera renvoyée au client, et permet de la personnaliser, en initialisant par exemple les en-têtes et le corps.



Le conteneur de servlets les transmet à l’application, et plus précisément aux servlets et filtres que l’on a éventuellement mis en place.

Une servlet est en réalité une simple classe Java, qui a la particularité de **permettre le traitement de requêtes et la personnalisation de réponses**. En principe, une servlet dans son sens générique est capable de gérer n'importe quel type de requête, mais dans les faits il s'agit principalement de requêtes HTTP. L'usage veut qu'on ne s'embête pas à préciser "servlet HTTP" lorsque l'on parle de ces dernières.

L’interface « **SERVLET** » est **l'interface mère que toute servlet doit obligatoirement implémenter**. Il existe de nombreuses classes préconçues pour gérer des requêtes. Ex : [HttpServlet](http://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpServlet.html)

La classe abstraite [HttpServlet](http://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpServlet.html) propose **les méthodes Java nécessaires au traitement des requêtes et réponses HTTP** ! Ainsi, on y trouve les méthodes :

* doGet() pour gérer la méthode GET
* doPost() pour gérer la méthode POST
* doHead() pour gérer la méthode HEAD

Sa méthode service() gère automatiquement l’association de chaque type de requête HTTP à la méthode Java qui lui correspond.

Ainsi, une servlet HTTP **doit hériter** de la classe abstraite **HttpServlet.** Une servlet **doit implémenter** au moins une des méthodes doXXX() afin d'être capable de traiter une requête entrante. Si cela n’est pas fait, un message d’erreur est renvoyé par la méthode doXXX() de la classe mère lorsque l’on essaye de l’appeler. Puisque ce sont elles qui prennent en charge les requêtes entrantes, **les servlets vont être les points d'entrée de notre application web, c'est par elles que tout va passer**. Contrairement au Java SE, il n'existe pas en Java EE de point d'entrée unique prédéfini, comme pourrait l'être la méthode main().

Pour pouvoir jouer correctement leur rôle d’aiguilleur, les servlets doivent être enregistrée auprès de l’application. Ce mapping entre une servlet et l’URL qui doit permettre d’y accéder se fait via le fichier web.xml. C'est le cœur de l’ application : ici vont se trouver tous les paramètres qui contrôlent son cycle de vie. **La mise en place d'une servlet se déroule en deux étapes : nous devons d'abord déclarer la servlet, puis lui faire correspondre une URL.**

**Définition de la servlet**

Pour ce faire, il faut ajouter une section au fichier web.xml qui se présente ainsi sous sa forme minimale :

<servlet>

<servlet-name>Test</servlet-name>

<servlet-class>com.sdzee.servlets.Test</servlet-class>

</servlet>

La balise responsable de la définition d'une servlet se nomme logiquement <servlet>, et les deux balises obligatoires de cette section sont très explicites :

* <servlet-name> permet de donner un nom à une servlet. C'est ensuite via ce nom qu'on fera référence à la servlet en question.
* <servlet-class> sert à préciser le chemin de la classe de la servlet dans votre application.

**Bonne pratique** : gardez un nom de classe et un nom de servlet identiques.

Autres balises facultatives :

* <description> permet de décrire plus amplement le rôle de la servlet. Cette description n'a aucune utilité technique et n'est visible que dans ce fichier.
* <init-param> permet de préciser des paramètres qui seront accessibles à la servlet lors de son chargement.
* <load-on-startup> permet de forcer le chargement de la servlet dès le démarrage du serveur.

**Mapping de la servlet**

Il faut ensuite faire correspondre la servlet déclarée à une URL :

<servlet-mapping>

<servlet-name>Test</servlet-name>

<url-pattern>/toto</url-pattern>

</servlet-mapping>

La balise responsable de la définition du mapping se nomme <servlet-mapping>.Il est tout à fait possible de déclarer plusieurs sections <servlet-mapping> pour une même section <servlet> dans le fichier **web.xml**. Les deux balises obligatoires sont :

* <servlet-name> permet de préciser le nom de la servlet à laquelle faire référence. Cette information doit correspondre avec le nom défini dans la précédente déclaration de la servlet.
* <url-pattern> permet de préciser la ou les URL relatives au travers desquelles la servlet sera accessible. Ici, ce sera **/toto** ! On peut choisir de rendre notre servlet responsable du traitement des requêtes issues d'une seule URL, ou bien d'un groupe d'URL.

**Cycle de vie d'une servlet**

Quand une servlet est demandée pour la première fois ou quand l'application web démarre, le conteneur de servlets va créer une instance de celle-ci et la garder en mémoire pendant toute l'existence de l'application. **La même instance sera réutilisée pour chaque requête entrante** dont les URL correspondent au pattern d'URL défini pour la servlet. La balise <load-on-startup>N</load-on-startup> permet de forcer le serveur à charger l'instance de la servlet en question directement pendant le chargement de l'application plutôt que lors de son premier appel. Le chiffre N correspond à la priorité que l’on souhaite donner au chargement de votre servlet, du plus petit au plus grand.

Exemple de doGet() pour renvoyer une réponse à la requête d’un client (attention, c’est sale de faire du HTML directement dans la servlet. Elle ne doit pas s’occuper de l’affichage, c’est à la vue de le faire !) :

public void doGet( HttpServletRequest request, HttpServletResponse response ) throws ServletException, IOException{

response.setContentType("text/html");

response.setCharacterEncoding( "UTF-8" );

PrintWriter out = response.getWriter();

out.println("<!DOCTYPE html>");

out.println("<html>");

out.println("<head>");

out.println("<meta charset=\"utf-8\" />");

out.println("<title>Test</title>");

out.println("</head>");

out.println("<body>");

out.println("<p>Ceci est une page générée depuis une servlet.</p>");

out.println("</body>");

out.println("</html>");

}

[setContentType()](http://docs.oracle.com/javaee/6/api/javax/servlet/ServletResponse.html#setContentType%28java.lang.String%29) : indique au client qu’on lui renvoi du html.

[setCharacterEncoding()](http://docs.oracle.com/javaee/6/api/javax/servlet/ServletResponse.html#setContentType%28java.lang.String%29) : modifie l’encodage par défaut.

PrintWriter permet d'envoyer du texte au client, via la méthode [getWriter()](http://docs.oracle.com/javaee/6/api/javax/servlet/ServletResponse.html#getWriter%28%29).

**Servlet avec vue**

La technologie utilisée pour réaliser une vue est **la page JSP**. Une telle page peut contenir des balises HTML, mais également des **balises JSP** qui appellent de manière transparente du code Java. Contrairement à une page HTML statique directement renvoyée au client, **une page JSP est exécutée côté serveur**, et génère alors une page renvoyée au client. L'intérêt est de rendre possible la création de pages **dynamiques**. Il est théoriquement possible de générer n'importe quel type de format avec : du HTML, du CSS, du XML, du texte brut, etc.

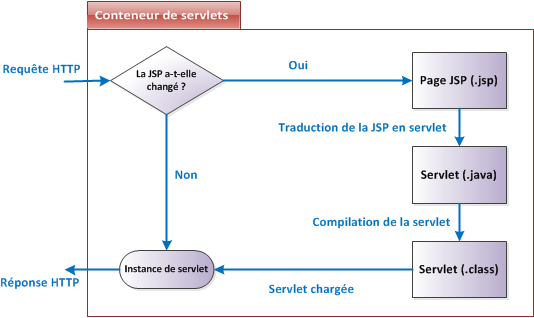
**Nature d'une JSP**

Le langage JSP combine à la fois les technologies HTML, XML, servlet et JavaBeans.

**Cycle de vie d'une JSP**

Quand une JSP est demandée pour la première fois, ou quand l'application web démarre, le conteneur de servlets va **vérifier, traduire puis compiler la page JSP en une classe héritant de HttpServlet**, et l'utiliser durant l'existence de l'application. Cela signifie qu'une JSP est littéralement transformée en servlet par le serveur. La technologie JSP consiste en une véritable abstraction de la technologie servlet.

Voici ce qui se passe au sein du conteneur de servlets lorsqu'une requête HTTP est destinée à une JSP :



Le processus initial de vérification/traduction/compilation n'est pas effectué à chaque appel ! La servlet générée et compilée étant **sauvegardée**, les appels suivants à la JSP sont beaucoup plus rapides : le conteneur se contente d'exécuter directement l'instance de la servlet stockée en mémoire si celle-ci n’a pas été modifiée.

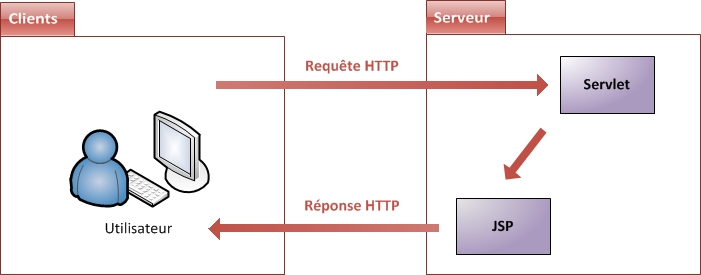
public void doGet( HttpServletRequest request, HttpServletResponse response ) throws ServletException, IOException{

// Permet de faire le lien entre la servlet et la page JSP.

this.getServletContext().getRequestDispatcher( "/WEB-INF/test.jsp" ).forward( request, response );

}

* [getServletContext()](http://docs.oracle.com/javaee/6/api/javax/servlet/ServletConfig.html#getServletContext%28%29)retourne un objet [ServletContext](http://docs.oracle.com/javaee/6/api/javax/servlet/ServletContext.html), qui fait référence au contexte commun à toute l'application : celui-ci contient un ensemble de méthodes qui permettent à une servlet de communiquer avec le conteneur de servlet.
* [getRequestDispatcher()](http://docs.oracle.com/javaee/6/api/javax/servlet/ServletContext.html#getRequestDispatcher%28java.lang.String%29), que nous appliquons à notre page JSP, retourne un objet [RequestDispatcher](http://docs.oracle.com/javaee/6/api/javax/servlet/RequestDispatcher.html), qui agit ici comme une enveloppe autour de notre page JSP. Il est impératif d'y préciser le chemin complet vers la JSP, en commençant obligatoirement par un /. –
* [forward()](http://docs.oracle.com/javaee/6/api/javax/servlet/RequestDispatcher.html#forward%28javax.servlet.ServletRequest,%20javax.servlet.ServletResponse%29)réexpédie la paire requête/réponse HTTP vers la page JSP.



**Transmission de données**

La méthode request.[setAttribute(«test»,message)](http://docs.oracle.com/javaee/6/api/javax/servlet/ServletRequest.html#setAttribute%28java.lang.String,%20java.lang.Object%29)permet d’enregistrer un attribut dans l’objet request.

<%

//Cast nécessaire car getAttribute() renvoi un Object.

String attribut = (String) request.getAttribute("test");

out.println( attribut );

%> Permet de récupérer l’objet message nommé «test» dans la page JSP.

<% %> Permet d’inclure du code Java dans une page JSP. On essaye cependant de mettre le moins possible de code Java dans une JSP. Ce n’est pas une bonne pratique de le faire.

Les paramètres ou *query strings* permettent au client e transmettre des données au serveur en les incluant directement dans l'URL.

<!-- URL avec deux paramètres nommés 'lang' et 'admin', et ayant pour valeur respectivement 'fr' et 'true' -->

/page.jsp?lang=fr&admin=true

Récupération d’un paramètre par le serveur :

String paramAuteur = request.getParameter( "auteur" );

**Les paramètres de requête** sont un concept appartenant au protocole HTTP. Ils sont envoyés par le client au serveur directement au sein de l'URL, il est uniquement possible d'y accéder en lecture. Ce sont TOUJOURS des Strings.

**Les attributs de requête** sont un concept appartenant au conteneur Java, et sont donc créés côté serveur. C'est au sein du code de l'application que l'on procède à leur initialisation, et qu'on les insère dans la version "objectifiée" de la requête, à savoir l'objet HttpServletRequest.

**Le JavaBean**

Souvent raccourci en "bean", un JavaBean désigne tout simplement un composant réutilisable qui suit certaines contraintes.

* **Les propriétés** : un bean est conçu pour être **paramétrable**. On appelle "propriétés" les champs non publics présents dans un bean. Elles permettent de paramétrer le bean, en y stockant des données.
* **La sérialisation** : un bean est conçu pour pouvoir être **persistant**. La sérialisation est un processus qui permet de sauvegarder l'état d'un bean, et donne ainsi la possibilité de le restaurer par la suite.
* **La réutilisation :** un bean est un composant conçu pour être réutilisable. Ne contenant que des données ou du code métier, un tel composant n'a en effet pas de lien direct avec la couche de présentation, et peut également être distant de la couche d'accès aux données**.**
* **L'introspection** : un bean est conçu pour être **paramétrable de manière dynamique**. L'introspection est un processus qui permet de connaître le contenu d'un composant (attributs, méthodes et événements) de manière dynamique, sans disposer de son code source.

**Attention :** Un JavaBean n'est pas un EJB, les EJB suivent un concept complètement différent.

### ****Structure****

Un bean :

* doit être une **classe publique**.
* doit avoir au moins **un constructeur par défaut, public et sans paramètres**. Java l'ajoutera de lui-même si aucun constructeur n'est explicité.
* peut implémenter l'interface Serializable, il devient ainsi persistant et son état peut être sauvegardé.
* **ne doit pas avoir de champs publics**.
* peut définir **des propriétés (des champs non publics), qui doivent être accessibles via des méthodes publiques *getter* et *setter***, suivant des **règles de nommage**.

/\* Cet objet est une classe publique \*/

public class MonBean{

/\* Cet objet ne possède aucun constructeur, Java lui assigne donc un constructeur par défaut public et sans paramètre. \*/

/\* Les champs de l'objet ne sont pas publics (ce sont donc des propriétés) \*/

private String proprieteNumero1;

/\* Les propriétés de l'objet sont accessibles via des getters et setters publics \*/

public String getProprieteNumero1() {

return this.proprieteNumero1;

}

public void setProprieteNumero1( String proprieteNumero1 ) {

this.proprieteNumero1 = proprieteNumero1;

}

/\* Cet objet suit donc bien la structure énoncée : c'est un bean ! \*/

}

### ****Configuration du projet sous Eclipse****

### ****Afin de rendre** les objets accessibles à l’application, il faut que les classes compilées à partir des fichiers sources soient placées dans un dossier "classes", lui-même placé sous le répertoire **/WEB-INF**. Il faut donc penser à configurer le BuildPath lorsque l’on crée une application.**

**La technologie JSP:**

### ****Les balises****

**Balises de commentaire :** <%-- --%>

**Balises de déclaration : Permet de déclarer une variable à l'intérieur d'une JSP (déconseillé)**

<%! String chaine = "Salut les zéros."; %>

**Balises de scriptlet : Elle sert en effet à inclure du code Java au sein de vos pages (déconseillé)**

<% out.println("toto"); %>

Balises d’expression : Elle retourne simplement le contenu d'une chaîne. <%= "Bip bip !" %> qui correspond à <%= "Bip bip !" %>.

### ****Les directives****

Les directives JSP permettent :

* d'importer un package.
* d'inclure d'autres pages JSP.
* d'inclure des bibliothèques de balises.
* de définir des propriétés et informations relatives à une page JSP.

Il en existe trois : **taglib**, **page** et **include**.   
Elles sont toujours comprises entre les balises <%@ %> , et hormis la directive d'inclusion de page qui peut être placée n'importe où, elles sont à placer en tête de page JSP.

Directive taglib : Inclut une bibliothèque personnalisée nommée maTagLib

<%@ taglib uri="maTagLib.tld" prefix="tagExemple" %>

Directive page : Définit des informations relatives à la page JSP. Voici par exemple comment importer des classes Java <%@ page import="java.util.List, java.util.Date" %>

Voici à titre d'information l'ensemble des propriétés accessibles via cette directive :

<%@ page

language="..."

extends="..."

import="..."

session="true | false"

buffer="none | 8kb | sizekb"

autoFlush="true | false"

isThreadSafe="true | false"

isELIgnored ="true | false"

info="..."

errorPage="..."

contentType="..."

pageEncoding="..."

isErrorPage="true | false"

%>

Directive include : Inclut le contenu d'un autre fichier dans le fichier courant. Par exemple l’insertion d’un menu dans toutes les pages d’un site. <%@ include file="uneAutreJSP.jsp" %>. Cette directive ne doit être utilisée que pour inclure du contenu "statique". Statique signifie que **l'inclusion est réalisée au moment de la compilation** ; par conséquent, si le code du fichier est changé par la suite, les répercussions sur la page l'incluant n'auront lieu qu'après une nouvelle compilation. Mais il peut inclure du code dynamique.

Action standard include : Permet d'inclure du contenu de manière "dynamique". Le contenu sera ici chargé à l'exécution, et non à la compilation comme c'est le cas avec la directive précédente

<%-- L'inclusion dynamique d'une page fonctionne par URL relative : --%>

<jsp:include page="page.jsp" />

<%-- Son équivalent en code Java est : --%>

<% request.getRequestDispatcher( "page.jsp" ).include( request, response ); %>

<%-- Et il est impossible d'inclure une page externe comme ci-dessous : --%>

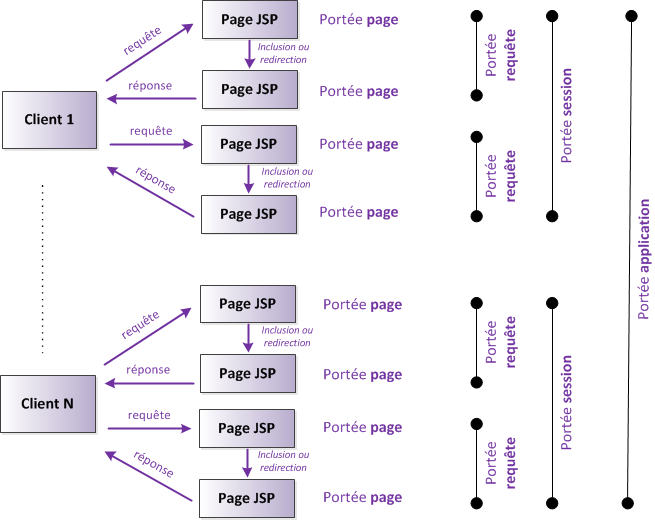
<jsp:include page="http://www.siteduzero.com" />

Ce type d'inclusion a un autre inconvénient : il ne prend pas en compte les imports et inclusions faits dans la page réceptrice. Si par exemple on utilise un type List dans une première page, et que l’on compte utiliser une liste dans la seconde page que l’on veut inclure dans cette première page, il faudra importer le type List dans cette seconde page. Pour faire simple, les pages incluses via la balise <jsp:include ... /> doivent en quelque sorte être "indépendantes". Elles ne peuvent pas dépendre les unes des autres et doivent pouvoir être compilées séparément. Ce n'est pas le cas des pages incluses via la directive <%@ include ... %>.

### ****La portée des objets****

Un concept important intervient dans la gestion des objets par la technologie JSP : **la portée des objets**. Souvent appelée visibilité, ou scope en anglais, elle définit tout simplement leur **durée de vie**. Il existe au total quatre portées différentes dans une application :

* **page** (JSP seulement) : les objets dans cette portée sont uniquement accessibles dans la page JSP en question. Il n'est possible de créer et manipuler des objets de portée page que depuis une page JSP, ce n'est pas possible via une servlet.
* **requête** : les objets dans cette portée sont uniquement accessibles durant l'existence de la requête en cours.
* **session** : les objets dans cette portée sont accessibles durant l'existence de la session en cours. **Une session est un objet associé à un utilisateur** en particulier. Elle existe pour la durée pendant laquelle un visiteur va utiliser l'application, cette durée se terminant lorsque l'utilisateur ferme son navigateur, reste inactif trop longtemps, ou encore lorsqu'il se déconnecte du site. Une session correspond en réalité à un navigateur particulier, plutôt qu'à un utilisateur.
* **application** : les objets dans cette portée sont accessibles durant toute l'existence de l'application.



### Les actions standard (ancienne façon de faire du JSP) :

**L'action standard useBean :** Permet d'utiliser un bean, ou de le créer s'il n'existe pas, depuis une page JSP.

<%-- L'action suivante récupère un bean de type Coyote et nommé "coyote" dans la portée requête s'il existe, ou en crée un sinon. --%>

<jsp:useBean id="coyote" class="com.sdzee.beans.Coyote" scope="request" />

<%-- Elle a le même effet que le code Java suivant : --%>

<%

com.sdzee.beans.Coyote coyote = (com.sdzee.beans.Coyote) request.getAttribute( "coyote" );

if ( coyote == null ){

coyote = new com.sdzee.beans.Coyote();

request.setAttribute( "coyote", coyote );

}

%>

Attributs :

* **id** = nom du bean à récupérer/créer.
* **class = classe du bean.** Obligatoire si on souhaite créer un bean, pas si on récupère un bean existant.
* **scope = portée de l’objet.** Si un bean du nom spécifié en id **existe déjà** dans ce scope, et qu'il est du type ou de la classe précisé(e), alors il est récupéré, sinon une erreur survient. Si aucun bean de ce nom n'existe dans ce scope, alors un nouveau bean est créé. Le scope par défaut est page.
* **type (optionnel) = indique l**e type de déclaration du bean. Il doit être une superclasse de la classe du bean, ou une interface implémentée par le bean. Cet attribut doit être spécifié si **class** ne l'est pas, et vice-versa.

Il est également possible de donner un corps à cette balise, qui ne sera exécuté que si le bean est créé. Si un bean du même nom existe déjà dans cette page, alors le bean sera simplement récupéré et le texte ne sera pas affiché. :

<jsp:useBean id="coyote" class="com.sdzee.beans.Coyote">

<%-- Ici, vous pouvez placer ce que vous voulez :

définir des propriétés, créer d'autres objets, etc. --%>

<p>Nouveau bean !</p>

</jsp:useBean>

L'action standard getProperty : Lorsque l'on utilise un bean au sein d'une page, il est possible par le biais de cette action d'obtenir la valeur d'une de ses propriétés.

<jsp:useBean id="coyote" class="com.sdzee.beans.Coyote" />

<%-- L'action suivante affiche le contenu de la propriété 'prenom' du bean 'coyote' : --%>

<jsp:getProperty name="coyote" property="prenom" />

<%-- Elle a le même effet que le code Java suivant : --%>

<%= coyote.getPrenom() %>

Attention : Alors que <jsp:useBean> récupère une instance dans une variable accessible par l'id défini, <jsp:getProperty> ne récupère rien, mais réalise seulement l'affichage du contenu de la propriété ciblée.

* **name** = contient le nom réel du bean, en l'occurrence l'id que l'on a saisi auparavant dans la balise de récupération du bean.
* **property =** contient le nom de la propriété dont on souhaite afficher le contenu.

L'action standard setProperty : Permet de modifier une propriété du bean utilisé. Il existe pour cela quatre façons de faire.

Méthode 1 :

<%-- L'action suivante associe une valeur à la propriété 'prenom' du bean 'coyote' : --%>

<jsp:setProperty name="coyote" property="prenom" value="Wile E." />

<%-- Elle a le même effet que le code Java suivant : --%>

<% coyote.setPrenom("Wile E."); %>

Méthode 2 :

<%-- L'action suivante associe directement la valeur récupérée

depuis le paramètre de la requête nommé ici 'prenomCoyote' à la propriété 'prenom' : --%>

<jsp:setProperty name="coyote" property="prenom" param="prenomCoyote"/>

<%-- Elle a le même effet que le code Java suivant : --%>

<% coyote.setPrenom( request.getParameter("prenomCoyote") ); %>

Méthode 3 :

<%-- L'action suivante associe directement la valeur récupérée

depuis le paramètre de la requête nommé ici 'prenom' à la propriété de même nom : --%>

<jsp:setProperty name="coyote" property="prenom" />

<%-- Elle a le même effet que le code Java suivant : --%>

<% coyote.setPrenom( request.getParameter("prenom") ); %>

Méthode 4 :

<%-- L'action suivante associe automatiquement la valeur récupérée

depuis chaque paramètre de la requête à la propriété de même nom : --%>

<jsp:setProperty name="coyote" property="\*" />

<%-- Elle a le même effet que le code Java suivant : --%>

<% coyote.setNom( request.getParameter("nom") ); %>

<% coyote.setPrenom( request.getParameter("prenom") ); %>

<% coyote.setGenius( Boolean.valueOf( request.getParameter("genius") ) ); %>

L'action standard forward : Permet d'effectuer une redirection vers une autre page. Elle s'effectue côté serveur et pour cette raison **il est impossible via cette balise de rediriger vers une page extérieure à l'application**. L'action de forwarding est ainsi limitée aux pages présentes dans le contexte de la servlet ou de la JSP utilisée. Il n'implique pas d'aller/retour passant par le navigateur de l'utilisateur final. Autrement dit, l'utilisateur final n'est pas au courant que sa requête a été redirigée vers une ou plusieurs JSP différentes, puisque l'URL qui est affichée dans son navigateur ne change pas.

<%-- Le forwarding vers une page de l'application fonctionne par URL relative : --%>

<jsp:forward page="/page.jsp" />

<%-- Son équivalent en code Java est : --%>

<% request.getRequestDispatcher( "/page.jsp" ).forward( request, response ); %>

<%-- Et il est impossible de rediriger vers un site externe comme ci-dessous : --%>

<jsp:forward page="http://www.siteduzero.com" />

Attention : lorsqu’on utilise le forwarding, **le code présent après cette balise dans la page n'est pas exécuté.**

**La technologie JSP (façon actuelle de faire) :**

### ****Expression Language ou EL****

Elles permettent de s’affranchir de l’écriture de scriptlets. Forme et la syntaxe d'une telle expression : ${ expression }. C'est que **ce qui est situé entre les accolades va être interprété.**

### ****La réalisation de tests****

Pour réaliser ces tests, il vous est possible d'inclure toute une série d'opérateurs : +, -, \*, /, %, &&, ||, !, equals(), compareTo(), == ou eq, != ou ne, < ou lt, > ou gt, <= ou le, >= ou ge.

Ex : ${ 'a' < 'b' } affiche true, ${ 6 \* 7 == 48 } affiche false, ${ 6 \* 7 } affiche 42,

${ 7/0 } affiche Infinity.

Dans les EL on peut utiliser des simples quotes ‘ pour déclarer des Strings.

<!-- Vérifications si vide ou null -->

${ empty 'test' } <!-- La chaîne testée n'est pas vide, le résultat est false -->

**<!-- Conditions ternaires -->**

${ 'a' > 'b' ? 'oui' : 'non' } <!-- Le résultat de la comparaison vaut false, non est affiché -->

La valeur retournée par une expression EL positionnée dans un texte ou un contenu statique sera insérée à l'endroit même où est située l'expression :

<!-- La ligne suivante : -->

<p>12 est inférieur à 8 : ${ 12 lt 8 }.</p>

<!-- Sera rendue ainsi après interprétation de l'expression, 12 n'étant pas inférieur à 8 : -->

<p>12 est inférieur à 8 : false.</p>

**La manipulation d'objets par les EL**

### ****Des beans****

Il est possible via une expression EL d'accéder directement à une propriété d'un bean.

${ coyote.prenom } : **coyote** est le nom du bean, **prenom** est un champ privé du bean accessible par sa méthode publique getPrenom(),l’opérateur point permet de séparer le bean visé de sa propriété.

Correspond à :

Coyote bean = (Coyote) pageContext.findAttribute( "coyote" );

if ( bean != null ) {

String prenom = bean.getPrenom();

if ( prenom != null ) {

out.print( prenom );

}

}

Les EL permettent aussi de tester des propriétés :

${ !empty coyote.prenom ? coyote.prenom : "Veuillez préciser un prénom" }

**Les expressions EL sont protégées contre un éventuel retour null :**

<!-- L'expression EL suivante n'affiche rien si la propriété "prenom" n'a pas été initialisée,

et n'affiche rien si l'objet "coyote" n'a pas été initialisé : -->

${ coyote.prenom }

**Des collections**

Il est possible d'accéder aux collections au sens large dans une EL.

<!-- Les quatre syntaxes suivantes retournent le deuxième élément de la liste de légumes -->

${ legumes.get(1) }

${ legumes[1] }

${ legumes['1'] }

${ legumes["1"] }

<!-- Les trois syntaxes suivantes retournent le troisième élément du tableau animaux -->

${ animaux[2] }

${ animaux['2'] }

${ animaux["2"] }

Attention : il est impossible d'utiliser directement l'opérateur point pour accéder à un élément d'une liste ou d'un tableau. Les syntaxes ${entiers.1} ou ${animaux.2} enverront une exception à l'exécution.

<!-- Les quatre syntaxes suivantes retournent la valeur associée à la clé "cookies" de la Map de desserts -->

${ desserts.cookies }

${ desserts.get("cookies") }

${ desserts['cookies'] }

${ desserts["cookies"] }

<!-- Il est également possible d'utiliser un objet au lieu d'initialiser la clé souhaitée directement dans l'expression. Ici ‘element’ est un String qui vaut ‘cookies’ -->

${ desserts[element] }

Attention à ne pas écrire ${desserts[«element»]} sinon le code essayera d’accéder à la clé « element » de la liste, et pas à l’objet element.

Une pratique efficace veut que l'on réserve la notation avec l'opérateur point pour l'accès aux propriétés d'un bean, et que l'on utilise les crochets pour accéder aux contenus d'une Map. Ainsi en lisant le code d'une page JSP, il devient très simple de savoir si l'objet manipulé dans une expression EL est un bean ou une Map.

**Désactiver l'évaluation des expressions EL**

Utile pour les vieilles versions de JSP où les EL ne seraient pas comprises :

Au cas par cas pour désactiver les EL dans une page JSP : <%@ page isELIgnored ="true" %>

Sur tout ou partie des pages via web.xml : Ceci se fait grâce à <el-ignored>.

<jsp-config>

<jsp-property-group>

<url-pattern>\*.jsp</url-pattern>

<el-ignored>true</el-ignored>

</jsp-property-group>

</jsp-config>

Les expressions EL sont supportées par les versions 2.4 et supérieur de l’API Servlet. Il faut aussi penser à inclure la balise web-app dans le fichier web.xml :

<web-app

xmlns="http://java.sun.com/xml/ns/javaee"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd"

version="3.0">

**Les objets implicites**

Il en existe deux types :

* ceux qui sont mis à disposition via la technologie JSP .
* ceux qui sont mis à disposition via la technologie EL.

**Les objets de la technologie JSP**

Ils permettent de ne pas avoir besoin d'instancier ou de récupérer les objets **out** et **request** avant de les utiliser dans le code d’une JSP. Dans les coulisses, **le conteneur s'en charge seul lorsqu'il traduit la page en servlet** ! C'est pour cette raison que ces objets sont dits "implicites". Il existe une liste d'objets implicites accessibles directement depuis les pages JSP :

|  |  |  |
| --- | --- | --- |
| **Identifiant** | **Type de l'objet** | **Description** |
| **pageContext** | PageContext | Il fournit des informations utiles relatives au contexte d'exécution. Entre autres, il permet d'accéder aux attributs présents dans les différentes portées de l'application. Il contient également une référence vers tous les objets implicites suivants. |
| **application** | ServletContext | Il permet depuis une page JSP d'obtenir ou de modifier des informations relatives à l'application dans laquelle elle est exécutée. |
| **session** | HttpSession | Il représente une session associée à un client. Il est utilisé pour lire ou placer des objets dans la session de l'utilisateur courant. |
| **request** | HttpServletRequest | Il représente la requête faite par le client. Il est généralement utilisé pour accéder aux paramètres et aux attributs de la requête, ainsi qu'à ses en-têtes. |
| **response** | HttpServletResponse | Il représente la réponse qui va être envoyée au client. Il est généralement utilisé pour définir le Content-Type de la réponse, lui ajouter des en-têtes ou encore pour rediriger le client. |
| **exception** | Throwable | Il est uniquement disponible dans les pages d'erreur JSP. Il représente l'exception qui a conduit à la page d'erreur en question. |
| **out** | JspWriter | Il représente le contenu de la réponse qui va être envoyée au client. Il est utilisé pour écrire dans le corps de la réponse. |
| **config** | ServletConfig | Il permet depuis une page JSP d'obtenir les éventuels paramètres d'initialisation disponibles. |
| **page** | objet this | Il est l'équivalent de la référence this et représente la page JSP courante. Il est déconseillé de l'utiliser, pour des raisons de dégradation des performances notamment. |

Il est possible d'utiliser n'importe lequel de ces neuf objets à travers le code Java écrit dans des pages JSP

**Les objets implicites de la technologie EL**

Ils permettent de **pouvoir profiter des objets implicites sans écrire de code Java** ! Il s'agit d'objets gérés automatiquement par le conteneur lors de l'évaluation des expressions EL, et auxquels nous pouvons directement accéder depuis nos expressions sans les déclarer auparavant.

|  |  |  |
| --- | --- | --- |
| **Catégorie** | **Identifiant** | **Description** |
| JSP | **pageContext** | Objet contenant des informations sur l'environnement du serveur. |
| Portées | **pageScope** | Une Map qui associe les noms et valeurs des attributs ayant pour portée la page. |
|  | **requestScope** | Une Map  qui associe les noms et valeurs des attributs ayant pour portée la requête. |
|  | **sessionScope** | Une Map  qui associe les noms et valeurs des attributs ayant pour portée la session. |
|  | **applicationScope** | Une Map  qui associe les noms et valeurs des attributs ayant pour portée l'application. |
| Paramètres de requête | **param** | Une Map qui associe les noms et valeurs des paramètres de la requête. |
|  | **paramValues** | Une Map  qui associe les noms et multiples valeurs \*\* des paramètres de la requête sous forme de tableaux de String. |
| En-têtes de requête | **header** | Une Map qui associe les noms et valeurs des paramètres des en-têtes HTTP. |
|  | **headerValues** | Une Map  qui associe les noms et multiples valeurs \*\* des paramètres des en-têtes HTTP sous forme de tableaux de String. |
| Cookies | **cookie** | Une Map qui associe les noms et instances des cookies. |
| Paramètres d’initialisation | **initParam** | Une Map qui associe les données contenues dans les champs <param-name> et <param-value> de la section <init-param> du fichier web.xml. |

<p>

Langue : ${ param.langue } (= fr)

<br />

Article : ${ param.article } (= 782)

</p>

Permet de récupérer les paramètres « langue » et « article » de la requête http://localhost:8080/test/test\_obj\_impl.jsp?langue=fr&article=782 grâce à l’objet implicite *param*. *paramValues* permettrait de récupérer plusieurs valeurs pour un seul paramètre (par exemple http://localhost:8080/test/test\_obj\_impl.jsp?langue=fr&article=782&langue=zh ). On écrira alors par exemple ${ paramValues.langue[1] } (= zh)

L'unique différence entre les objets implicites **param** et **paramValues**, ainsi qu'entre **header** et **headerValues**, se situe au niveau de la nature de l'objet utilisé dans la Map et des valeurs qui y sont stockées :

* Pour **param** et **header**, une seule valeur est associée à chaque nom de paramètre, via une Map<String,String> ;
* Pour **paramValues** et **headerValues** par contre, ce sont plusieurs valeurs qui vont être associées à un même nom de paramètre, via une Map<String,String[]>.

**Problèmes de vue**

<p>

${test} Renvoi l’attribut de requête test (définit par le code).

${param.auteur} Renvoi le paramètre de requête auteur (défini dans l’url).

</p>

<p>

Récupération du bean :

${coyote.prenom} Renvoi la propriété prenom du bean coyote

${coyote.nom} Renvoi la propriété nom du bean coyote

</p>

Dans une expression EL, lorsqu’on accède à un objet présent dans une des quatre portées de l’application, on n’a pas besoin de spécifier l'objet implicite (c'est-à-dire la portée) auquel on souhaite accéder (par exemple ${request.coyote.prenom}). Une expression est capable de réaliser d'elle-même un parcours automatique des différentes portées accessibles à la recherche d'un objet portant le nom précisé, de la plus petite à la plus grande portée. La bonne pratique de développement est de ne jamais donner le même nom à des objets existant dans des portées différentes ! Pour éviter ce parcours et cibler directement la portée voulue, il faut utiliser les objets implicites fournis par la technologie EL donnant accès aux attributs existants : **pageScope**, **requestScope**, **sessionScope** et **applicationScope**. Ex : ${requestScope.test}

La bonne pratique veut qu'en complément de la rigueur dans le nommage des objets conseillée précédemment, le développeur précise toujours la portée qu'il souhaite cibler dans une expression EL.

Pour cibler un objet qui n'est pas présent dans une des quatre portées, il est nécessaire d'expliciter l'objet implicite qui permet d'y accéder au sein de l'expression EL. Voilà pourquoi pour accéder au paramètre de requête **auteur**, nous devons bien préciser ${param.auteur}.

**Manipulation d'une liste ou utilisation d’une condition**

Pour cela on utilise la JSTL : une bibliothèque de balises préconçues qui permettent, à la manière des balises JSP, de mettre en place les fonctionnalités dont on a couramment besoin dans une vue, mais qui ne sont pas accessibles nativement au sein de la technologie JSP.

**Objectifs et configuration**

La JSTL est une bibliothèque, une collection regroupant des balises implémentant des fonctionnalités à des fins générales, communes aux applications web. Son objectif est de permettre au développeur d'éviter l'utilisation de code Java dans les pages JSP, et ainsi de respecter au mieux le découpage en couches recommandé par le modèle MVC. Les balises JSTL évitent l'utilisation de code Java dans les pages JSP qui est déconseillé. Le code devient plus lisible.

**Une boucle avec une scriptlet Java**

<%@ page import="java.util.List, java.util.ArrayList" %>

<%

List<Integer> list = (ArrayList<Integer>)request.getAttribute("tirage");

for(int i = 0; i < list.size();i++){

out.println(list.get(i));

}

%>

**La même boucle avec des tags JSTL**

<c:forEach var="item" items="${tirage}" >

<c:out value="${item}" />

</c:forEach>

**Versions de la JSTL**

* JSTL 1.0 pour la plate-forme J2EE 3, et un conteneur JSP 1.2 (ex: Tomcat 4) ;
* JSTL 1.1 pour la plate-forme J2EE 4, et un conteneur JSP 2.0 (ex: Tomcat 5.5) ;
* JSTL 1.2, qui est partie intégrante de la plate-forme Java EE 6, avec un conteneur JSP 2.1 ou 3.0 (ex: Tomcat 6 et 7).

Les différences entre ces versions résident principalement dans le conteneur JSP nécessaire. Le changement majeur à retenir dans le passage de la première version à la seconde version de ce conteneur, c'est la gestion de la technologie EL. Le conteneur JSP 1.2 sur lequel est basée la JSTL 1.0 ne gérait pas les expressions EL, cette dernière proposait donc deux implémentations pour pallier ce manque : une les interprétant et l'autre non. La version 1.1 est basée sur le conteneur JSP 2.0, qui intègre nativement un interpréteur d'expressions EL. La version actuelle, à savoir la JSTL 1.2, apporte des EL "unifiées", ainsi qu'une meilleure intégration dans le framework JSF.

**Configuration de la JSTL**

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %> permet d’inclure une bibliothèque de balises JSTL dans une page JSP.

* Dans le paramètre **uri** se trouve le lien vers la définition de la bibliothèque. Le lien **/jstl/core** est celui de la version antérieure 1.0 de la JSTL qui ne gère pas les EL. **/jsp/jstl/core** est le lien actuel.
* Dans le paramètre **prefix** se trouve l'alias qui sera utilisé dans notre page JSP pour faire appel aux balises de la bibliothèque en question.

Si cette bibliothèque n’est pas reconnue, c’est que le serveur utilisé n’est pas à jour concernant les spécifications Java EE (c’est le cas avec Tomcat). Pour pallier à cela, il suffit d’ajouter la bibliothèque en question dans le dossier /WEB-INF/lib de notre projet (ici le jar **jstl-1.2.jar** pour utiliser la bibliothèque Core). Rem : si Eclipse continu à afficher un warning suite à cela, il suffit d’ajouter une modification à notre page JSP puis à la sauvegarder pour que la nouvelle bibliothèque soit prise en compte et que le warning disparaisse.

**La bibliothèque Core**

**Les variables et expressions**

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

**Directive JSP nécessaire** pour permettre l'utilisation des balises de la bibliothèque Core dans les pages JSTL.

**Affichage d'une expression**

Le seul attribut obligatoirement requis pour ce tag est **value**.

<c:out value="test" /> <%-- Affiche test --%>

<c:out value="${ 'a' < 'b' }" /> <%-- Affiche true --%>

À celui-ci s'ajoutent deux attributs optionnels :

* **default** : permet de définir une valeur affichée par défaut si le contenu de l'expression évaluée est vide.
* **escapeXml** : permet de remplacer les caractères de scripts < , > , " , ' et & par leurs équivalents en code html &lt;, &gt;, &#034;, &#039;, &amp;. Cette option est activée par défaut, et il faut expliciter <c:out ... escapeXml="false" /> pour la désactiver.

Ex :

<c:out value="<p>Je suis un 'paragraphe'.</p>" />

<%-- La balise affichera : --%>

&lt;p&gt;Je suis un &#039;paragraphe&#039;.&lt;/p&gt;

<c:out value="<p>Je suis un 'paragraphe'.</p>" escapeXml="false" />

<%-- La balise affichera : --%>

<p>Je suis un 'paragraphe'.</p>

Cela permet d’éviter les failles de sécurité XSS et l’injection de code lorsque l’on récupère des données entrées par l’utilisateur.

**Création**

<%-- Cette balise met l'expression "Salut les zéros !" dans l'attribut "message" de la requête : --%>

<c:set var="message" value="Salut les zéros !" scope="request" />

<%-- Et est l'équivalent du scriplet Java suivant : --%>

<% request.setAttribute( "message", "Salut les zéros !" ); %>

L'attribut **var** contient le nom de la variable que l'on veut stocker, **value** sa valeur, et **scope** la portée de cette variable.

<%-- Affiche l'expression contenue dans la variable "message" de la requête --%>

<c:out value="${requestScope.message}" />

**Modification**

La modification d'une variable s'effectue de la même manière que sa création.

<%-- L'attribut scope n'est pas obligatoire. Rappelez-vous, le scope par défaut est dans ce cas la page,

puisque c'est le premier dans la liste des scopes parcourus --%>

<c:set var="maVariable" value="12" />

Il est également possible d'initialiser une variable en utilisant le corps de la balise, plutôt qu'en utilisant l'attribut **value** : <c:set var="maVariable"> 12 </c:set>

Voici par exemple comment vous initialiser la valeur d'une variable de session depuis une valeur lue dans un paramètre de l'URL :

<c:set var="locale" scope="session">

<c:out value="${param.lang}" default="FR"/>

</c:set>

**Modification des propriétés d'un objet**

<%-- Crée un objet du type du bean ici spécifié dans l'attribut 'value'--%>

<c:set scope="session" var="tonBean" value="${monBean}" />

<!-- Définir ou modifier la propriété 'prenom' du bean 'coyote' -->

<c:set target="${coyote}" property="prenom" value="Wile E."/>

<!-- Définir ou modifier la propriété 'prenom' du bean 'coyote' via le corps de la balise -->

<c:set target="${coyote}" property="prenom">

Wile E.

</c:set>

<!-- Passer à null la valeur de la propriété 'prenom' du bean 'coyote' -->

<c:set target="${coyote}" property="prenom" value="${null}" />

* **target** : contient le nom de l'objet dont la propriété sera modifiée.
* **property** : contient le nom de la propriété qui sera modifiée.

**Suppression**

Un seul attribut est requis **var**. Par défaut toujours, c'est le scope page qui sera parcouru si l'attribut **scope** n'est pas explicité :

<%-- Supprime la variable "maVariable" de la session --%>

<c:remove var="maVariable" scope="session" />

**Les conditions**

**Une condition simple**

Permet de tester une seule expression, et correspond au bloc if() du langage Java. Le seul attribut obligatoire est **test**.

<c:if test="${ 12 > 7 }" var="maVariable" scope="session">

Ce test est vrai.

</c:if>

Le corps de la balise ne sera affiché dans la page finale que si la condition est vraie. Avec  **var** et **scope** le résultat du test conditionnel sera stocké dans la variable et dans le scope défini, et sinon dans le scope page par défaut.

**Des conditions multiples**

Cela correspond à une suite de if() / else if() ou au bloc switch().

<c:choose>

<c:when test="${expression}">Action ou texte.</c:when>

...

<c:otherwise>Autre action ou texte.</c:otherwise>

</c:choose>

La balise <c:choose> ne peut contenir aucun attribut, et son corps ne peut contenir qu'une ou plusieurs balises <c:when> et une ou zéro balise <c:otherwise>. <c:when> doit obligatoirement se voir définir un attribut **test** contenant la condition. À l'intérieur d'un même bloc <c:choose>, un seul <c:when> verra son corps évalué, les conditions étant mutuellement exclusives. <c:otherwise> ne peut contenir aucun attribut, et son corps ne sera évalué que si aucune des conditions la précédant dans le bloc n'est vérifiée.

**Les boucles**

Il existe <c:forEach> pour parcourir une collection, et <c:forTokens> pour parcourir une chaîne de caractères.

**Boucle "classique"**

<%-- Boucle calculant le cube des entiers de 0 à 7 et les affichant dans un tableau HTML --%>

<table>

<tr>

<th>Valeur</th>

<th>Cube</th>

</tr>

<c:forEach var="i" begin="0" end="7" step="1">

<tr>

<td><c:out value="${i}"/></td>

<td><c:out value="${i \* i \* i}"/></td>

</tr>

</c:forEach>

</table>

* **begin** : la valeur de début de notre compteur.
* **end** : la valeur de fin de notre compteur. Correspond ici à i <= 7.
* **step** : c'est le pas d'incrémentation de la boucle. Par défaut la valeur 1 sera utilisée.
* **var** : cet attribut est, contrairement à ce qu'on pourrait croire a priori, non obligatoire. Si on ne le spécifie pas, on ne pourra simplement pas accéder à la valeur du compteur en cours. Par ailleurs, tout comme en Java, la variable n'est accessible qu'à l'intérieur de la boucle, autrement dit dans le corps de la balise <c:forEach>.

**Itération sur une collection**

La syntaxe utilisée pour parcourir une collection est similaire à celle d'une boucle simple, sauf que cette fois, un attribut **items** est requis.

<%@ page import="java.util.\*" %>

<%

/\* Création de la liste et des données \*/

List<Map<String, String>> maListe = new ArrayList<Map<String, String>>();

Map<String, String> news = new HashMap<String, String>();

news.put("titre", "Titre de ma première news");

news.put("contenu", "corps de ma première news");

maListe.add(news);

news = new HashMap<String, String>();

news.put("titre", "Titre de ma seconde news");

news.put("contenu", "corps de ma seconde news");

maListe.add(news);

pageContext.setAttribute("maListe", maListe);

%>

<c:forEach items="${maListe}" var="news">

<div class="news">

<div class="titreNews">

<c:out value="${news['titre']}" />

</div>

<div class="corpsNews">

<c:out value="${news['contenu']}" />

</div>

</div>

</c:forEach>

Il est également possible d’utiliser à la place des crochets l'opérateur point : ${news.titre} et ${news.contenu}.

* **items :** indique la collection sur laquelle la boucle porte, en l'occurrence la List nommée maListe.
* **var :** indique le nom de la variable qui sera liée à l’élément courant de la collection parcourue par la boucle.

Les attributs présentés précédemment lors de l'étude d'une boucle simple sont là aussi valables.

Ex :

<c:forEach items="${maListe}" var="news" begin="0" end="9">

...

</c:forEach>

Si les attributs **begin** et **end** spécifiés dépassent le contenu réel de la collection, la boucle s'arrêtera automatiquement lorsque le parcours de la liste sera terminé, peu importe l'indice **end** spécifié. Enfin, il est également possible d'itérer directement sur le résultat d'une requête SQL.

**L'attribut varStatus**

C’est un attribut utilisable pour tout type d'itérations. Comme **var,** il est utilisé pour créer une variable de scope, mais présente une différence majeure : alors que **var** permet de stocker la valeur de l'index courant ou l'élément courant de la collection parcourue, le **varStatus** permet de stocker un objet **LoopTagStatus** qui définit un ensemble de propriétés définissant l'état courant d'une itération. L'objet créé par cet attribut **varStatus** n'est visible que dans le corps de la boucle, tout comme l'attribut **var** :

|  |  |
| --- | --- |
| **Propriété** | **Description** |
| **begin** | La valeur de l'attribut begin. |
| **end** | La valeur de l'attribut end. |
| **step** | La valeur de l'attribut step. |
| **first** | Booléen précisant si l'itération courante est la première. |
| **last** | Booléen précisant si l'itération courante est la dernière. |
| **count** | Compteur d'itérations (commence à 1). |
| **index** | Index d'itérations (commence à 0). |
| **current** | Élément courant de la collection parcourue. |

<c:forEach items="${maListe}" var="news" varStatus="status">

<div class="news">

News n° <c:out value="${status.count}"/> :

<div class="titreNews">

<c:out value="${news['titre']}" />

</div>

<div class="corpsNews">

<c:out value="${news['contenu']}" />

</div>

</div>

</c:forEach>

**Itération sur une chaîne de caractères**

Une variante de la boucle <c:forEach> existe, spécialement dédiée aux chaînes de caractères.

<p>

<%-- Affiche les différentes sous-chaînes séparées par une virgule ou un point-virgule --%>

<c:forTokens var="sousChaine" items="salut; je suis un,gros;zéro+!" delims=";,+">

${sousChaine}<br/>

</c:forTokens>

</p>

* **delims** : Permet de spécifier quels sont les caractères qui serviront de séparateurs dans la chaîne que la boucle parcourra. Il suffit de les spécifier les uns à la suite des autres, comme c'est le cas ici dans notre exemple.

**Ce que la JSTL ne permet pas (encore) de faire**

Il est possible en Java d'utiliser les commandes **break** et **continue** pour sortir d'une boucle en cours de parcours. Ces fonctionnalités ne sont pas implémentées dans la JSTL au moment de l’écriture de ce tutorial (vérifier si ces fonctionnalités sont désormais possibles).

**Les liens**

La balise <c:url> a pour objectif de générer des URL. Elle possède trois fonctionnalités qui lui sont associées :

* ajouter le nom du contexte aux URL absolues.
* réécrire l'adresse pour la gestion des sessions (si les cookies sont désactivés ou absents, par exemple).
* encoder les noms et contenus des paramètres de l'URL.

L'attribut **value** contient l'adresse, et l'attribut **var** permet comme pour les tags vus auparavant de stocker le résultat dans une variable.

<%-- Génère une url simple, positionnée dans un lien HTML --%>

<a href="<c:url value="test.jsp" />">lien</a>

<%-- Génère une url et la stocke dans la variable lien --%>

<c:url value="test.jsp" var="lien" />

1. Ajout du contexte

Lorsqu'une URL est absolue, c'est-à-dire lorsqu'elle fait référence à la racine de l'application et commence par le caractère / , le contexte de l'application sera par défaut ajouté en début d'adresse.

Pour les adresses absolues, pointant à la racine, sans cette fonctionnalité il serait nécessaire d'écrire en dur le contexte de l'application dans les URL lors du développement, et de toutes les modifier si le contexte est changé par la suite lors du déploiement.

<%-- L'url absolue ainsi générée --%>

<c:url value="/test.jsp" />

<%-- Sera rendue ainsi dans la page web finale si le contextPath est "Test" --%>

/Test/test.jsp

<%-- Et une url relative ainsi générée --%>

<c:url value="test.jsp" />

<%-- Ne sera pas modifiée lors du rendu --%>

test.jsp

2. Gestion des sessions

Si le conteneur JSP détecte un cookie stockant l'identifiant de session dans le navigateur de l'utilisateur, alors aucune modification ne sera apportée à l'URL. Par contre, si ce cookie est absent, les URL générées via la balise <c:url> seront réécrites pour intégrer l'identifiant de session en question. Via ce système une application Java EE ne dépendra pas de l'activation des cookies du côté utilisateur. Ainsi la balise <c:url> est équipée pour leur gestion automatique

<%-- L'url ainsi générée --%>

<c:url value="test.jsp" />

<%-- Sera rendue ainsi dans la page web finale,

si le cookie est présent --%>

test.jsp

<%-- Et sera rendue sous cette forme si le cookie est absent --%>

test.jsp;jsessionid=A6B57CE08012FB431D

3. Encodage

En utilisant la balise <c:url>, les paramètres passé à cette URL seront encodés : les caractères spéciaux qu'ils contiennent éventuellement vont être transformés en leurs codes HTML respectifs. Toutefois, il ne faut pas faire de confusion ici : **ce sont seulement les paramètres (leur nom et contenu) qui seront encodés, le reste de l'URL ne sera pas modifié**. La raison de ce comportement est de pouvoir assurer la compatibilité avec l'action standard d'inclusion <jsp:include>, qui ne sait pas gérer une URL encodée. Pour transmettre proprement des paramètres à une URL, une balise particulière existe : <c:param>. Elle ne peut exister que dans le corps des balises <c:url>, <c:import> ou <c:redirect>. Elle se présente sous cette forme assez intuitive :

<c:url value="/monSiteWeb/countZeros.jsp">

<c:param name="nbZeros" value="${countZerosBean.nbZeros}"/>

<c:param name="date" value="22/06/2010"/>

</c:url>

L'attribut **name** contient donc le nom du paramètre, et **value** son contenu. C'est en réalité cette balise, ici fille de la balise <c:url>, qui se charge de l'encodage des paramètres, et non directement la balise <c:url>. Retenez enfin qu'une telle balise ne peut exister qu'entre deux balises d'URL, de redirection ou d'import, et qu'il est possible d'en utiliser autant que nécessaire.

<%-- Une URL générée de cette manière --%>

<a href="<c:url value="/monSiteWeb/test.jsp">

<c:param name="date" value="22/06/2010"/>

<c:param name="donnees" value="des données contenant des c@r#ct%res bi&a\*\*es..."/>

</c:url>">Lien HTML</a>

<%-- Sera rendue ainsi dans la page web finale --%>

<a href="/test/monSiteWeb/test.jsp?date=22%2f06%2f2010&donnees=des+donn%e9es+contenant+des+c%40r%23ct%25res+bi%26a\*\*es...">Lien HTML</a>

On voit dans cet exemple que :

* les caractères spéciaux contenus dans les paramètres de l'URL ont été transformés : **/** est devenu **%2f**, **é** est devenu **%e9**, etc.
* les caractères **&** séparant les différents paramètres, qui font quant à eux partie intégrante de l'URL, n'ont pas été modifiés en leur code HTML &amp;.

Attention : la balise <c:url> ne permet pas d’encoder proprement l'URL. Elle ne s'occupe que des paramètres. Par conséquent, il faut transformer soi-même les caractères spéciaux, comme le & séparant les paramètres d'une URL, en leur code HTML équivalent (en l'occurrence, & doit devenir &amp; pour que la syntaxe soit valide). Il est possible d'effectuer ces transformations à l'aide de la balise <c:out> !

Pour résumer :

<%-- L'url avec paramètres ainsi générée --%>

<c:url value="/monSiteWeb/countZeros.jsp">

<c:param name="nbZeros" value="123"/>

<c:param name="date" value="22/06/2010"/>

</c:url>

<%-- Sera rendue ainsi dans la page web finale,

si le cookie est présent et le contexte est Test --%>

/Test/monSiteWeb/countZeros.jsp?nbZeros=123&date=22%2f06%2f2010

<%-- Et sera rendue sous cette forme si le cookie est absent --%>

/Test/monSiteWeb/countZeros.jsp;jsessionid=A6B57CE08012FB431D?nbZeros=123&date=22%2f06%2f2010

**Redirection**

La balise <c:redirect> est utilisée pour envoyer un message de redirection HTTP au navigateur de l'utilisateur. Si elle ressemble à l'action <jsp:forward>, elle va entraîner un changement de l'URL dans le navigateur de l'utilisateur final, contrairement à <jsp:forward> qui est transparente du point de vue de l'utilisateur (l'URL dans la barre de navigation du navigateur n'est pas modifiée). La raison de cette différence de comportement est simple : le forwarding se fait côté serveur, contrairement à la redirection qui est effectuée par le navigateur. Ceci permet de rediriger l’utilisateur vers une page extérieure à l’application.

Au final, le forwarding est plus performant, ne nécessitant pas d'aller-retour passant par le navigateur de l'utilisateur final, mais il est moins flexible que la redirection. De plus, utiliser le forwarding impose certaines contraintes : concrètement, **l'utilisateur final n'est pas au courant que sa requête a été redirigée vers une ou plusieurs servlets ou JSP différentes, puisque l'URL qui est affichée dans son navigateur ne change pas.** En d'autres termes, cela veut dire que l'utilisateur ne sait pas si le contenu qu'il visualise dans son navigateur a été produit par la page qu'il a appelée via l'URL d'origine, ou par une autre page vers laquelle la première servlet ou JSP appelée a effectué un forwarding ! Ceci peut donc poser problème si l'utilisateur rafraîchit la page par exemple, puisque cela appellera à nouveau la servlet ou JSP d'origine. Enfin que lorsque l’on utilise le forwarding, le code présent après cette instruction dans la page n'est pas exécuté.

<%-- Forwarding avec l'action standard JSP --%>

<jsp:forward page="/monSiteWeb/erreur.jsp">

<%-- Redirection avec la balise redirect --%>

<c:redirect url="http://www.siteduzero.com"/>

<%-- Les attributs valables pour <c:url/> le sont aussi pour la redirection.

Ici par exemple, l'utilisation de paramètres --%>

<c:redirect url="http://www.siteduzero.com">

<c:param name="mascotte" value="zozor"/>

<c:param name="langue" value="fr"/>

</c:redirect>

<%-- Redirigera vers --%>

http://www.siteduzero.com?mascotte=zozor&langue=fr

**Imports**

La balise <c:import> est en quelque sorte un équivalent à <jsp:include>. L'attribut **url** est le seul paramètre obligatoire lors de l'utilisation de cette balise, il désigne le fichier à importer :

<%-- Copie le contenu du fichier ciblé dans la page actuelle --%>

<c:import url="exemple.html"/>

La fonction d'import permet d'utiliser une variable pour stocker le flux récupéré, et ne propose pas simplement de l'inclure dans votre JSP comme c'est le cas avec <jsp:include>. Cette fonctionnalité est importante, puisqu'elle permet d'effectuer des traitements sur les pages importées. L'attribut utilisé pour ce faire est nommé **varReader**. Ce système prend toute son importance lorsqu'il s'agit de lire et de parser des fichiers XML :

<%-- Copie le contenu d'un fichier xml dans une variable (fileReader),

puis parse le flux récupéré dans une autre variable (doc). --%>

<c:import url="test.xml" varReader="fileReader">

<x:parse var="doc" doc="${fileReader}" />

</c:import>

Le contenu du **varReader** n'est accessible qu'à l'intérieur du corps de la balise d'import, entre les tags <c:import> et </c:import>. Il est possible de palier à cela par le biais d'une variable de scope.

<c:import> permet d'inclure des pages extérieures au contexte de la servlet ou du serveur.

<%-- Les attributs valables pour <c:url/> le sont aussi pour la redirection.

Ici par exemple, l'utilisation de paramètres --%>

<c:import url="footer.jsp">

<c:param name="design" value="bleu"/>

</c:import>

**Les autres bibliothèques de la JSTL**

La JSTl contient en réalité cinq bibliothèques :

* **fmt** : destinée au formatage et au parsage des données. Permet notamment la localisation de l'affichage.
* **fn** : destinée au traitement de chaînes de caractères.
* **sql** : destinée à l'interaction avec une base de données. Attention : le code ayant trait au stockage des données dans une application web Java EE suivant le modèle MVC doit être masqué de la vue, et être encapsulé dans le modèle, éventuellement dans une couche dédiée (cf. modèle de conception DAO).
* **xml** : destinée au traitement de fichiers et données XML. À l'instar de la bibliothèque **sql**, celle-ci trouve difficilement sa place dans une application MVC, ces traitements ayant bien souvent leur place dans des objets du modèle, et pas dans la vue. Cependant elle peut s'avérer très utile dans certains cas.

## La bibliothèque xml

Seuls certains cas particuliers justifient l'utilisation de la bibliothèque **xml**. Dans la plupart des applications MVC, ces actions ont leur place dans le modèle, et pas dans la vue !

**La syntaxe XPath ou XML Path Language**

Le langage XPath permet d'identifier les nœuds dans un document XML. Les éléments d'une expression XPath sont séparés par des slashs '/'.

Fichier XML d’exemple :

<news>

<article id="1">

<auteur>Pierre</auteur>

<titre>Foo...</titre>

<contenu>...bar !</contenu>

</article>

<article id="27">

<auteur>Paul</auteur>

<titre>Bientôt un LdZ J2EE !</titre>

<contenu>Woot ?</contenu>

</article>

<article id="102">

<auteur>Jacques</auteur>

<titre>Coyote court toujours</titre>

<contenu>Bip bip !</contenu>

</article>

</news>

Sélection de diverses portions de ce document :

<!-- Sélection du nœud racine -->

/

<!-- Sélection des nœuds 'article' enfants des nœuds 'news' -->

/news/article

<!-- Sélection de tous les nœuds inclus dans les nœuds 'article' enfants des nœuds 'news' -->

/news/article/\*

<!-- Sélection de tous les nœuds 'auteur' qui ont deux parents quelconques -->

/\*/\*/auteur

<!-- Sélection de tous les nœuds 'auteur' du document via l'opérateur '//' -->

//auteur

<!-- Sélection de tous les nœuds 'article' ayant au moins un parent -->

/\*//article

<!-- Sélection de l'attribut 'id' des nœuds 'article' enfants de 'news' -->

/news/article/@id

<!-- Sélection des nœuds 'article' enfants de 'news' dont la valeur du nœud 'auteur' est 'Paul' -->

/news/article[auteur='Paul']

<!-- Sélection des nœuds 'article' enfants de 'news' dont l'attribut id vaut '12' -->

/news/article[@id='12']

**Les actions de base**

Directive JSP nécessaire pour permettre l'utilisation des balises de la bibliothèque xml dans les pages.

<%@ taglib uri="http://java.sun.com/jsp/jstl/xml" prefix="x" %>

Importe du fichier .xml dans la page.

<c:import url="monDocument.xml" varReader="monReader">

...

</c:import>

L'attribut **varReader** est en quelque sorte un buffer, une variable qui sera utilisée pour une utilisation postérieure du contenu du fichier importé. Lorsque l’on utilise cet attribut, il est impossible d'utiliser conjointement l'attribut **var**. La portée du **varReader** défini est uniquement l'intérieur du corps du <c:import>.Pour pouvoir travailler à l’extérieur de la balise, on utilise l’attribut **var**.

<c:import url="monDocument.xml" varReader="monReader">

<%-- Parse le contenu du fichier XML monDocument.xml dans une variable nommée 'doc' --%>

<x:parse var="doc" doc="${monReader}" />

...

</c:import>

* **var** : contient le nom de la variable de scope qui contiendra les données qui représentent notre document XML parsé.
* **doc** : permet de préciser que l'on souhaite parser le contenu de notre **varReader** défini précédemment lors de l'import. Le **varReader** ici nommé monReader est une variable ; il nous faut donc utiliser une EL pour y faire référence.
* Le **varReader** de la balise <c:import> contient le contenu brut du fichier XML, le **var** de la balise <x:parse> contient le résultat du parsing du fichier XML.
* Importer un fichier n'est pas nécessaire. Il est en effet possible de traiter directement un flux XML depuis la page JSP, en le plaçant dans le corps de la balise <x:parse> :

<%-- Parse le flux XML contenu dans le corps de la balise --%>

<x:parse var="doc">

<news>

<article id="1">

<auteur>Pierre</auteur>

<titre>Foo...</titre>

<contenu>...bar !</contenu>

</article>

<article id="27">

<auteur>Paul</auteur>

<titre>Bientôt un LdZ J2EE !</titre>

<contenu>Woot ?</contenu>

</article>

<article id="102">

<auteur>Jacques</auteur>

<titre>Coyote court toujours</titre>

<contenu>Bip bip !</contenu>

</article>

</news>

</x:parse>

Il reste deux attributs que pas encore abordés :

* **filter** : permet de limiter le contenu traité par l'action de parsing <x:parse> à une portion d'un flux XML seulement. Cet attribut peut s'avérer utile lors de l'analyse de documents XML lourds, afin de ne pas détériorer les performances à l'exécution de la page.
* **systemId** : cet attribut ne sera utile que si le fichier XML contient des références vers des entités externes. On doit y saisir l'adresse URI qui permettra de résoudre les liens relatifs contenus dans le fichier XML. Le processus accédera à ces fichiers externes lors du parsage du document XML spécifié.

**Afficher une expression**

Les balises de la bibliothèque xml vont accéder à des données issues de documents XML, via des expressions XPath.

Pour afficher un élément : Dans une expression XPath, pour faire référence à une variable nommée nomVar on n'utilise pas ${nomVar} comme c'est le cas dans une EL, mais $nomVar.

<c:import url="monDocument.xml" varReader="monReader">

<%-- Parse le contenu du fichier XML monDocument.xml dans une variable nommée 'doc' --%>

<x:parse var="doc" doc="${monReader}" />

<x:out select="$doc/news/article/auteur" />

</c:import>

Affichera « Pierre ». Cela vient du fait que l'expression XPath renvoie un ensemble de nœuds, en l'occurrence les nœuds "auteur". Cet ensemble de nœuds est stocké dans une structure de type NodeSet, un type propre à XPath qui implémente le type Java standard NodeList. <x:out> ne gère pas réellement un ensemble de nœuds et n'affiche que le premier nœud de cet ensemble.

<x:out select="count($doc/news/article/auteur)" /> renvoie le nombre d'éléments que l'expression XPath a sélectionnés et stockés dans le NodeSet.

<%-- Récupère le document nommé 'doc' enregistré auparavant en session, via l'objet implicite sessionScope --%>

<x:out select="$sessionScope:doc/news/article" />

<%-- Sélectionne le nœud 'article' dont l'attribut 'id' a pour valeur le contenu de la variable

nommée 'idArticle' qui a été passée en paramètre de la requête, via l'objet implicite param --%>

<x:out select="$doc/news/article[@id=$param:idArticle]"/>

**Créer une variable**

<%-- Enregistre le résultat de l'expression XPath, spécifiée dans l'attribut select, dans une variable de session nommée 'auteur' --%>

<x:set var="auteur" scope="session" select="$doc//auteur" />

<%-- Affiche le contenu de la variable nommée 'auteur' enregistrée en session --%>

<x:out select="$sessionScope:auteur" />

* L'attribut **select** remplace l'attribut **value**. Une expression XPath est attendue, et non pas une EL.
* L'attribut **var** est obligatoire.

**Une condition simple**

<%-- Afficher le titre de la news postée par 'Paul' --%>

<x:if select="$doc/news/article[auteur='Paul']">

Paul a déjà posté une news dont voici le titre :

<x:out select="$doc/news/article[auteur='Paul']/titre" />

</x:if>

Il est possible de stocker le résultat du test conditionnel en spécifiant un attribut **var**.

**Des conditions multiples**

<%-- Affiche le titre de la news postée par 'Nicolas' si elle existe, et un simple message sinon --%>

<x:choose>

<x:when select="$doc/news/article[auteur='Nicolas']">

Nicolas a déjà posté une news dont voici le titre :

<x:out select="$doc/news/article[auteur='Nicolas']/titre" />

</x:when>

<x:otherwise>

Nicolas n'a pas posté de news.

</x:otherwise>

</x:choose>

**Les boucles**

Il n'existe qu'un seul type de boucles dans la bibliothèque xml de la JSTL, la balise <x:forEach> :

<!-- Affiche les auteurs et titres de tous les articles -->

<p>

<x:forEach var="element" select="$doc/news/article">

<strong><x:out select="$element/auteur" /></strong> :

<x:out select="$element/titre" />.<br/>

</x:forEach>

</p>

Il est possible de faire intervenir un pas de parcours via l'attribut **step**, de définir les index de début et de fin via les attributs **begin** et **end**, ou encore d'utiliser l'attribut **varStatus** pour accéder à l'état de chaque itération.

**Les transformations**

La bibliothèque xml de la JSTL permet d'appliquer des transformations à un flux XML via une feuille de style XSL. La balise dédiée à cette tâche est <x:transform>.

On le même fichier XML que pour les exemples précédents, ainsi que la feuille de style XSL suivante :

<?xml version="1.0" encoding="utf-8"?>

<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">

<html xmlns="http://www.w3.org/1999/xhtml">

<head>

<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

<title>Mise en forme avec XSLT</title>

</head>

<body>

<table width="1000" border="1" cellspacing="0" cellpadding="0">

<tr>

<th scope="col">Id</th>

<th scope="col">Auteur</th>

<th scope="col">Titre</th>

<th scope="col">Contenu</th>

</tr>

<xsl:for-each select="/news/article">

<tr>

<td>

<xsl:value-of select="@id" />

</td>

<td>

<xsl:value-of select="auteur" />

</td>

<td>

<xsl:value-of select="titre" />

</td>

<td>

<xsl:value-of select="contenu" />

</td>

</tr>

</xsl:for-each>

</table>

</body>

</html>

</xsl:template>

</xsl:stylesheet>

Cette feuille affiche simplement les différents éléments de notre fichier XML dans un tableau HTML.  
Et voici comment appliquer la transformation basée sur cette feuille de style à notre document XML :

<c:import varReader="xslFile" url="test.xsl">

<c:import varReader="xmlFile" url="monDocument.xml">

<x:transform doc="${xmlFile}" xslt="${xslFile}"/>

</c:import>

</c:import>

* **doc** : contient la référence au document XML sur lequel la transformation doit être appliquée. Attention, ici on parle bien du document XML d'origine, et pas d'un document analysé via <x:parse>. On travaille bien directement sur le contenu XML.
* **xslt** : contient logiquement la feuille de style XSL.

En l'absence d'attribut **var**, le contenu transformé sera automatiquement généré dans la page HTML finale. Et lorsqu’on accédera à cette page JSP depuis le navigateur, on apercevra un tableau contenant les données de notre fichier XML. Ceci est particulièrement intéressant lorsque l’on souhaite formater un contenu XML en HTML, par exemple lors de la lecture de flux RSS. Si par contre on précise un attribut **var**, le résultat de cette transformation sera alors stocké dans la variable de scope ainsi créée, de type Document. Il existe également un attribut **result** qui, en l'absence des attributs **var** et **scope**, stocke l'objet créé par la transformation.

Pour terminer, il est possible de passer des paramètres à une transformation XSLT, en utilisant la balise <x:param>. Cette dernière ne peut exister que dans le corps d'une balise <x:transform>.

<c:import var="xslFile" url="test.xsl"/>

<c:import var="xmlFile" url="monDocument.xml"/>

<x:transform doc="${xmlFile}" xslt="${xslFile}">

<x:param name="couleur" value="orange" />

</x:transform>

Le comportement et l'utilisation sont identiques à ceux de <c:param> : deux attributs **name** et **value** contiennent simplement le nom et la valeur du paramètre à transmettre. Ici dans cet exemple, ma feuille de style ne traite pas de paramètre, et donc ne fait rien de ce paramètre nommé **couleur** que je lui passe.

Il reste deux attributs non explicités : **docSystemId** and **xsltSystemId**. Ils ont tous deux la même utilité que l'attribut **systemId** de la balise <x:parse>, et s'utilisent de la même façon : il suffit d'y renseigner l'URI destinée à résoudre les liens relatifs contenus respectivement dans le document XML et dans la feuille de style XSL.

## Inclure automatiquement la JSTL *Core* à toutes ses JSP

L’inclusion automatique d’une bibliothèque à toutes les pages JSP d’un projet se fait via le fichier web.xml :

<?xml version="1.0" encoding="UTF-8"?>

<web-app>

<jsp-config>

<jsp-property-group>

<url-pattern>\*.jsp</url-pattern>

<include-prelude>/WEB-INF/taglibs.jsp</include-prelude>

</jsp-property-group>

</jsp-config>

...

La balise <jsp-property-group> ne contient dans notre cas que deux balises :

* <url-pattern>, qui permet de **spécifier à quels fichiers appliquer l'inclusion** automatique.
* <include-prelude>, qui permet de préciser l'emplacement du fichier à inclure **en tête de chacune des pages** couvertes par le pattern précédemment défini.
* <include-coda> permettrait de préciser l'emplacement du fichier à inclure **en fin de chacune des pages** couvertes par le pattern précédemment défini.

Ce système est équivalent à une inclusion statique, en d'autres termes une directive include <%@ include file="/WEB-INF/taglibs.jsp" %> placée en tête de chaque JSP.

## Formulaires : le b.a.-ba

La balise <form> permet de créer un formulaire.

<form method="post" action="inscription">

</form>

**La méthode**

Il est possible d'envoyer les données d'un formulaire par deux méthodes différentes :

* **get** : les données transiteront par l'URL via des paramètres dans une requête HTTP GET. En raison des limitations de la taille d'une URL, cette méthode est peu utilisée pour l'envoi de données.
* **post** : les données ne transiteront pas par l'URL mais dans le corps d'une requête HTTP POST, l'utilisateur ne les verra donc pas dans la barre d'adresses de son navigateur. Malgré leur invisibilité apparente, les données envoyées via la méthode POST restent aisément accessibles, et ne sont donc pas plus sécurisées qu'avec la méthode GET : il faut donc **toujours vérifier la présence et la validité des paramètres avant de les utiliser.**

**La cible**

L'attribut **action** de la balise <form> permet de définir la page à laquelle seront envoyées les données du formulaire. C'est l'URL permettant de joindre cette servlet, c'est-à-dire l'URL qui est spécifiée dans le fichier web.xml, qui doit être précisée dans le champ **action** du formulaire.

## Formulaires : à la mode MVC

C'est le modèle qui doit s'occuper de traiter les données. Le contrôleur doit avoir pour unique but d'aiguiller les requêtes entrantes et d'appeler les éventuels traitements correspondants.

## La session : connectez vos clients

**Le principe de la session**

Le protocole HTTP est un protocole dit "sans état" : cela signifie que le serveur, une fois qu'il a envoyé une réponse à la requête d'un client, ne conserve pas les données le concernant. Le serveur traite les clients requête par requête et est absolument incapable de faire un rapprochement entre leur origine : pour lui, chaque nouvelle requête émane d'un nouveau client. C'est pour pallier cette lacune que le concept de session a été créé : il permet au serveur de mémoriser des informations relatives au client, d'une requête à l'autre.

* La session représente un espace mémoire alloué pour chaque utilisateur, permettant de sauvegarder des informations tout le long de leur visite.
* Le contenu d'une session est conservé jusqu'à ce que l'utilisateur ferme son navigateur, reste inactif trop longtemps, ou encore lorsqu'il se déconnecte du site.
* L'objet Java sur lequel se base une session est l'objet [HttpSession](http://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpSession.html).
* Il existe un objet implicite **sessionScope** permettant d'accéder directement au contenu de la session depuis une expression EL dans une page JSP. Pour manipuler cette objet depuis une servlet, il faut le récupérer depuis un objet HttpServletRequest :

/\* Permet de récupérer la session associée à la requête HTTP en cours si elle existe, ou d'en créer une si elle n'existe pas encore. \*/

HttpSession session = request.getSession();

Tant que cette ligne de code n'a pas été appelée, la session n'existe pas !

* Cet objet HttpSession propose un couple de méthodes [setAttribute()](http://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpSession.html#setAttribute%28java.lang.String,%20java.lang.Object%29) / [getAttribute()](http://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpSession.html#getAttribute%28java.lang.String%29), permettant la mise en place d'objets au sein de la session et leur récupération.
* Il propose une méthode [getId()](http://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpSession.html#getId%28%29), retournant un identifiant unique permettant de déterminer à qui appartient telle session.

Il suffit d'appeler le code suivant pour enregistrer un objet en session depuis la servlet puis le récupérer :

/\* Création ou récupération de la session \*/

HttpSession session = request.getSession();

/\* Mise en session d'une chaîne de caractères \*/

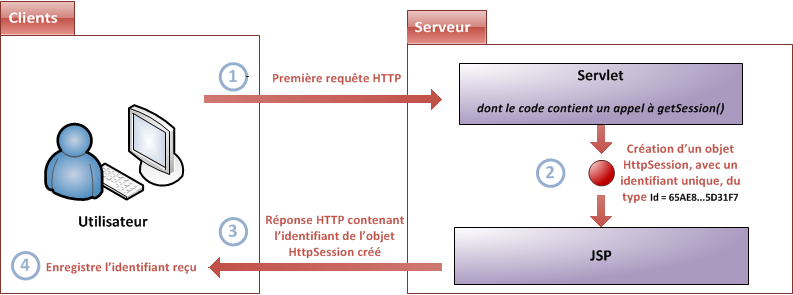
String exemple = "abc";

session.setAttribute( "chaine", exemple );

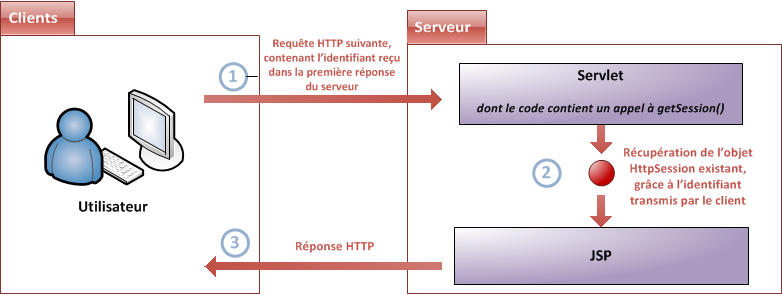
/\* Récupération de l'objet depuis la session \*/

String chaine = (String) session.getAttribute( "chaine" );

Ce qu’il se passe concernant la session lors de la première visite d’un utilisateur sur une page ou servlet contenant un appel à request.getSession() :



Ce qu’il se passe lors des visites suivantes :



**Les vérifications**

Exemple d’utilisation de la session dans la Vue (dans une page .jsp) :

<%-- Vérification de la présence d'un objet utilisateur en session

sessionUtilisateur = objet envoyé par la servlet--%>

<c:if test="${!empty sessionScope.sessionUtilisateur}">

<%-- Si l'utilisateur existe en session, alors on affiche son adresse email. --%>

<p class="succes">Vous êtes connecté(e) avec l'adresse : ${sessionScope.sessionUtilisateur.email}</p>

</c:if>

Il est possible de personnaliser le temps d’expiration de la session via le fichier web.xml de l’application :

<session-config>

<session-timeout>1</session-timeout>

</session-config>

<session-timeout> : permet de définir en minutes le temps d'inactivité de l'utilisateur après lequel sa session est détruite.

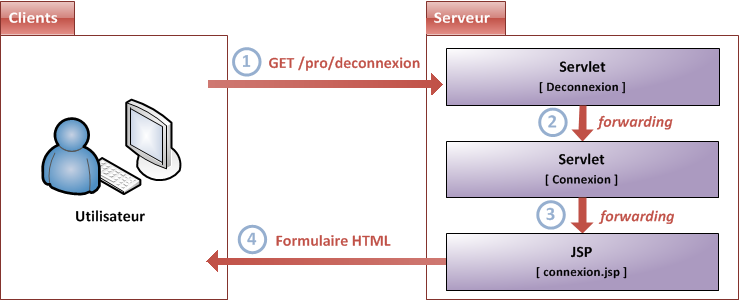
**L'utilisateur se déconnecte**

Pour détruire manuellement une session, la méthode [invalidate()](http://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpSession.html#invalidate%28%29) supprime une session et les objets qu'elle contient, et envoie une exception si jamais elle est appliquée sur une session déjà détruite.

response.sendRedirect( URL\_REDIRECTION ); Méthode de l'objet HttpServletResponse, redirection vers la page spécifiée en lieu et place du forwarding que nous utilisions auparavant. Une redirection HTTP implique l'envoi d'une réponse au client, alors que le forwarding s'effectue sur le serveur et le client n'en est pas tenu informé. Cela implique notamment que, via un forwarding, il est uniquement possible de cibler des pages internes à l'application, alors que via la redirection il est possible d'atteindre n'importe quelle URL publique !

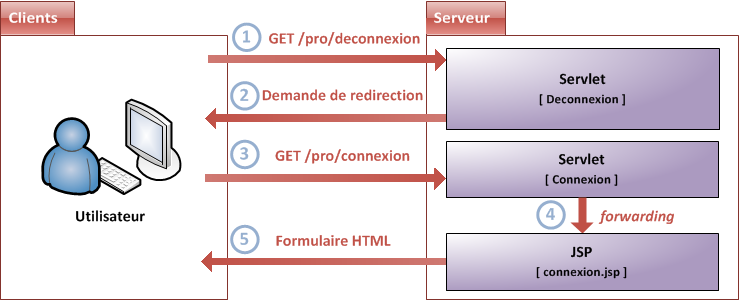
**Différence entre forwarding et redirection**

Ce qui se passe lors d’un forwarding :



Ainsi, lors d’un forwarding, la page affichée chez le client change, mais pas l’url visible dans son navigateur.

Ce qui se passerait pour le même processus si on effectuait une redirection :



Dans ce cas-là, l’url visible dans le navigateur du client change.

Pour éviter que l’utilisateur ne renvoi des données sans le savoir en réactualisant sa page (un transfert d’argent par exemple), il vaut mieux effectuer une redirection plutôt qu’un forwarding.

**La théorie : principe de fonctionnement d’une session**

La chose la plus importante à retenir, C’est que c’est le développeur qui contrôle l'existence d'une session dans l’application. Un objet [HttpSession](http://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpSession.html) dédié à un utilisateur sera créé ou récupéré **uniquement lorsque la page qu'il visite implique un appel à** request.getSession(), en d'autres termes uniquement lorsque l’on a placé un tel appel dans le code. En ce qui concerne la gestion de l'objet, c'est le conteneur de servlets qui va s'en charger, en le créant et le stockant en mémoire. Le serveur dispose d'un moyen pour identifier chaque session qu'il crée : il leur attribue un identifiant unique, que l’on peut retrouver via la méthode session.getId(). Ensuite, le conteneur va mettre en place un élément particulier dans la réponse HTTP qu'il va renvoyer au client : un [Cookie](http://docs.oracle.com/javaee/6/api/javax/servlet/http/Cookie.html). Un cookie est un petit fichier texte qui :

* contient des informations envoyées par le serveur.
* est stocké par votre navigateur, directement sur votre poste.
* a obligatoirement un nom et une valeur (En l'occurrence, le cookie mis en place lors de la gestion d'une session utilisateur par le serveur se nomme **JSESSIONID**, et contient l'identifiant de session unique en tant que valeur.).

Chaque fois qu'il crée une session pour un nouveau client, le serveur va envoyer son identifiant au navigateur de celui-ci. La [spécification du cookie HTTP](http://www.faqs.org/rfcs/rfc2965.html) demande au navigateur de renvoyer ce cookie dans les requêtes suivantes. Le conteneur de servlets va analyser chaque requête HTTP entrante, y chercher le cookie et utiliser sa valeur afin de récupérer l'objet HttpSession associé dans la mémoire du serveur.

Côté serveur l'objet HttpSesssion existera tant que sa durée de vie n'aura pas dépassé le temps qu'il est possible de spécifier dans la section <session-timeout> du fichier **web.xml**, qui est par défaut de trente minutes. Aucune des requêtes suivantes, y compris celles contenant le cookie, n'aura alors accès à la précédente session : le conteneur de servlets en créera une nouvelle.

Côté client, le cookie de session a une durée de vie également, qui par défaut est limitée au temps durant lequel le navigateur reste ouvert. Si le client ouvre à nouveau son navigateur, le cookie associé à la précédente session ne sera alors plus envoyé.

**Remarque : Par défaut, une page JSP va toujours tenter de créer ou récupérer une session.** En effet,la servlet auto-générée par le serveur lors de l’interprétation de notre page JSP contient un « session = pageContext.getSession(); ». Pour éviter la création automatique d'une session depuis une page JSP, il faut utiliser la directive **page :**

<%@ page session="false" %>

Toutefois, cette directive désactive la session sur toute la page JSP. Autrement dit, en procédant ainsi on interdit la manipulation de sessions depuis la page JSP. Dans une page qui accède à des objets présents en session, on ne doit pas mettre en place cette directive ! Cette directive peut être utilisée sur des applications de grandes envergures pour des raisons d’optimisation.

**L'accès à une page sans cookie**

Dernier scénario : l'accès à une page faisant intervenir un appel à request.getSession() depuis un navigateur qui n'accepte pas les cookies ! Pour pallier à cela, on peut remplacer une directive dans nos pages JSP :

<!-- Dans la page connexion. Jsp, on remplace la ligne suivante : -->

<form method="post" action="connexion">

<!-- Par cette ligne : -->

<form method="post" action="<c:url value="/connexion" />">

En effet, [la balise <c:url> de la JSTL Core](http://www.siteduzero.com/tutoriel-3-285009-la-bibliotheque-core.html#ss_part_4) est équipée pour la gestion automatique des sessions. Elle a l’effet suivant :

<%-- L'url ainsi générée --%>

<c:url value="test.jsp" />

<%-- Sera rendue ainsi dans la page web finale, si le cookie est présent --%>

test.jsp

<%-- Et sera rendue sous cette forme si le cookie est absent --%>

test.jsp;jsessionid=BB569C7F07C5E887A4D

Puisque le navigateur n'accepte plus les cookies, nous n'avons pas d'autre choix que de faire passer l'identifiant de session directement au sein de l'URL.

**Remarque :** Le serveur est capable de nous reconnaitre tant que l’on utilisera une url contenant cet id. Dès l’instant on l’id est retiré de l’url, le serveur ne nous reconnait plus.

## Le filtre : créez un espace membre

**Restreindre l'accès à une page**

Nous avons pris l'habitude de placer toutes nos JSP sous le répertoire **/WEB-INF**, et de les rendre accessibles à travers des servlets. Ainsi pour limiter l'accès à une page donnée, la première intuition qui nous vient à l'esprit, c'est de nous servir de la servlet qui lui est associée pour effectuer un test sur le contenu de la session, afin de vérifier si le client est déjà connecté ou non.

Exemple :

/\* Si l'objet utilisateur n'existe pas dans la session en cours, alors l'utilisateur n'est pas connecté. \*/

**if** ( session.getAttribute( *ATT\_SESSION\_USER* ) == **null** ) {

/\* Redirection vers la page publique \*/

response.sendRedirect( request.getContextPath() + *ACCES\_PUBLIC* );

} **else** {

/\* Affichage de la page restreinte \*/

**this**.getServletContext().getRequestDispatcher( *ACCES\_RESTREINT* ).forward( request, response );

}

Remarque : Ici, nous avons mis en place un forwarding vers la page publique au lieu d'une redirection http. Avec un forwarding, l'URL dans le navigateur d'un utilisateur non connecté ne changerait pas lorsqu'il échoue à accéder à la page restreinte. Il serait redirigé vers la page publique de manière transparente, et l'URL de son navigateur lui suggèrerait donc qu'il se trouve sur la page restreinte, ce qui n'est pas le cas.

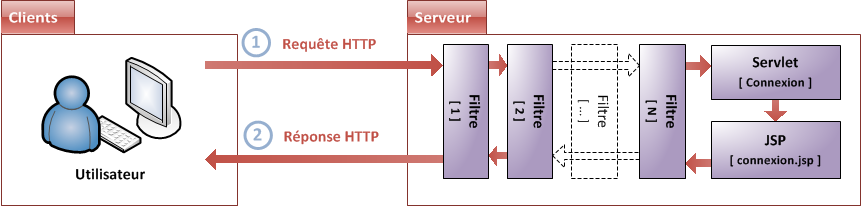
L'URL prise en argument par la méthode de forwarding est relative au **contexte de l'application**, alors que l'URL prise en argument par la méthode de redirection est relative à la **racine de l'application** ! Cette différence est très importante :

* l'URL passée à la méthode getRequestDispatcher() doit être interne à l'application. En l’occurrence, dans notre projet cela signifie qu'il est impossible de préciser une URL qui cible une page en dehors du projet **pro**. Ainsi, un appel à getRequestDispatcher("/accesPublic.jsp") ciblera automatiquement la page /pro/accesPublic.jsp, vous n'avez pas à préciser vous-mêmes le contexte /pro.
* l'URL passée à la méthode sendRedirect() peut être externe à l'application. Cela veut dire que l’on doit spécifier manuellement l'application dans laquelle se trouve la page, et non pas comme avec la méthode de forwarding, dans laquelle toute URL est considérée par défaut comme étant interne à l'application. Cela signifie donc que nous devons préciser le contexte de l'application dans l'URL passée à sendRedirect(). En l'occurrence, nous devons lui dire que nous souhaitons joindre une page contenue dans le projet **pro** : plutôt que d'écrire en dur /pro/accesPublic.jsp, et risquer de devoir manuellement modifier cette URL si nous changeons le nom du contexte du projet plus tard, nous utilisons ici un appel à request.getContextPath(), qui retourne automatiquement le contexte de l'application courante, c'est-à-dire /pro dans notre cas.

Cependant, cette méthode demande de mettre en place un test sur chaque servlet gérant l’accès à une page restreinte. Pour pallier à cela, nous allons mettre en place un filtre.

**Le principe du filtre**

Un filtre est un objet Java qui peut modifier les en-têtes et le contenu d'une requête ou d'une réponse. Il se positionne avant la servlet, et intervient donc en amont dans le cycle de traitement d'une requête par le serveur. Il peut être associé à une ou plusieurs servlets. La figure représente le cas où plusieurs filtres seraient associés à notre servlet de connexion.



Les filtres peuvent intervenir à la fois sur la requête entrante et sur la réponse émise, et ils s'appliquent dans un ordre précis, en cascade.

Quelle est la différence entre un filtre et une servlet ? :

Alors qu'un composant web comme la servlet est utilisé pour générer une réponse HTTP à envoyer au client, le filtre ne crée habituellement pas de réponse. Il se contente généralement d'appliquer d'éventuelles modifications à la paire requête / réponse existante. Voici une liste des actions les plus communes réalisables par un filtre :

* interroger une requête et agir en conséquence.
* empêcher la paire requête / réponse d'être transmise plus loin, autrement dit bloquer son cheminement dans l'application.
* modifier les en-têtes et le contenu de la requête courante.
* modifier les en-têtes et le contenu de la réponse courante.

Le filtre offre trois avantages majeurs, qui sont interdépendants :

* il permet de modifier de manière transparente un échange HTTP. En effet, il n'implique pas nécessairement la création d'une réponse, et peut se contenter de modifier la paire requête / réponse existante.
* tout comme la servlet, il est défini par un *mapping*, et peut ainsi être appliqué à plusieurs requêtes.
* plusieurs filtres peuvent être appliqués en cascade à la même requête.

Ainsi, le filtre est un composant parfaitement adapté à tous les traitements de masse, nécessitant d'être appliqués systématiquement à tout ou partie des pages d'une application.

**Fonctionnement**

Le filtre doit obligatoirement implémenter l'interface [Filter](http://docs.oracle.com/javaee/6/api/javax/servlet/Filter.html). Contrairement au cas de la servlet qui peut par exemple hériter de HttpServlet, il n'existe ici pas de classe fille. L’interface Filter ne contient que trois définitions de méthodes : init(), doFilter() et destroy().

Les méthodes init() et destroy() concernent le cycle de vie du filtre dans l'application. La méthode qui va contenir les traitements effectués par le filtre est donc doFilter().Elle reçoit en arguments la requête, la réponse et **la chaîne des filtres**. C'est à travers cette chaîne qu'un ordre va pouvoir être établi entre les filtres : chaque filtre qui doit être appliqué à la requête va être inclus à la chaîne, qui ressemble en fin de compte à une file d'invocations. Cette chaîne est entièrement gérée par le conteneur, nous n’avons pas à nous en soucier. La seule chose que l’on contrôle, c'est le passage d'un filtre à l'autre dans cette chaîne via l'appel de sa seule et unique méthode, elle aussi nommée [doFilter()](http://docs.oracle.com/javaee/6/api/javax/servlet/FilterChain.html#doFilter%28javax.servlet.ServletRequest,%20javax.servlet.ServletResponse%29).

Comment l'ordre des filtres dans la chaîne est-il établi ? :

Tout comme une servlet, un filtre doit être déclaré dans le fichier **web.xml** de l'application pour être reconnu.

<filter>

<filter-name>Exemple</filter-name>

<filter-class>package.ExempleFilter</filter-class>

</filter>

<filter-mapping>

<filter-name>Exemple</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

L'ordre des déclarations des mappings des filtres dans le fichier est important : c'est cet ordre qui va être suivi lors de l'invocation de plusieurs filtres appliqués à une même requête

**Cycle de vie**

Lorsque l'application web démarre, le conteneur de servlets va créer une instance du filtre et la garder en mémoire durant toute l'existence de l'application. La même instance va être réutilisée pour chaque requête entrante dont l'URL correspond au contenu du champ <url-pattern> du mapping du filtre. Lors de l'instanciation, la méthode init() est appelée par le conteneur : si on souhaite passer des paramètres d'initialisation au filtre, on peut alors les récupérer depuis l'objet [FilterConfig](http://docs.oracle.com/javaee/6/api/javax/servlet/FilterConfig.html) passé en argument à la méthode. Pour chacune de ces requêtes, la méthode doFilter() dont le contenu est défini par le développeur va être appelée. Une fois les traitements appliqués, soit on appelle la méthode [doFilter()](http://docs.oracle.com/javaee/6/api/javax/servlet/FilterChain.html#doFilter%28javax.servlet.ServletRequest,%20javax.servlet.ServletResponse%29) de l'objet FilterChain pour passer au filtre suivant dans la liste, soit on effectue une redirection ou un forwarding pour changer la destination d'origine de la requête.

**Restreindre l'accès à un ensemble de pages**

Restreindre un répertoire :

Code du filtre :

public void doFilter( ServletRequest req, ServletResponse res, FilterChain chain ) throws IOException,ServletException {

/\* Cast des objets request et response \*/

HttpServletRequest request = (HttpServletRequest) req;

HttpServletResponse response = (HttpServletResponse) res;

/\* Récupération de la session depuis la requête \*/

HttpSession session = request.getSession();

/\*\* Si l'objet utilisateur n'existe pas dans la session en cours, alors l'utilisateur n'est pas connecté. \*/

if ( session.getAttribute( ATT\_SESSION\_USER ) == null ) {

/\* Redirection vers la page publique \*/

response.sendRedirect( request.getContextPath() + ACCES\_PUBLIC );

} else {

/\* Affichage de la page restreinte \*/

chain.doFilter( request, response );

}

}

* les objets transmis en arguments à notre méthode doFilter() sont convertis. Il n'existe pas de classe fille implémentant l'interface Filter, alors que côté servlet nous avons bien HttpServlet qui implémente Servlet. Ce qui signifie que notre filtre n'est pas spécialisé, il implémente uniquement Filter et peut traiter n'importe quel type de requête et pas seulement les requêtes HTTP. C'est donc pour cela que nous devons manuellement spécialiser nos objets, en effectuant un cast vers les objets dédiés aux requêtes et réponses HTTP : c'est seulement en procédant à cette conversion que nous aurons accès ensuite à la session, qui est propre à l'objet HttpServletRequest, et n'existe pas dans l'objet ServletRequest.
* nous avons remplacé le forwarding auparavant en place dans notre servlet par un appel à la méthode [doFilter()](http://docs.oracle.com/javaee/6/api/javax/servlet/FilterChain.html#doFilter%28javax.servlet.ServletRequest,%20javax.servlet.ServletResponse%29) de l'objet FilterChain (chain.doFilter( request, response )). Celle-ci a en effet une particularité intéressante : si un autre filtre existe après le filtre courant dans la chaîne, alors c'est vers ce filtre que la requête va être transmise. Par contre, si aucun autre filtre n'est présent ou si le filtre courant est le dernier de la chaîne, alors c'est vers la ressource initialement demandée que la requête va être acheminée.

Enfin, pour appliquer notre filtre RestrictionFilter à un dossier spécifique, nommé « restreint » par exemple, cela se configure dans le fichier **web.xml** :

<filter>

<filter-name>RestrictionFilter</filter-name>

<filter-class>com.sdzee.filters.RestrictionFilter</filter-class>

</filter>

<filter-mapping>

<filter-name>RestrictionFilter</filter-name>

<url-pattern>/restreint/\*</url-pattern>

</filter-mapping>

Restreindre l'application entière :

Cela permet par exemple de forcer l'utilisateur à se connecter pour accéder à l’application. La première chose à faire, c'est de modifier la portée d'application du filtre. Puisque nous souhaitons couvrir l'intégralité des requêtes entrantes, il suffit d'utiliser le caractère \* appliqué à la racine. La déclaration de notre filtre devient donc :

<filter-mapping>

<filter-name>RestrictionFilter</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

Cependant, de cette manière la requête n’aboutira jamais si on ne permet pas d’accéder à au moins une page de l’application sans être connecté. Pour cela, on modifie le code de notre filtre en utilisant request.getRequestDispatcher( URL\_AUTORISE ).forward( request, response ); . Cela permet d’effectuer non plus une redirection HTTP mais un forwarding, afin qu'aucun nouvel échange HTTP n'ait lieu et que la demande aboutisse.

Cependant, cette méthode bloque toute mise en page de notre application via du CSS car lorsque l’on accède à une page web sur laquelle est attachée une feuille de style CSS, le navigateur va, dans les coulisses, envoyer une deuxième requête au serveur pour récupérer silencieusement cette feuille et ensuite appliquer les styles au contenu HTML. Pour régler cela on peut :

* ne plus appliquer le filtre à la racine de l'application, mais seulement sur des répertoires ou pages en particulier, en prenant soin d'éviter de restreindre l'accès à notre page CSS.
* continuer à appliquer le filtre sur toute l'application, mais déplacer notre feuille de style dans un répertoire, et ajouter un passe-droit au sein de la méthode doFilter() du filtre.

Une bonne pratique d'organisation consiste à placer sous un répertoire commun toutes les ressources destinées à être incluses, afin de permettre un traitement simplifié. Par "ressources incluses", on entend généralement les feuilles de style CSS, les feuilles Javascript ou encore les images, bref tout ce qui est susceptible d'être inclus dans une page HTML ou une page JSP.

Par exemple, si ces ressources sont placées dans un répertoire nommé « inc », on rajoute dans la méthode doFilter() de notre filtre :

/\* Non-filtrage des ressources statiques \*/

String chemin = request.getRequestURI().substring(request.getContextPath().length() );

if ( chemin.startsWith( "/inc" ) ) {

chain.doFilter( request, response );

return;

}

* On récupère l'URL d'appel de la requête HTTP via la méthode [getRequestURI()](http://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpServletRequest.html#getRequestURI%28%29), puis on place dans la chaîne **chemin** sa partie finale, c'est-à-dire la partie située après le contexte de l'application. Typiquement, dans notre cas si nous nous rendons sur <http://localhost:8080/pro/restreint/accesRestreint.jsp>, la méthode getRequestURI() va renvoyer /pro/restreint/accesRestreint.jsp et **chemin** va contenir uniquement **/restreint/accesRestreint.jsp** ;
* On teste si cette chaîne **chemin** commence par **/inc** : si c'est le cas, cela signifie que la page demandée est une des ressources statiques que nous avons placées sous le répertoire **inc**, et qu'il ne faut donc pas lui appliquer le filtre !
* On laisse donc la requête poursuivre son cheminement en appelant la méthode doFilter() de la chaîne.
* Le « return » permet de sortir de la classe avant d’exécuter le contrôle.

Enfin, des problèmes de rendu peuvent encore être observés dans certains cas à cause du forwarding mis en place dans le filtre, qui empêchera l’application de trouver le fichier CSS. Pour pallier à cela, on inclut notre CSS dans la page HTML à l’aide de la JSTL :

<link type="text/css" rel="stylesheet" href="<c:url value="/inc/form.css"/>" />

Cela fonctionne car la balise <c:url> ajoute automatiquement le contexte de l'application aux URL absolues qu'elle contient.

**Modifier le mode de déclenchement d'un filtre**

Un filtre est déclenché lors de la réception d'une requête http. En fait il s'agit là d'un comportement par défaut ! Un filtre est tout à fait capable de s'appliquer à un forwarding, mais il faut pour cela modifier sa déclaration dans le fichier **web.xml** :

<filter>

<filter-name>RestrictionFilter</filter-name>

<filter-class>com.sdzee.filters.RestrictionFilter</filter-class>

</filter>

<filter-mapping>

<filter-name>RestrictionFilter</filter-name>

<url-pattern>/\*</url-pattern>

<dispatcher>REQUEST</dispatcher>

<dispatcher>FORWARD</dispatcher>

</filter-mapping>

Il suffit de rajouter un champ <dispatcher> à la fin de la section <filter-mapping>.

De même, si dans le projet nous mettons en place des inclusions et souhaitez leur appliquer un filtre, alors il faudra ajouter cette ligne à la déclaration du filtre :

<dispatcher>INCLUDE</dispatcher>

**Retour sur l'encodage UTF-8**

A ce stade, notre application ne sait toujours pas gérer les caractères accentués. En effet, Eclipse a été paramétré pour encoder en UTF-8. Cependant les problématiques d'encodage interviennent à deux niveaux : côté navigateur et côté serveur. Ainsi, le navigateur du client est capable de gérer l’UTF-8, cependant mais le serveur quant à lui est incapable de déterminer l'encodage des données envoyées par le client, lors d'une requête GET ou POST. Le serveur considère par défaut que les données qui lui sont transmises suivent l'encodage latin ISO-8859-1.

Pour corriger ce comportement, il est nécessaire d'effectuer un appel à la méthode [setCharacterEncoding()](http://docs.oracle.com/javaee/6/api/javax/servlet/ServletRequest.html#setCharacterEncoding%28java.lang.String%29) depuis l'objet HttpServletRequest. Afin que cela soit fait de manière pratique sur l’ensemble des pages de notre application, nous pouvons appliquer cela à l’aide d’un filtre. Or, avec TomCat (marche aussi avec JBoss), un tel filtre est fourni nativement par le serveur. Il suffit d’ajouter le contenu suivant au fichier **web.xml** de l’application :

<filter>

<filter-name>Set Character Encoding</filter-name>

<filter-class>

org.apache.catalina.filters.SetCharacterEncodingFilter

</filter-class>

<init-param>

<param-name>encoding</param-name>

<param-value>UTF-8</param-value>

</init-param>

<init-param>

<param-name>ignore</param-name>

<param-value>false</param-value>

</init-param>

</filter>

<filter-mapping>

<filter-name>Set Character Encoding</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

## Le cookie : le navigateur vous ouvre ses portes

**Le principe du cookie**

Il s'agit d'un petit fichier placé directement dans le navigateur du client. Il lui est envoyé par le serveur à travers les en-têtes de la réponse HTTP, et ne contient que du texte. Il est propre à un site ou à une partie d'un site en particulier, et sera renvoyé par le navigateur dans toutes les requêtes HTTP adressées à ce site ou à cette partie du site.

**Côté http**

* un cookie a obligatoirement un **nom** et une **valeur** associée.
* un cookie peut se voir attribuer certaines **options**, comme une date d'expiration.
* le serveur demande la mise en place ou le remplacement d'un cookie par le paramètre **Set-Cookie** dans l'en-tête de la réponse HTTP qu'il envoie au client.
* le client transmet au serveur un cookie par le paramètre **Cookie** dans l'en-tête de la requête HTTP qu'il envoie au serveur.

**Côté Java EE**

La plate-forme Java EE permet de manipuler un cookie à travers l'objet Java [Cookie](http://docs.oracle.com/javaee/6/api/javax/servlet/http/Cookie.html).

* un cookie doit obligatoirement avoir un nom et une valeur.
* il est possible d'attribuer des options à un cookie, telles qu'une date d'expiration ou un numéro de version. Toutefois, certains navigateurs présentent des bugs dans leur gestion de ces options. Il est préférable d'en limiter l'usage autant que faire se peut afin de rendre notre application aussi multiplateforme que possible.
* la méthode [addCookie()](http://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpServletResponse.html#addCookie%28javax.servlet.http.Cookie%29) de l'objet HttpServletResponse est utilisée pour ajouter un cookie à la réponse qui sera envoyée au client.
* la méthode [getCookies()](http://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpServletRequest.html#getCookies%28%29) de l'objet HttpServletRequest est utilisée pour récupérer la liste des cookies envoyés par le client.

**À propos de la sécurité**

L'information stockée sur un cookie est placée chez le client, par conséquent le développeur n’a absolument aucun contrôle dessus. Ainsi, il y a une règle à suivre lorsque l’on utilise des cookies : **n'y stocker jamais d'informations sensibles en clair**.

## Formulaires : l'envoi de fichiers

**Création du formulaire**

Pour permettre au visiteur de naviguer et sélectionner un fichier pour envoi via un champ de formulaire, il faut utiliser la balise HTML <input type="file">. Pour envoyer un fichier il faut utiliser la méthode POST lors de l'envoi des données du formulaire. En outre, l'attribut optionnel **enctype** de la balise <form> doit être défini à "multipart/form-data".

<form action="<c:url value="/upload" />" method="post" enctype="multipart/form-data">

<fieldset>

<label for="fichier">Emplacement du fichier</label>

<input type="file" id="fichier" name="fichier" />

<br />

<input type="submit" value="Envoyer" class="sansLabel" />

<br />

</fieldset>

</form>

**Récupération des données + API servlet 3.0**

Les données envoyées sont dans un format binaire "multipart" et sont disponibles dans le corps de la requête POST. Une nouvelle [getParts()](http://java.sun.com/javaee/6/docs/api/javax/servlet/http/HttpServletRequest.html#getParts%28%29) est mise à disposition dans l'objet HttpServletRequest, et permet de collecter très simplement les éléments de données de type **multipart**. Pour commencer, nous devons compléter la déclaration de notre servlet dans le fichier **web.xml** avec une section <multipart-config> afin de faire en sorte que la méthode getParts() fonctionne. cette section s'ajoute au sein de la balise de déclaration <servlet> de notre servlet d'upload :

<servlet>

<servlet-name>Upload</servlet-name>

<servlet-class>com.sdzee.servlets.Upload</servlet-class>

<multipart-config>

<location>c:/fichiers</location>

<max-file-size>10485760</max-file-size><!-- 10 Mo -->

<max-request-size>52428800</max-request-size><!-- 5 x 10 Mo -->

<file-size-threshold>1048576</file-size-threshold><!-- 1 Mo -->

</multipart-config>

</servlet>

Voici une rapide description des paramètres optionnels existants :

* <location> contient une **URL absolue** vers un répertoire du système. Un chemin relatif au contexte de l'application n'est pas supporté dans cette balise, il s'agit bien là d'un chemin absolu vers le système. Cette URL sera utilisée pour stocker temporairement un fichier lors du traitement des fragments d'une requête, lorsque la taille du fichier est plus grande que la taille spécifiée dans <file-size-threshold>. Si on précise un répertoire qui n'existe pas sur le disque, alors Tomcat enverra une java.io.IOException lorsqu’on tentera d'envoyer un fichier plus gros que cette limite.
* <file-size-threshold> précise la taille en octets à partir de laquelle un fichier reçu sera temporairement stocké sur le disque.
* <max-file-size> précise la taille maximum en octets autorisée pour **un fichier** envoyé. Si la taille d'un fichier envoyé dépasse cette limite, le conteneur enverra une exception. En l'occurrence, Tomcat lancera une IllegalStateException.
* <max-request-size> précise la taille maximum en octets autorisée pour **une requête multipart/form-data**. Si la taille totale des données envoyées dans une seule requête dépasse cette limite, le conteneur enverra une exception.

En paramétrant ainsi notre servlet, toutes les données **multipart/form-data** seront disponibles à travers la méthode [request.getParts()](http://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpServletRequest.html#getParts%28%29).Celle-ci retourne une collection d'éléments de type [Part](http://docs.oracle.com/javaee/6/api/javax/servlet/http/Part.html), et doit être utilisée en lieu et place de l'habituelle méthode request.getParameter()pour récupérer les contenus des champs de formulaire.

Les contenus des champs du formulaire seront disponibles en tant que collection d'éléments de type Part. Cependant, l'interface [Part](http://docs.oracle.com/javaee/6/api/javax/servlet/http/Part.html) ne propose aucune méthode permettant de déterminer si une donnée reçue renferme un champ classique ou un champ de type fichier.

Afin de déterminer si les données transmises dans une requête HTTP contiennent d'éventuels fichiers ou non, il suffit d'analyser ses en-têtes :

// Pour un champ <input type="text"> nommé 'description'

Content-Disposition: form-data; name="description"

// Pour un champ <input type="file"> nommé 'fichier'

Content-Disposition: form-data; name="fichier"; filename="nom\_du\_fichier.ext"

La seule différence est la présence d'un attribut nommé **filename**. Il suffit donc de s'assurer qu'un en-tête contient le mot-clé **filename** pour être certain que le fragment traité est un fichier. Or, l'interface [Part](http://docs.oracle.com/javaee/6/api/javax/servlet/http/Part.html) contient une méthode part.getHeader() qui renvoie l'en-tête correspondant à un élément.

Exemple de code permettant d’analyser cet en-tête «Content-Disposition» :

private static String getNomFichier( Part part ) {

/\* Boucle sur chacun des paramètres de l'en-tête "content-disposition". \*/

for ( String contentDisposition : part.getHeader( "content-disposition" ).split( ";" ) ) {

/\* Recherche de l'éventuelle présence du paramètre "filename". \*/

if ( contentDisposition.trim().startsWith("filename") ) {

/\* Si "filename" est présent, alors renvoi de sa valeur, c'est-à-dire du nom de fichier. \*/

return contentDisposition.substring( contentDisposition.indexOf( '=' ) + 1 );

}

}

/\* Et pour terminer, si rien n'a été trouvé... \*/

return null;

}

**La différence entre la théorie et la pratique**

En fonction de la version du serveur d’application employée (si c’est une ancienne version), request.getParameter() qui est utilisé pour récupérer le contenu des champs qui ne sont pas des fichiers peut renvoyer null. Dans un tel cas, il va falloir récupérer nous-mêmes le contenu binaire de l'élément Part correspondant au champ de type texte, et le reconstruire sous forme d'un String :

/\*

\* Méthode utilitaire qui a pour unique but de lire l'InputStream contenu

\* dans l'objet part, et de le convertir en une banale chaîne de caractères.

\*/

private String getValeur( Part part ) throws IOException {

BufferedReader reader = new BufferedReader( new InputStreamReader( part.getInputStream(), "UTF-8" ) );

StringBuilder valeur = new StringBuilder();

char[] buffer = new char[1024];

int longueur = 0;

while ( ( longueur = reader.read( buffer ) ) > 0 ) {

valeur.append( buffer, 0, longueur );

}

return valeur.toString();

}

**Enregistrement du fichier**

Pour définir où enregistrer le fichier uploadé, il y a plusieurs solutions. Nous pouvons nous contenter d'écrire en dur le chemin dans une constante directement au sein de notre servlet, ou encore mettre en place un fichier [Properties](http://docs.oracle.com/javase/6/docs/api/java/util/Properties.html) dans lequel nous préciserons le chemin.

On peut également passer le chemin à notre servlet via un paramètre d’initialisation, définit dans le fichier **web.xml** à l’intérieur de la balise <init-param> :

<servlet>

<servlet-name>Upload</servlet-name>

<servlet-class>com.sdzee.servlets.Upload</servlet-class>

<init-param>

<param-name>chemin</param-name>

<param-value>/fichiers/</param-value>

</init-param>

</servlet>

En procédant ainsi, notre servlet va pouvoir accéder à un paramètre nommé **chemin**, disponible à travers la méthode [getInitParameter()](http://docs.oracle.com/javaee/6/api/javax/servlet/ServletConfig.html#getInitParameter%28java.lang.String%29) de l'objet ServletConfig.

Note à propos du **chemin** utilisé : il représente le répertoire du disque local sur lequel les fichiers vont être écrits. Par exemple, si votre serveur tourne sur le disque **C:\** d'une machine sous Windows, alors le chemin **/fichiers/** fera référence au répertoire **C:\fichiers\**. Ainsi, contrairement au champ <location> abordé un peu plus tôt dans lequel nous devions écrire un chemin complet, nous pouvons spécifier ici un chemin relatif au système. Même remarque toutefois, nous devons nous assurer que ce répertoire existe, sinon Tomcat enverra une java.io.IOException lors de la tentative d'écriture sur le disque.

**Écriture du fichier sur le disque**

/\*

\* Lecture du paramètre 'chemin' passé à la servlet via la déclaration

\* dans le web.xml

\*/

String chemin = this.getServletConfig().getInitParameter( CHEMIN );

/\* Écriture du fichier sur le disque \*/

ecrireFichier( part, nomFichier, chemin );

/\*

\* Méthode utilitaire qui a pour but d'écrire le fichier passé en paramètre

\* sur le disque, dans le répertoire donné et avec le nom donné.

\*/

private void ecrireFichier( Part part, String nomFichier, String chemin ) throws IOException {

/\* Prépare les flux. \*/

BufferedInputStream entree = null;

BufferedOutputStream sortie = null;

try {

/\* Ouvre les flux.

\* TAILLE\_TAMPON est un int indiquant la taille du TAMPON en octets.

\*/

entree = new BufferedInputStream( part.getInputStream(), TAILLE\_TAMPON );

sortie = new BufferedOutputStream( new FileOutputStream( new File( chemin + nomFichier ) ),

TAILLE\_TAMPON );

/\*

\* Lit le fichier reçu et écrit son contenu dans un fichier sur le

\* disque.

\*/

byte[] tampon = new byte[TAILLE\_TAMPON];

int longueur;

while ( ( longueur = entree.read( tampon ) ) > 0 ) {

sortie.write( tampon, 0, longueur );

}

} finally {

try {

sortie.close();

} catch ( IOException ignore ) {

}

try {

entree.close();

} catch ( IOException ignore ) {

}

}

}

**Problèmes et limites**

Plusieurs problématiques importantes se posent avec la solution mise en place :

* nous ne gérons pas les fichiers de mêmes noms.
* nous ne savons pas éviter les doublons.
* nous n'avons pas réfléchi à l'endroit choisi pour le stockage.

Comment gérer les fichiers de mêmes noms ?

Le plus simple, c'est de toujours renommer les fichiers reçus avant de les enregistrer sur le disque. Tout un tas de solutions, que nous pouvons combiner, s'offrent alors à nous :

* Ajouter un suffixe à votre fichier si un fichier du même nom existe déjà sur le disque.

Placer nos fichiers dans des répertoires différents selon leur type ou leur extension.

* Renommer, préfixer ou suffixer vos fichiers par un [timestamp](http://fr.wikipedia.org/wiki/Horodatage).
* Nous baser sur un hashCode du contenu binaire du fichier pour générer une arborescence et un nom unique. Etc, etc….

Comment éviter les doublons ?

Une des solutions précédentes peut convenir : en nous basant sur un hashCode du contenu binaire du fichier pour générer une arborescence et un nom unique, nous pouvons ainsi à la fois nous affranchir du nom que l'utilisateur a donné à son fichier, et nous assurer qu'un contenu identique ne sera pas dupliqué à deux endroits différents sur notre disque. Bien évidemment cela demande un peu de réflexion et d'ajustements. Il faudrait en effet générer un hashCode qui soit absolument unique : si deux contenus différents peuvent conduire au même hashCode, alors ce système ne fonctionne plus ! Mais c'est une piste sérieuse qui peut, si elle est bien développée, remplir la mission sans accrocs.

Où stocker les fichiers reçus ?

Deux possibilités s'offrent à nous :

* Stocker les fichiers au sein de l'application web, dans un sous-répertoire du dossier **WebContent** d'Eclipse par exemple.
* Stocker les fichiers en dehors de l'application, dans un répertoire du disque local.

La seconde solution peut poser un problème important : tous les fichiers placés en dehors de l'application, un peu à la manière des fichiers placés sous son répertoire **/WEB-INF**, sont invisibles au contexte web, c'est-à-dire qu'ils ne sont pas accessibles directement via une URL. Autrement dit, nous ne pourrons pas proposer aux utilisateurs de télécharger ces fichiers !

Cependant, en stockant les fichiers directement dans le conteneur de notre application nous les rendons vulnérables à un écrasement lors d'un prochain redémarrage serveur ou d'un prochain redéploiement de l'application. Il est donc déconseillé de stocker les fichiers uploadés par les utilisateurs dans le conteneur.

Pouvoir continuer à utiliser les méthodes request.getParameter() dans tous nos formulaires sans avoir à nous soucier si les champs contiennent des fichiers ou non, il faudra mettre en place un filtre à appliquer à nos servlets qui rendra disponible le contenu des fichiers simplement. C’est un travail assez long et complexe, mais de nombreux exemples existent sur le net ( cf : http://balusc.omnifaces.org/2009/12/uploading-files-in-servlet-30.html ).

**Intégration dans MVC**

Il suffit pour cela de créer un bean regroupement els informations de notre fichier, créer un objet métier qui regroupe les traitements jusqu'à présent réalisés dans la servlet, et y ajouter des méthodes de validation de la description et du fichier si besoin.

## Le téléchargement de fichiers

Cela permet aux utilisateurs de récupérer un fichier présent sur le serveur. Dans cette optique nous allons réaliser une servlet qui aura pour unique objectif de permettre aux clients de télécharger des fichiers.

**Une servlet dédiée**

Les seules ressources auxquelles l'utilisateur peut accéder depuis son navigateur sont les fichiers et dossiers placés sous la racine de votre application, c'est-à-dire sous le dossier **WebContent** de notre projet Eclipse, à l'exception bien entendu du répertoire privé **/WEB-INF**. Ainsi, lorsque nous enregistrons nos fichiers en dehors de notre application web (ailleurs sur le disque dur, ou bien sur un FTP distant, dans une base de données, etc.), **le client ne peut pas y accéder directement par une URL.**

Une des solutions possibles est alors de créer une servlet dont l'unique objectif est de charger ces fichiers depuis le chemin en dehors du conteneur web (ou depuis une base de données, mais nous y reviendrons bien plus tard), et de les **transmettre en flux continu** (en anglais, on parle de streaming) à l'objet HttpServletResponse. Le client va alors visualiser sur son navigateur une fenêtre de type "Enregistrer sous...".

La seule action réalisée par le client sera un clic sur un lien pour télécharger un fichier, notre servlet aura donc uniquement besoin d'implémenter la méthode doGet().

Dans le fichier **web.xml** on écrit :

<servlet-mapping>

<servlet-name>Download</servlet-name>

<url-pattern>/fichiers/\*</url-pattern>

</servlet-mapping>

Nous faisons ici correspondre notre servlet à toute URL commençant par **/fichiers/**, à travers la balise <url-pattern>. Ainsi, toutes les adresses du type http://localhost:8080/pro/fichiers/fichier.ext ou encore http://localhost:8080/pro/fichiers/dossier/fichier.ext pointeront vers notre servlet de téléchargement.

Nous devons maintenant préciser à notre servlet où elle va devoir aller chercher les fichiers sur le disque. Il y a plusieurs manières de faire. On peut par exemple passer ce paramètre via le fichier de configuration **web.xml**:

<servlet>

<servlet-name>Download</servlet-name>

<servlet-class>com.sdzee.servlets.Download</servlet-class>

<init-param>

<param-name>chemin</param-name>

<param-value>/fichiers/</param-value>

</init-param>

</servlet>

Ainsi, côté servlet il nous suffit de lire la valeur associée depuis notre méthode doGet() :

public void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

/\* Lecture du paramètre 'chemin' passé à la servlet via la déclaration dans le web.xml \*/

String chemin = this.getServletConfig().getInitParameter( "chemin" );

}

Maintenant que tout est précisé côté serveur, il nous faut donner au client un moyen de préciser quel fichier il souhaite télécharger. Pour cela, nous éviterons d’utiliser un paramètre de requête car certains navigateurs (IE) ne sont pas capables de détecter correctement un nom de fichier dans les en-têtes http. Pour pallier à cela, nous allons inclure notre nom de fichier directement dans l'URL. Autrement dit, faire en sorte qu'il nous suffise d'appeler une URL du type http://localhost:8080/pro/fichiers/test.txt pour télécharger le fichier nommé **test.txt** !

Pour ce faire, côté servlet nous allons utiliser une méthode de l'objet HttpServletRequest : getPathInfo(). Elle retourne la fraction de l'URL qui correspond à ce qui est situé entre le chemin de base de la servlet et les paramètres de requête. Il faut donc ajouter à notre méthode doGet() la ligne suivante :

String fichierRequis = request.getPathInfo();

Elle retourne null si aucun chemin n'existe dans l'URL, et un chemin existant commence toujours par /. Nous devons donc vérifier si un chemin vide est transmis :

/\* Vérifie qu'un fichier a bien été fourni \*/

if ( fichierRequis == null || "/".equals( fichierRequis ) ) {

/\* Si non, alors on envoie une erreur 404, qui signifie que la ressource demandée n'existe pas \*/

response.sendError(HttpServletResponse.SC\_NOT\_FOUND);

return;

}

[sendError()](http://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpServletResponse.html#sendError%28int%29) de HttpServletResponse permet de retourner au client les messages et codes d'erreur HTTP souhaités.

L'étape suivante consiste à contrôler le nom du fichier transmis et à vérifier si un tel fichier existe :

/\* Décode le nom de fichier récupéré, susceptible de contenir des espaces et autres caractères spéciaux, et prépare l'objet File \*/

fichierRequis = URLDecoder.decode( fichierRequis, "UTF-8");

File fichier = new File( chemin, fichierRequis );

/\* Vérifie que le fichier existe bien \*/

if ( !fichier.exists() ) {

/\* Si non, alors on envoie une erreur 404, qui signifie que la ressource demandée n'existe pas \*/

response.sendError(HttpServletResponse.SC\_NOT\_FOUND);

return;

}

Avant de créer un objet File basé sur le chemin du fichier récupéré, il est nécessaire de convertir les éventuels caractères spéciaux qu'il contient à l'aide de la méthode URLDecoder.decode().

Une fois ces contrôles réalisés, il nous faut encore récupérer le type du fichier transmis, à l'aide de la méthode getMimeType() de l'objet ServletContext. Elle retourne le type du contenu d'un fichier en prenant pour argument son nom uniquement. Si le type de contenu est inconnu, alors la méthode renvoie null :

/\* Récupère le type du fichier \*/

String type = getServletContext().getMimeType( fichier.getName() );

/\* Si le type de fichier est inconnu, alors on initialise un type par défaut \*/

if ( type == null ) {

type = "application/octet-stream";

}

Les types de fichiers sont déterminés par le conteneur lui-même. Lorsque le conteneur reçoit une requête demandant un fichier et qu'il le trouve, il le renvoie au client. Dans la réponse HTTP retournée, il renseigne alors l'en-tête **Content-Type**. Pour ce faire, il se base sur les types MIME dont il a connaissance, en fonction de l'extension du fichier à retourner. Ces types sont spécifiés dans le fichier **web.xml** global du conteneur (qui est situé dans le répertoire **/conf/** du Tomcat Home pour Tomcat par exemple. Pour JBoss, cela peut-être fait directement dans le fichier **web.xml** de l’application). Si nous l'éditons, nous pouvons voir qu'il en contient déjà une bonne quantité ! Par exemple :

<mime-mapping>

<extension>jpeg</extension>

<mime-type>image/jpeg</mime-type>

</mime-mapping>

<mime-mapping>

<extension>jpg</extension>

<mime-type>image/jpeg</mime-type>

</mime-mapping>

Il est possible d'ajouter un type inconnu au serveur, il suffit pour cela d'ajouter une section <mime-mapping> au fichier.

**Génération de la réponse HTTP**

Nous devons initialiser une réponse HTTP et y renseigner les en-têtes nécessaires, à savoir Content-Type, Content-Length et Content-Disposition :

/\* Initialise la réponse HTTP \*/

response.reset();

response.setBufferSize(10240); // 10ko

response.setContentType( type );

response.setHeader( "Content-Length", String.valueOf( fichier.length() ) );

response.setHeader( "Content-Disposition", "attachment; filename=\"" + fichier.getName() + "\"" );

* reset() : efface littéralement l'intégralité du contenu de la réponse initiée par le conteneur.
* setBufferSize() : méthode à appeler impérativement après un reset().
* setContentType() : spécifie le type des données contenues dans la réponse.
* Nous retrouvons ensuite les deux en-têtes HTTP, qu'il faut construire "à la main" via des appels à setHeader().

**Lecture et envoi du fichier**

Nous arrivons enfin à la dernière étape du processus : la lecture du flux et l'envoi au client ! Commençons par mettre en place proprement l'ouverture des flux :

/\* Prépare les flux \*/

BufferedInputStream entree = null;

BufferedOutputStream sortie = null;

try {

/\* Ouvre les flux \*/

entree = new BufferedInputStream( new FileInputStream( fichier ), TAILLE\_TAMPON );

sortie = new BufferedOutputStream( response.getOutputStream(), TAILLE\_TAMPON );

/\* Lit le fichier et écrit son contenu dans la réponse HTTP \*/

/\* À l'aide d'un tableau d'octets jouant le rôle de tampon, la boucle mise en place parcourt le fichier et l'écrit, morceau par morceau, dans la réponse. \*/

byte[] tampon = new byte[TAILLE\_TAMPON];

int longueur;

while ( ( longueur= entree.read( tampon ) ) > 0 ) {

sortie.write( tampon, 0, longueur );

}

} finally {

try {

sortie.close();

} catch ( IOException ignore ) {

}

try {

entree.close();

} catch ( IOException ignore ) {

}

}

Nous pourrions ici très bien utiliser directement les flux de type FileInputStream et ServletOutputStream, mais les objets BufferedInputStream et BufferedOutputStream permettent via l'utilisation d'une mémoire tampon une gestion plus souple de la mémoire disponible sur le serveur :

* dans le flux **entree**, nous ouvrons un FileInputStream sur le fichier demandé. Nous décorons ensuite ce flux avec un BufferedInputStream, avec ici un tampon de la même taille que le tampon mis en place sur la réponse http.
* dans le flux **sortie**, nous récupérons directement le ServletOutpuStream depuis la méthode getOutputStream() de l'objet HttpServletResponse. Nous décorons ensuite ce flux avec un BufferedOutputStream, avec là encore un tampon de la même taille que le tampon mis en place sur la réponse HTTP.

Remarque : **toujours ouvrir les flux dans un bloc** try**, et les fermer dans le bloc** finally **associé**.

**L'état d'un téléchargement**

La méthode vue précédent pour la gestion d’une téléchargement est un approche plutôt simpliste, car elle ne permet pas de connaître l'état du téléchargement en cours : autrement dit, en cas de coupure côté client il est impossible de lui proposer la reprise d'un téléchargement en cours.

Le principe se corse un peu dès lors qu'on souhaite proposer ce type de service. Depuis notre servlet, il faudrait manipuler au minimum trois nouveaux en-têtes de la réponse HTTP afin d'activer cette fonctionnalité :

* Accept-Ranges : cet en-tête de réponse, lorsqu'il contient la valeur "bytes", informe le client que le serveur supporte les requêtes demandant une plage définie de données. Avec cette information, le client peut alors demander une section particulière d'un fichier à travers l'en-tête de requête Range.
* ETag : cet en-tête de réponse doit contenir une valeur unique permettant d'identifier le fichier concerné. Il est possible d'utiliser le système de notre choix, il n'y a aucune contrainte : tout ce qui importe, c'est que chacune des valeurs associées à un fichier soit unique. Certains serveurs utilisent par exemple l'algorithme MD5 pour générer un hash basé sur le contenu du fichier, d'autres utilisent une combinaison d'informations à propos du fichier (son nom, sa taille, sa date de modification, etc.), d'autres génèrent un hash de cette combinaison... Avec cette information, le client peut alors renvoyer l'identifiant obtenu au serveur à travers l'en-tête de requête If-Match ou If-Range, et le serveur peut alors déterminer de quel fichier il est question.
* Last-Modified : cet en-tête de réponse doit contenir un timestamp, qui représente la date de la dernière modification du fichier côté serveur. Avec cette information, le client peut alors renvoyer le timestamp obtenu au serveur à travers l'en-tête de requête If-Unmodified-Since, ou bien là encore If-Range. À propos de ce dernier en-tête, note importante : un timestamp Java est précis à la milliseconde près, alors que le timestamp attendu dans l'en-tête n'est précis qu'à la seconde près. Afin de combler cet écart d'incertitude, il est donc nécessaire d'ajouter une seconde à la date de modification retournée par le client dans sa requête avant de la traiter.

**Réaliser des statistiques**

Avec notre servlet en place, nous avons centralisé la gestion des fichiers demandés par les clients, qui passent dorénavant tous par cette unique passerelle de sortie. Si nous envisageons dans une application de réaliser des statistiques sur les téléchargements effectués par les utilisateurs, il est intuitif d'envisager la modification de cette servlet pour qu'elle relève les données souhaitées : combien de téléchargements par fichier, combien de fichiers par utilisateur, combien de téléchargements simultanés, la proportion d'images téléchargées par rapport aux autres types de fichiers, etc. Cependant, pour bien suivre MVC, cette servlet, unique dans l'application, ne devra en réalité pas se charger de cette tâche.

Le composant idéal pour s'occuper de ce type de traitements, c'est le filtre ! Il suffit en effet d'en appliquer un sur le pattern **/fichiers/\*** pour que celui-ci ait accès à toutes les demandes de téléchargement effectuées par les clients. Tout passera par la méthode doFilter() du filtre en question.

**Vérifier le type d’un fichier**

Pour vérifier le type d’un fichier envoyé par un utilisateur il existe deux solutions :

* Vérifier « à la main » l’extension du fichier. Cependant, cela ne garantit pas son intégrité car l’utilisateur peut avoir changer l’extension pour cacher le véritable contenu du fichier (fichier exécutable, script…).
* Utiliser des bibliothèques, dont par exemple MimeUtil + SLF4J. Pour vérifier si le fichier envoyé est une image par exemple :

/\* Extraction du type MIME du fichier depuis l'InputStream nommé "contenu" \*/

MimeUtil.registerMimeDetector( "eu.medsea.mimeutil.detector.MagicMimeMimeDetector" );

Collection<?> mimeTypes = MimeUtil.getMimeTypes( contenu );

/\*

\* Si le fichier est bien une image, alors son en-tête MIME

\* commence par la chaîne "image"

\*/

if ( mimeTypes.toString().startsWith( "image" ) ) {

/\* Appeler ici la méthode d'écriture du fichier sur le disque... \*/

} else {

/\* Envoyer ici une exception précisant que le fichier doit être une image... \*/

}

Pour afficher directement cette image sans afficher la fenêtre « Enregistrer sous » du navigateur, dans la servlet de download on doit modifier l'en-tête "Content-Disposition" de "attachment" vers "inline".

## Introduction à MySQL et JDBC

La base de données (BDD, ou DB en anglais) est un système qui enregistre des informations. Il existe un vocabulaire spécifique pour désigner les différents éléments composant une BDD :

* la **base** désigne le volume englobant l'ensemble.
* une **table** désigne un tableau de données, elle contient des lignes et des colonnes.
* une **entrée** désigne une ligne.
* un **champ** désigne une colonne.

Les tableaux et les données que contient une BDD sont sauvegardés dans des fichiers écrits sur le disque. Leur format est bien particulier et dépend du système de gestion utilisé (les données n’y sont pas représentées en « tableaux » comme dans la BDD).

**SGBD**

Il n'existe plusieurs solutions de gestion, qu'on raccourcit en SGBD. Par exemple :

* MySQL : solution libre et gratuite, c'est le SGBD le plus répandu.
* PostgreSQL : solution libre et gratuite, moins connue du grand public mais proposant des fonctionnalités inexistantes dans MySQL.
* Oracle : solution propriétaire et payante, massivement utilisée par les grandes entreprises. C'est un des SGBD les plus complets, mais un des plus chers également.
* SQL Server : la solution propriétaire de Microsoft.
* DB2 : la solution propriétaire d'IBM, utilisée principalement dans les très grandes entreprises sur des Mainframes.

Chaque système utilise un ensemble d'algorithmes pour trier et stocker les informations et pour y accéder, via des requêtes écrites en langage SQL. Un tel ensemble porte un nom bien particulier : le **moteur de stockage**. Ex : un moteur **relationnel** s'assure que les relations mises en place entre les données de plusieurs tables sont cohérentes et que si l'on modifie certaines données, ces changements seront répercutés aux tables liées.

Requête pour créer une base :

CREATE DATABASE bdd\_sdzee DEFAULT CHARACTER SET utf8 COLLATE utf8\_general\_ci;

Création d’un utilisateur spécifique à notre application, qui n'aura accès qu'à la base sur laquelle nous travaillons :

CREATE USER 'java'@'localhost' IDENTIFIED BY ‘pass';

GRANT ALL ON bdd\_sdzee.\* TO 'java'@'localhost' IDENTIFIED BY 'pass';

Ici, « java » est le login et « pass » le mot de passe.

**Création d'une table**

CREATE TABLE bdd\_sdzee.Utilisateur (

id INT( 11 ) NOT NULL AUTO\_INCREMENT ,

email VARCHAR( 60 ) NOT NULL ,

mot\_de\_passe VARCHAR( 32 ) NOT NULL ,

nom VARCHAR( 20 ) NOT NULL ,

date\_inscription DATETIME NOT NULL ,

PRIMARY KEY ( id ),

UNIQUE ( email )

) ENGINE = INNODB;

Pour afficher les tables de la base sélectionnée : SHOW tables;

Pour afficher une description d’une de ces tables : DESCRIBE utilisateur;

**Insertion de données**

INSERT INTO Utilisateur (email, mot\_de\_passe, nom, date\_inscription) VALUES ('coyote@mail.acme', MD5('bipbip'), 'Coyote', NOW());

Pour visualiser toutes les données d’une table : SELECT \* FROM Utilisateur;

**Mise en place de JDBC dans le projet**

JDBC est une API qui fait partie intégrante de la plate-forme Java, et qui est constituée de classes permettant l'accès depuis nos applications Java à des données rangées sous forme de tables. Dans la très grande majorité des cas, il s'agira bien entendu de bases de données stockées dans un SGBD ! Les actions rendues possibles par cette API sont :

* la connexion avec le SGBD.
* l'envoi de requêtes SQL au SGBD depuis une application Java.
* le traitement des données et éventuelles erreurs retournées par le SGBD lors des différentes étapes du dialogue (connexion, requête, exécution, etc.).

Il existe plusieurs SGBD différents et bien qu'ils se basent tous sur le langage SQL, chacun a sa manière de gérer les données. L'avantage de JDBC, c'est qu'il est nativement prévu pour pouvoir s'adapter à n'importe quel SGBD. Pour faire en sorte que notre application puisse dialoguer avec MySQL par exemple, nous aurons simplement besoin d'ajouter à notre projet **un driver** qui est spécifique à MySQL. Si nous utilisions PostgreSQL, il nous faudrait, à la place, ajouter **un driver** propre à PostgreSQL, etc. Il s'agit tout bonnement d'une archive .jar que nous allons ajouter au **build-path** de notre projet.

## Communiquez avec votre BDD

**Chargement du driver**

Il est nécessaire de procéder à ce que l'on nomme **le chargement du driver** depuis le code de notre application.

/\* Chargement du driver JDBC pour MySQL \*/

try {

Class.forName( "com.mysql.jdbc.Driver" );

} catch ( ClassNotFoundException e ) {

/\* Gérer les éventuelles erreurs ici. \*/

}

"com.mysql.jdbc.Driver" est le nom du driver, et il est fourni par son constructeur (MySQL).

Dans la pratique, **il est absolument inutile de charger le driver avant chaque connexion**, une seule fois durant le chargement de l'application suffit !

Pour communiquer avec notre BDD il faut :

* nous connecter à la base.
* créer et exécuter une requête SQL
* analyser son résultat.
* fermer les différentes ressources mises en jeu.

**Remarque :** le driver doit-être mis dans le répertoire WEB-INF/lib/ de l’application, ou alors dans un dossier bien spécifique sur le serveur.

**Connexion à la base, création et exécution d'une requête**

**Identification de l'URL**

Pour nous connecter à la base de données MySQL depuis notre application Java, nous avons besoin d'une URL spécifique à JDBC, qui respecte la syntaxe générale suivante : jdbc:mysql://nomhote:port/nombdd.

* **nomhote** : le nom de l'hôte sur lequel le serveur MySQL est installé. S'il est en place sur la même machine que l'application Java exécutée, alors nous pouvons simplement spécifier **localhost**. Cela peut également être une adresse IP comme 127.0.0.1. Au passage, si nous rencontrons des problèmes de connectivité en spécifiant **localhost** et que l'utilisation de **127.0.0.1** à la place les résout, alors nous avons un souci de configuration réseau (DNS, hosts, etc.).
* **port** : le port TCP/IP écouté par notre serveur MySQL. Par défaut, il s'agit du port **3306 (8080 pour JBoss)**.
* **nombdd** : le nom de la base de données à laquelle nous souhaitons nous connecter.

**Établissement de la connexion**

Après le chargement du driver, nous pouvons tenter d'établir une connexion avec notre base de données :

/\* Connexion à la base de données \*/

String url = "jdbc:mysql://localhost:3306/bdd\_sdzee";

String utilisateur = "java";

String motDePasse = "pass";

Connection connexion = null;

try {

connexion = DriverManager.getConnection( url, utilisateur, motDePasse );

/\* Ici, nous placerons nos requêtes vers la BDD \*/

/\* ... \*/

} catch ( SQLException e ) {

/\* Gérer les éventuelles erreurs ici \*/

} finally {

if ( connexion != null )

try {

/\* Fermeture de la connexion \*/

connexion.close();

} catch ( SQLException ignore ) {

/\* Si une erreur survient lors de la fermeture, il suffit de l'ignorer. \*/

}

}

L'établissement d'une connexion s'effectue à travers l'objet DriverManager. Il suffit d'appeler sa méthode statique getConnection() pour récupérer un objet de type Connection. Celle-ci prend en argument l'adresse de la base de données, le nom d'utilisateur et le mot de passe associé. L'appel à cette méthode peut retourner des erreurs de type SQLException :

* si une erreur SQLException: No suitable driver est envoyée, alors cela signifie que le driver JDBC n'a pas été chargé ou que l'URL n'a été reconnue par aucun des drivers chargés par notre application.
* si une erreur SQLException: Connection refused ou Connection timed out ou encore CommunicationsException: Communications link failure est envoyée, alors cela signifie que la base de données n'est pas joignable.

Si un de ces derniers cas survient, voici une liste des causes possibles et les pistes de résolution associées :

|  |  |
| --- | --- |
| **Cause éventuelle** | **Piste de résolution** |
| Le serveur MySQL est éteint ? | Démarrez le serveur MySQL... |
| Le numéro de port dans l'URL est manquant ou incorrect ? | Ouvrez le fichier de configuration **my.cnf** de votre serveur MySQL, et vérifiez le port qui y est spécifié. |
| Le nom d'hôte ou l'adresse IP dans l'URL est incorrect(e) ? | Testez la connectivité en effectuant un simple ping. |
| Le serveur MySQL n'accepte pas de connexions TCP/IP ? | Vérifiez que MySQL a été lancé sans l'option --skip-networking. |
| Il n'y a plus aucune connexion disponible sur le serveur MySQL ? | Redémarrez MySQL, et corrigez le code de votre application pour qu'il libère les connexions efficacement. |
| Quelque chose entre l'application Java et le serveur MySQL bloque la connexion, comme un pare-feu ou un proxy ? | Configurez votre pare-feu et/ou proxy pour qu'il(s) autorise(nt) le port écouté par votre serveur MySQL. |
| Le nom d'hôte dans l'URL n'est pas reconnu par votre serveur DNS local ? | Utilisez l'adresse IP dans l'URL au lieu du nom d'hôte, ou actualisez si possible votre DNS. |

Enfin, peu importe que la connexion ait réussi ou non, sa fermeture dans un bloc finally est extrêmement importante ! Si nous ne fermons pas les connexions que nous ouvrons, et en gardons un grand nombre ouverts sur une courte période, le serveur risque d'être saturé et de ne plus accepter aucune nouvelle connexion. L’application risque alors de planter. La bonne pratique à suivre est de toujours ouvrir et fermer la connexion le plus finement possible, c'est-à-dire au plus près de l'exécution de nos requêtes, dans un ensemble try-catch-finally.

Remarque : L'ouverture d'une connexion a un coût non négligeable en termes de performances. Dans une application très fréquentée, il devient hors de question de procéder à des ouvertures/fermetures à chaque requête effectuée.

**Création d'une requête**

Avant de pouvoir créer des instructions SQL, nous devons tout d'abord créer un objet de type Statement. Il s'agit d'une interface dont le rôle est de permettre l'exécution de requêtes. Pour initialiser cet objet il suffit d'appeler la méthode createStatement() de l'objet Connection précédemment obtenu.

/\* Création de l'objet gérant les requêtes \*/

Statement statement = connexion.createStatement();

**Exécution de la requête**

Une fois l'objet Statement initialisé, il devient alors possible d'exécuter une requête. Pour ce faire, celui-ci met à notre disposition une série de méthodes, notamment les deux suivantes :

* executeQuery() : cette méthode est dédiée à la lecture de données via une requête de type SELECT.
* executeUpdate() : cette méthode est réservée à l'exécution de requêtes ayant un effet sur la base de données (écriture ou suppression), typiquement les requêtes de type INSERT, UPDATE, DELETE, etc.

En outre, il existe des variantes de chaque méthode prenant en compte d'autres arguments, ainsi que deux autres méthodes nommées execute() et executeBatch().

**Exécution d'une requête de lecture**

La méthode executeQuery()retourne un objet de type ResultSet contenant le résultat de la requête.

/\* Exécution d'une requête de lecture \*/

ResultSet resultat = statement.executeQuery( "SELECT id, email, mot\_de\_passe, nom FROM Utilisateur;" );

**Exécution d'une requête d'écriture**

La méthode [executeUpdate()](http://docs.oracle.com/javase/6/docs/api/java/sql/Statement.html#executeUpdate%28java.lang.String%29)retourne un entier représentant le nombre de lignes affectées par la requête réalisée. Si par exemple nous réalisons une insertion de données via un INSERT, cette méthode retournera 0 en cas d'échec et 1 en cas de succès. Si nous réalisons une mise à jour via un UPDATE, elle retournera le nombre de lignes mises à jour. Idem en cas d'une suppression via un DELETE, etc.

/\* Exécution d'une requête d'écriture \*/

int statut = statement.executeUpdate( "INSERT INTO Utilisateur (email, mot\_de\_passe, nom, date\_inscription) VALUES ('jmarc@mail.fr', MD5('lavieestbelle78'), 'jean-marc', NOW());" );

**Accès aux résultats de la requête**

**Retour d'une requête de lecture**

Une requête de lecture via la méthode statement.executeQuery() retourne un objet de type ResultSet. C’est en gros un tableau qui contient les éventuelles données retournées par la base de données sous forme de lignes. Pour parcourir ce tableau, il faut fait comme suit :

/\* Exécution d'une requête de lecture \*/

ResultSet resultat = statement.executeQuery( "SELECT id, email, mot\_de\_passe, nom FROM Utilisateur;" );

/\* Récupération des données du résultat de la requête de lecture \*/

while ( resultat.next() ) {

int idUtilisateur = resultat.getInt( "id" );

String emailUtilisateur = resultat.getString( "email" );

String motDePasseUtilisateur = resultat.getString( "mot\_de\_passe" );

String nomUtilisateur = resultat.getString( "nom" );

/\* Traiter ici les valeurs récupérées. \*/

}

Il existe une méthode de récupération proposée par l'objet ResultSet par type de données récupérables :

* une méthode resultat.getInt() pour récupérer un entier.
* une méthode resultat.getString() pour récupérer une chaîne de caractères.
* Etc etc…

Chacune de ces méthodes existe sous deux formes différentes :

* soit elle prend en argument le nom du champ visé dans la table de la base de données.
* soit elle prend en argument l'index du champ visé dans la table de la base de données.

Si nous précisons un nom de champ qui n'existe pas, ou un index de champ qui dépasse l'index de la dernière colonne, alors la méthode de récupération enverra une SQLException.

**Retour d'une requête d'écriture**

Lorsque nous effectuons une modification sur une table de notre base de données via la méthode statement.executeUpdate(), celle-ci renvoie des informations différentes selon le type de la requête effectuée :

* l'exécution d'un INSERT renvoie 0 en cas d'échec de la requête d'insertion, et 1 en cas de succès.
* l'exécution d'un UPDATE ou d'un DELETE renvoie le nombre de lignes mises à jour ou supprimées.
* l'exécution d'un CREATE, ou de toute autre requête ne retournant rien, renvoie 0.

Voici par exemple comment récupérer le statut d'une requête d'insertion de données :

/\* Exécution d'une requête d'écriture \*/

int statut = statement.executeUpdate( "INSERT INTO Utilisateur (email, mot\_de\_passe, nom, date\_inscription) VALUES ('jmarc@mail.fr', MD5('lavieestbelle78'), 'jean-marc', NOW());" );

**Libération des ressources**

Tout comme il est nécessaire de fermer proprement une connexion ouverte, il est extrêmement recommandé de disposer proprement des objets Statement et ResultSet initialisés au sein d'une connexion :

Connection connexion = null;

Statement statement = null;

ResultSet resultat = null;

try {

/\*

\* Ouverture de la connexion, initialisation d'un Statement, initialisation d'un ResultSet, etc.

\*/

} catch ( SQLException e ) {

/\* Traiter les erreurs éventuelles ici. \*/

} finally {

if ( resultat != null ) {

try {

/\* On commence par fermer le ResultSet \*/

resultat.close();

} catch ( SQLException ignore ) {

}

}

if ( statement != null ) {

try {

/\* Puis on ferme le Statement \*/

statement.close();

} catch ( SQLException ignore ) {

}

}

if ( connexion != null ) {

try {

/\* Et enfin on ferme la connexion \*/

connexion.close();

} catch ( SQLException ignore ) {

}

}

}

Les objets utilisés doivent obligatoirement être fermés du plus récemment ouvert au plus ancien. Ainsi il faut commencer par fermer le ResultSet, puis le Statement et enfin l'objet Connection.

Il est possible de récupérer l’id d’une ligne en base générée par la méthode executeUpdate() :

/\* Exécution d'une requête d'écriture avec renvoi de l'id auto-généré \*/

int statut = statement.executeUpdate( "INSERT INTO Utilisateur (email, mot\_de\_passe, nom, date\_inscription) VALUES ('jmarc2@mail.fr', MD5('lavieestbelle78'), 'jean-marc', NOW());" , Statement.RETURN\_GENERATED\_KEYS);

/\* Récupération de l'id auto-généré par la requête d'insertion. \*/

resultat = statement.getGeneratedKeys();

/\* Parcours du ResultSet et formatage pour affichage de la valeur qu'il contient. \*/

while ( resultat.next() ) {

messages.add( "ID retourné lors de la requête d'insertion :" + resultat.getInt( 1 ) );

}

L'id créé par MySQL pour l'entrée insérée en base est présent dans le ResultSet retourné par statement.getGeneratedKeys().

La fonction MD5() dans notre requête SQL génère un hash si la chaîne contient 'null'.

**Les limites du système**

Il faut faire attention de bien former sa requête. En effet, une requête de type int statut = statement.executeUpdate( "INSERT INTO Utilisateur (email, mot\_de\_passe, nom, date\_inscription) " + "VALUES ('" + paramEmail + "', MD5('" + paramMotDePasse + "'), '" + paramNom + "', NOW());" ); permettra d’insérer en base des champs valant null même si nous avons spécifié le contraire à notre SGBD, puisque la requête contiendra ‘null’ et non pas null. Dans le cas de certains SGBD, un champ dans une requête « vide », à savoir un string de type «  », sera considéré de la même façon qu’un champ valant null.

**Les injections SQL**

Ce type « d’erreur » est fait lorsque l’utilisateur entre de mauvais paramètres, dans un formulaire par exemple. Il pourrait ainsi écrire « Marcel’ » dans le champ correspondant à son nom, ce qui ferait échouer la requête à cause de l’ajout d’une apostrophe. Cela peut aussi conduire à des cas de piratage.

Pour contrer ces problèmes, il existe des requêtes dites « préparées ».

**Les requêtes préparées**

Ce type de requêtes s’utilise via l’objet PreparedStatement. C’est une nouvelle interface qui implémente l'interface Statement et présente trois différence principale avec elle :

* l'utilisation de PreparedStatement peut permettre de pré-compiler une requête SQL.
* une requête SQL ainsi créée peut être paramétrée, c'est-à-dire contenir des trous ou jokers qui ne seront comblés que lors de son exécution.
* une requête SQL ainsi créée est protégée contre les injections SQL, et contre d'éventuelles erreurs sur les types des paramètres.

**1. Des requêtes pré-compilées**

Le premier avantage se situe au niveau des performances. Dès lors que nous souhaitons exécuter une requête d'un même objet Statement plusieurs fois, il devient intéressant d'utiliser un PreparedStatement à la place, afin de réduire le temps d'exécution. La principale fonctionnalité de ce nouvel objet est qu'à la différence d'un classique objet Statement, il prend en argument une requête SQL dès sa création. L'avantage, c'est que dans le cas de certains drivers, la requête SQL sera, dès la création de l'objet, directement envoyée au SGBD où elle sera compilée. Ainsi, un objet PreparedStatement ne contient pas simplement une requête SQL, mais une requête SQL pré-compilée. Cela signifie que lorsque ce PreparedStatement sera exécuté, le SGBD n'aura plus qu'à exécuter la requête sans avoir besoin de la compiler au préalable.

**2. Des requêtes paramétrées**

A travers un objet PreparedStatement, il est possible de créer des requêtes qui prennent en compte des paramètres. Ceux-ci sont représentés par des points d'interrogation ? dans la requête qui sert de modèle, et doivent être précisés avant l'exécution. Par exemple : SELECT \* FROM Utilisateur WHERE email = ?. L’objet PreparedStatement pré-compilera la requête et se chargera ensuite de remplacer le paramètre manquant par la valeur que nous souhaitons lui donner au moment de l'exécution.

**3. Des requêtes protégées !**

Le dernier avantage, c’est qu’en utilisant des requêtes préparées nous prévenons tout risque de failles SQL. Cette fonctionnalité, contrairement à l'étape de pré-compilation, est disponible quel que soit le driver JDBC utilisé.

**Initialisation de l'objet**

Contrairement à un Statement classique, nous allons désormais directement préciser la requête SQL dans cette nouvelle initialisation :

/\* Création de l'objet gérant la requête préparée définie \*/

PreparedStatement preparedStatement = connexion.prepareStatement( "SELECT id, email, mot\_de\_passe, nom FROM Utilisateur;" );

La seule différence est l'appel à la méthode prepareStatement() de l'objet Connection qui attend une requête SQL en argument.

**Exécution de la requête**

Puisque notre requête est déjà déclarée lors de l'initialisation, la seconde différence avec l'utilisation d'un Statement classique se situe au niveau de l'appel à la méthode d'exécution :

preparedStatement.executeQuery();

**Et avec des paramètres ?**

/\* Création de l'objet gérant les requêtes préparées \*/

preparedStatement = connexion.prepareStatement( "INSERT INTO Utilisateur (email, mot\_de\_passe, nom, date\_inscription) VALUES(?, MD5(?), ?, NOW());" );

/\*

\* Remplissage des paramètres de la requête grâce aux méthodes

\* setXXX() mises à disposition par l'objet PreparedStatement.

\*/

preparedStatement.setString( 1, paramEmail );

preparedStatement.setString( 2, paramMotDePasse );

preparedStatement.setString( 3, paramNom );

/\* Exécution de la requête \*/

int statut = preparedStatement.executeUpdate();

Pour récupérer l'id auto-généré après exécution d'une requête d'insertion, nous devons préciser dès la création d'un PreparedStatement si nous souhaitons récupérer l'id auto-généré ou non :

/\* Création d'un PreparedStatement avec renvoi de l'id auto-généré \*/

PreparedStatement preparedStatement = connexion.prepareStatement( "INSERT INTO Utilisateur (email, mot\_de\_passe, nom, date\_inscription) VALUES ('jmarc@mail.fr', MD5('lavieestbelle78'), 'jean-marc', NOW());", Statement.RETURN\_GENERATED\_KEYS );

Le reste ne change pas : il faut parcourir le ResultSet retourné par la méthode preparedStatement.getGeneratedKeys() et y récupérer le premier champ, tout comme nous l'avions fait avec un Statement classique.

## Le modèle DAO

**Inconvénients de la solution JDBC précédemment utilisée**

Si nous codons une vraie application de cette manière, nous lions alors très fortement le code responsable des traitements métier au code responsable du stockage des données. En fin de compte, il devient impossible d'exécuter séparément l'un ou l'autre.

* il est impossible de mettre en place des tests unitaires.
  + impossible de tester le code métier de l'application sans faire intervenir le stockage.
  + impossible de ne tester que le code relatif au stockage des données, obligation de lancer le code métier.
* il est impossible de changer de mode de stockage. Que ce soit vers un autre SGBD, voire vers un système complètement différent d'une base de données, cela impliquerait une réécriture complète de tout le modèle, car le code métier est mêlé avec et dépendant du code assurant le stockage.

**Isoler le stockage des données**

L'idée est qu'au lieu de faire communiquer directement nos objets métier avec la base de données, ceux-ci vont parler avec la couche DAO. Et c'est cette couche DAO qui va ensuite de son côté communiquer avec le système de stockage.



L'objectif du pattern DAO est de permettre de faire la distinction entre les données auxquelles nous souhaitons accéder, et la manière dont elles sont stockées.

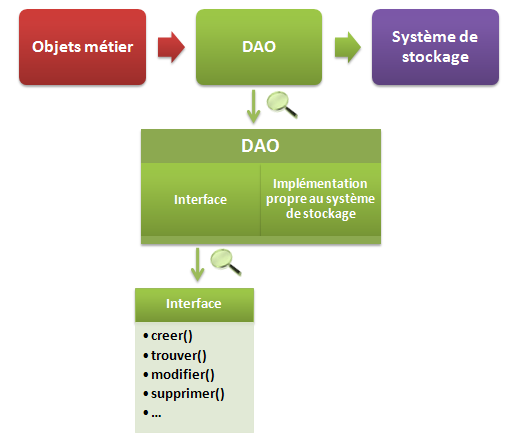
**Principe**

Le principe du pattern DAO est de séparer la couche modèle d'une application en deux sous-couches distinctes :

* une couche gérant les traitements métier appliqués aux données, souvent nommée couche service ou métier. Typiquement, tout le travail de validation réalisé dans nos objets **InscriptionForm** et **ConnexionForm** en fait partie.
* une couche gérant le stockage des données, logiquement nommée couche de données. Il s'agit là des opérations classiques de stockage : la création, la lecture, la modification et la suppression. Ces quatre tâches basiques sont souvent raccourcies à l'anglaise en CRUD.

Pour réaliser efficacement une telle opération, il est nécessaire d'encapsuler les exceptions spécifiques au mode de stockage dans des exceptions personnalisées et propres à la couche DAO. Dans notre cas par exemple, nous allons devoir faire en sorte que les exceptions propres à SQL ou à JDBC ne soient pas vues comme telles par nos objets métier, mais uniquement comme des exceptions émanant de la « boîte noire » qu'est notre DAO.

De même, il va falloir masquer le code responsable du stockage au code « extérieur », et l'exposer uniquement via des interfaces. Dans notre cas, il s'agira donc de faire en sorte que le code basé sur JDBC soit bien à l'abri dans des implémentations de DAO, et que nos objets métier n'aient connaissance que des interfaces qui les décrivent.



La couche modèle est constituée de la couche métier (en rouge) et de la couche données (en vert). La couche métier n'a connaissance que des interfaces décrivant les objets de la couche données. Ainsi, peu importe le système de stockage final utilisé, du point de vue du code métier les méthodes à appeler ne changent pas, elles seront toujours celles décrites dans l'interface. C'est uniquement l'implémentation qui sera spécifique au mode de stockage.

La couche DAO ne va pas seulement contenir les interfaces et implémentations des méthodes CRUD. Elle va également renfermer quelques classes dédiées à l'isolement des concepts liés au mode de stockage, comme les exceptions dont nous avons déjà brièvement parlé, mais également le chargement du driver et l'obtention d'une connexion. Pour ce faire, nous allons créer une Factory (une fabrique) qui sera unique dans l'application, ne sera instanciée que si les informations de configuration sont correctes et aura pour rôle de fournir les implémentations des différents DAO.

**Création**

Modification d’une colonne en base (type de données contenues, etc…) : ALTER TABLE Utilisateur CHANGE mot\_de\_passe mot\_de\_passe CHAR(56) NOT NULL;

Afin de cacher la nature du mode de stockage des données au reste de l'application, c'est une bonne pratique de masquer les exceptions spécifiques (celles qui surviennent au runtime, c'est-à-dire lors de l'exécution) derrière des exceptions propres au DAO. Typiquement, nous allons dans notre application avoir besoin de gérer deux types d'exceptions concernant les données :

* celles qui sont liées à la configuration du DAO et du driver JDBC.
* celles qui sont liées à l'interaction avec la base de données.

**Création d'un fichier de configuration**

Précédemment nous avions stocké en dur l'adresse et les identifiants de connexion à notre base de données dans le code Java de notre objet. Afin de séparer les informations de configuration du reste de l’application, il est recommandé de les placer dans un endroit accessible par le code Java et aisément modifiable à la main. Une bonne pratique très courante dans ce genre de cas est la mise en place d'un fichier properties, qui n'est rien d'autre qu'un fichier texte dont les lignes respectent un certain format. Puisque ce fichier contient des informations confidentielles, comme le mot de passe de connexion à la base de données, il est hors de question de le placer dans un répertoire public de l'application. Il est impératif de le placer sous **/WEB-INF** ou un de ses sous-répertoires !

**Création d'une Factory**

La **Factory** va être en charge de l'instanciation des différents DAO de notre application. Elle va :

* lire les informations de configuration depuis le fichier properties.
* charger le driver JDBC du SGBD utilisé.
* fournir une connexion à la base de données.

L’intérêt de cette approche, c’est que nous voulons instancier notre **DAOFactory** uniquement sous certaines conditions :

* si le fichier **dao.properties** est accessible.
* si les données qu'il contient sont valides.
* si le driver JDBC est bien présent dans l'application.

Pour cela, nous utiliserons une méthode static que nous pourrons appeler à tout moment, même lorsque notre objet ne sera pas encore instancié. (ceci nous évitera de passer par un « new » qui lui instancierait un nouvel objet à chaque appel).

Il faut dans un premier temps initialiser un objet Properties qui nous permettra de gérer notre fichier de configuration.

Properties properties = new Properties();

ClassLoader classLoader = Thread.currentThread().getContextClassLoader();

InputStream fichierProperties = classLoader.getResourceAsStream( FICHIER\_PROPERTIES );

properties.load(FICHIER\_PROPERTIES );

url = properties.getProperty( “PROPERTY\_URL” );

driver = properties.getProperty( “PROPERTY\_DRIVER” );

nomUtilisateur = properties.getProperty( « PROPERTY\_NOM\_UTILISATEUR » );

motDePasse = properties.getProperty( « PROPERTY\_MOT\_DE\_PASSE » );

La méthode getResourceAsStream() de l'objet ClassLoader se charge pour nous d'ouvrir le flux demandé et de retourner null en cas d'erreur. Nous récupérons le ClassLoader depuis le thread courant grâce à la méthode getContextClassLoader().

**Création de l'interface du DAO Utilisateur**

Il faut tout d’abord créer l’interface qui définira les méthodes à implémenter par notre DAO.

**Création de l'implémentation du DAO**

Il s'agit de la classe qui va manipuler la table **ciblée** de notre base de données. C’est elle qui contiendra les méthodes désignées par notre interface.

Pour avoir accès aux données nécessaires à la connexion à la BDD, l’implémentation de notre DAO doit avoir accès à notre Factory. Cela se fait via son constructeur :

private DAOFactory daoFactory;

UtilisateurDaoImpl( DAOFactory daoFactory ) {

this.daoFactory = daoFactory;

}

En général, toute méthode communiquant avec une base de données doit :

* initialiser une requête préparée avec des paramètres.
* récupérer une ligne d'une table et enregistrer son contenu dans un bean.
* fermer proprement les ressources ouvertes (Connection, PreparedStatement, ResultSet).

Pour accomplir ces différentes tâches, a priori nous allons avoir besoin de trois méthodes utilitaires :

* une qui récupère une liste de paramètres et les ajoute à une requête préparée donnée.
* une qui récupère un ResultSet et enregistre ses données dans un bean.
* une qui ferme toutes les ressources ouvertes.

**1. Initialisation d'une requête préparée**

/\*

\* Initialise la requête préparée basée sur la connexion passée en argument,

\* avec la requête SQL et les objets donnés.

\*/

public static PreparedStatement initialisationRequetePreparee( Connection connexion, String sql, boolean returnGeneratedKeys, Object... objets ) throws SQLException {

PreparedStatement preparedStatement = connexion.prepareStatement( sql, returnGeneratedKeys ? Statement.RETURN\_GENERATED\_KEYS : Statement.NO\_GENERATED\_KEYS );

for ( int i = 0; i < objets.length; i++ ) {

preparedStatement.setObject( i + 1, objets[i] );

}

return preparedStatement;

}

**2. Mapping d'un ResultSet dans un bean**

/\*

\* Simple méthode utilitaire permettant de faire la correspondance (le

\* mapping) entre une ligne issue de la table des utilisateurs (un

\* ResultSet) et un bean Utilisateur.

\*/

private static Utilisateur map( ResultSet resultSet ) throws SQLException {

Utilisateur utilisateur = new Utilisateur();

//«id» = nom du champ en base.

utilisateur.setId( resultSet.getLong( "id" ) );

utilisateur.setEmail( resultSet.getString( "email" ) );

//Etc…

return utilisateur;

}

**3. Fermeture des ressources**

Plutôt que de tout mettre dans une seule et même méthode, nous allons écrire une méthode pour fermer chaque type de ressource :

public static void fermetureSilencieuse( ResultSet resultSet ) {

if ( resultSet != null ) {

try {

resultSet.close();

} catch ( SQLException e ) {

System.out.println( "Échec de la fermeture du ResultSet : " + e.getMessage() );

}

}

}

**Chargement de la DAOFactory**

Notre **DAOFactory** est un objet que nous ne souhaitons instancier qu'une seule fois. Le démarrage de l'application semble donc l'instant approprié pour procéder à son initialisation. Dans une application Java classique, il nous suffirait de placer quelques lignes de code en tête de la méthode main(). Dans une application Java EE, la solution c'est l'interface ServletContextListener. Elle fournit une méthode contextInitialized() qui est appelée dès le démarrage de l'application, avant le chargement des servlets et filtres du projet.

**Création du Listener**

public class InitialisationDaoFactory implements ServletContextListener {

private static final String ATT\_DAO\_FACTORY = "daofactory";

private DAOFactory daoFactory;

@Override

public void contextInitialized( ServletContextEvent event ) {

/\* Récupération du ServletContext lors du chargement de l'application \*/

ServletContext servletContext = event.getServletContext();

/\* Instanciation de notre DAOFactory \*/

this.daoFactory = DAOFactory.getInstance();

/\* Enregistrement dans un attribut ayant pour portée toute l'application \*/

servletContext.setAttribute( ATT\_DAO\_FACTORY, this.daoFactory );

}

}

**Configuration du Listener**

Pour que notre **Listener** soit pris en compte lors du démarrage de notre application, il nous faut ajouter une section au fichier **web.xml** :

<listener>

<listener-class>com.sdzee.config.InitialisationDaoFactory</listener-class>

</listener>

**Utilisation depuis la servlet**

Notre **DAOFactory** étant prête à l'emploi dans notre projet, nous pouvons maintenant récupérer une instance de notre DAO Utilisateur depuis notre servlet d'inscription :

/\* Récupération d'une instance de notre DAO Utilisateur \*/

this.utilisateurDao = ( (DAOFactory) getServletContext().getAttribute( CONF\_DAO\_FACTORY ) ).getUtilisateurDao();

Pour rappel, une servlet n'est créée qu'une seule et unique fois par le conteneur, lors du démarrage de l'application ou bien lors de son premier accès (ceci dépendant de la présence ou non d'une section <load-on-startup> dans la déclaration de la servlet au sein du fichier **web.xml**). Lorsque la servlet est instanciée, sa méthode init() va être appelée, une seule et unique fois donc. Ainsi, nous avons la possibilité à travers cette méthode d'effectuer des tâches uniques, qui ne sont pas destinées à être lancées à chaque appel aux méthodes doGet() ou doPost() par exemple. Nous pouvons donc y placer la récupération de notre instance du DAO Utilisateur.

Un **test de régression** consiste à vérifier que les modifications apportées sur le code n'ont pas modifié le fonctionnement d’un page ou de l'application par rapport par rapport à ce qui existait avant cette modification.

**Création d’une contrainte de clé étrangère dans une table**

Pour ramener la clé « id » de la table « Client » dans la table « Commande » on écrit :

CREATE TABLE tp\_sdzee.Commande (

id INT( 11 ) NOT NULL AUTO\_INCREMENT ,

id\_client INT( 11 ) ,

PRIMARY KEY ( id ) , -- On indique que id sera une clé primaire

CONSTRAINT fk\_id\_client -- On donne un nom à notre clé étrangère

FOREIGN KEY (id\_client) -- Colonne sur laquelle on crée la clé étrangère

REFERENCES Client(id) -- Colonne de référence (de la table Client)

ON DELETE SET NULL -- Action à effectuer lors de la suppression du Client correspondant. Sans cette commande, on ne peut pas supprimer une ligne de Client si elle est liée à une ligne de Commande.

) ENGINE = INNODB;

## Gérer un pool de connexions

Remarquer : Le code servant précédemment d’exemple sur l’utilisation des Dao n’est pas optimal. En effet, il ne faut pas ouvrir/fermer une connexion à la BDD à chaque requête, car le coup de cette action est trop important en termes de performances. Il faudrait normalement se contenter d'ouvrir une seule connexion et de la partager à l'ensemble des méthodes d'un DAO.

**Réutilisation des connexions**

Afin d'alléger la charge que le SGBD doit actuellement supporter à chaque requête, nous allons utiliser une technique très simple : nous allons précharger un certain nombre de connexions à la base de données, et les réutiliser. L'expression employée pour nommer cette pratique est le « ***connection pooling*** », souvent très sauvagement francisée « ***pool de connexions*** ». Le pool de connexions va pré-initialiser un nombre donné de connexions au SGBD lorsque l'application démarre. Autrement dit, il va créer plusieurs objets **Connection** et les garder ouverts et bien au chaud.

Ensuite, le pool de connexions va se charger de distribuer ses objets **Connection** aux méthodes de l'application qui en ont besoin. Cela signifie que ces méthodes ne vont plus faire appel à DriverManager.getConnection(), mais plutôt à quelque chose comme pool.getConnection().

Enfin, puisque l'objectif du système est de partager un nombre prédéfini de ressources, un appel à la méthode Connection.close() ne devra pas provoquer la fermeture réelle d'une connexion. En lieu et place, c'est tout simplement un renvoi dans le pool de l'objet Connection qui va avoir lieu. De cette manière la boucle est bouclée : l'objet Connection inutilisé retourne à la source, et est alors prêt à être à nouveau distribué.

**Remplacement du DriverManager par une DataSource**

Jusqu'à présent, nous avons utilisé le DriverManager du package java.sql pour obtenir une connexion à notre base de données. Or il ne faut plus passer par cet objet. Sur une application Java classique, avec un utilisateur unique et donc une seule connexion vers une base de données, il convient très bien. Mais une application Java EE est multi-threads et multi-utilisateurs, et sous ces contraintes cet objet ne convient plus. Dans une application Java EE en conditions réelles, plusieurs connexions parallèles sont ouvertes avec la base de données et pour cela il n'est pas envisageable d'ouvrir des connexions à la volée pour chaque objet ou méthode agissant sur la base. Nous avons besoin d'une gestion des ressources plus efficace, et notre choix va se porter sur l'objet DataSource du nouveau package javax.sql. Il s'agit en réalité d'une interface qu'Oracle recommande d'utiliser en lieu de place du DriverManager, et ce peu importe le cas d'utilisation.

**Choix d'une implémentation**

Une DataSource n'est qu'une interface, il est donc nécessaire d'en écrire une implémentation. Il existe plusieurs bibliothèques, libres et gratuites, qui s’occupent de cette tâche. Par exemple Apache DBCP, BoneCP, c3p0, DBPool etc.

## Les annotations

Les annotations sont un système introduit avec Java EE 5 qui va nous permettre de nous débarrasser de notre fichier web.xml.

**Présentation**

Une annotation est une description d'un élément. Elle peut ainsi s'appliquer à un package, une classe, une interface, un constructeur, une méthode, un champ, un argument de méthode, une variable locale ou encore à une autre annotation.

**Écrire des méta-données**

Une méta-donnée désigne une donnée qui donne une information sur une autre donnée. Java a toujours proposé une forme de méta-programmation que nous connaissons tous : il s'agit de la Javadoc ! La Javadoc n'est, ni plus ni moins, qu'un outil permettant de décrire certaines portions de notre code. Dans ce cas, les méta-données sont utilisées par le développeur, qui va tout simplement lire les informations contenues dans la documentation.

Java EE a toujours proposé une forme de méta-programmation par le biais du fichier web.xml. Dans ce cas, les méta-données sont utilisées par le conteneur, qui analyse le contenu du fichier au démarrage du serveur afin d'identifier quels composants web agissent sur quelles requêtes.

**Simplifier le développement**

La raison d'être des annotations est de permettre de simplifier considérablement le processus de développement et de maintenance d'une application. Principalement parce qu'une annotation est écrite directement dans le code, au sein d'une classe ou d'une méthode. Ainsi, l'information associée à un élément du code est accessible et visible directement. Il n'est pas nécessaire de se référer à des données écrites dans des fichiers externes. De même, lors de la phase de développement il n'est pas nécessaire de créer et maintenir ces fichiers externes.

**Principe**

Une annotation est uniquement constituée d'un mot-clé précédé du signe @, par exemple l'annotation @Override.

@Override

public Utilisateur trouver( String email ) throws DAOException {

...

}

Il s'agit d'une annotation Java qui sert à décrire une méthode. Elle est constituée du caractère @ suivi du mot-clé Override, et elle est placée juste avant le début de la méthode :

* elle permet de préciser au compilateur qu'une méthode redéfinit une méthode d'une interface. En cas d'erreur - c'est-à-dire si le nom de la méthode ne correspond à aucune méthode d'aucune interface - alors le compilateur doit prévenir le développeur et éventuellement faire échouer la compilation.
* elle rend le code plus lisible, en différenciant les méthodes redéfinies des autres.

**Syntaxe avec paramètres**

Il existe une seconde forme d'annotation, qui attend en argument des paramètres.

@WebServlet( name="TestServlet", urlPatterns = {"/test", "/ok"} )

* une annotation peut attendre un ou plusieurs paramètres, séparés par une virgule et placés entre parenthèses juste après l'annotation. Dans cet exemple, les paramètres sont nommés **name** et **urlPatterns**.
* un paramètre peut attendre une ou plusieurs valeurs :
  + si une seule valeur est attendue, celle-ci est placée entre guillemets et liée au paramètre par le symbole =. Ici, le paramètre **name** attend une unique valeur définie à "TestServlet".
  + si plusieurs valeurs sont attendues, celles-ci sont placées entre guillemets, séparées par une virgule et l'ensemble ainsi formé est placé entre accolades, puis lié au paramètre par le symbole =. Ici, le paramètre **urlPatterns** reçoit deux valeurs définies respectivement à "/test" et "/ok".

**Avec l'API Servlet 3.0**

Nous allons aborder quelques annotations spécifiques à Java EE 6, introduites par l'API Servlet en version 3.0. Celles-ci font toutes partie du package javax.servlet.annotation.

**WebServlet**

Elle permet de déclarer une servlet : @WebServlet. Elle peut accepter tous les paramètres qu'il est possible de définir dans une section <servlet> ou <servlet-mapping> du fichier web.xml.

@WebServlet( name="Inscription", urlPatterns = "/inscription" )

public class Inscription extends HttpServlet { … }

* Il faut placer l'annotation juste avant la déclaration de la classe dans le code de la servlet.
* Il n'est pas nécessaire de préciser le contenu de <servlet-class> dans l'annotation : puisque celle-ci se trouve directement dans le code de la servlet, le compilateur sait déjà à quelle classe elle s'applique.

La propriété **name** de la servlet n'est plus utile. Dans le fichier **web.xml**, le champ <servlet-name> servait à établir un lien entre les sections <servlet> et <servlet-mapping>, mais maintenant que nous précisons directement l'url-pattern dans l'annotation de la servlet, son nom ne nous sert plus à rien. Par conséquent, il est possible d'écrire l'annotation sous sa forme la plus simple. Seul l'url-pattern sur lequel est mappée notre servlet est nécessaire à son bon fonctionnement

@WebServlet( "/inscription" )

public class Inscription extends HttpServlet { … }

**WebFilter**

Il existe une annotation qui permet de déclarer un filtre : @WebFilter.

@WebFilter( urlPatterns = "/restreint/\*" )

public class RestrictionFilter implements Filter { .. }

Ce filtre s’applique à toutes les url de type /restreint/.

**WebInitParam**

Il existe également une annotation qui ne peut être utilisée qu'au sein d'une annotation @WebServlet ou @WebFilter : @WebInitParam. Comme son nom l'indique, elle est destinée à remplacer la section <init-param> qu'il est possible d'inclure aux déclarations d'une servlet ou d'un filtre dans le fichier **web.xml**.

@WebServlet( urlPatterns = "/fichiers/\*", initParams = @WebInitParam( name = "chemin", value = "/fichiers/" ) )

public class Download extends HttpServlet { … }

**WebListener**

L'annotation dédiée aux Listener est @WebListener.

@WebListener

public class InitialisationDaoFactory implements ServletContextListener{…}

**MultipartConfig**

L'annotation dédiée au traitement des requêtes de type **Multipart** est @MultipartConfig. Nous pouvons y préciser quatre paramètres : **location**, **fileSizeThreshold**, **maxFileSize** et **maxRequestSize**.

@WebServlet( urlPatterns = "/upload", initParams = @WebInitParam( name = "chemin", value = "/fichiers/" ) )

@MultipartConfig( location = "c:/fichiers", maxFileSize = 10 \* 1024 \* 1024, maxRequestSize = 5 \* 10 \* 1024 \* 1024, fileSizeThreshold = 1024 \* 1024 )

public class Upload extends HttpServlet { … }

Nous avons besoin de deux annotations différentes appliquées sur la même classe pour remplacer intégralement la précédente déclaration de la servlet ! En effet, alors que @WebInitParam est incluse dans le corps de l'annotation @WebServlet, l'annotation @MultipartConfig est, quant à elle, indépendante et doit donc être spécifiée à part. Côté syntaxe :

* Lorsqu'un paramètre attendu est de type numérique, il ne faut pas l'entourer de guillemets comme nous le faisions jusqu'alors pour tous nos paramètres de type String.
* Lorsqu'un paramètre attendu est de type numérique, il est possible d'utiliser des opérateurs mathématiques simples dans sa valeur. Par exemple l'opérateur de multiplication \*.

Le fichier **web.xml** est encore utile pour plusieurs raisons :

**1. Déclarer la version de l'API servlet**

La première utilisation est la déclaration de la version de l'API Servlet requise par notre application, autrement dit la balise <web-app> :

<?xml version="1.0" encoding="UTF-8"?>

<web-app

xmlns="http://java.sun.com/xml/ns/javaee"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd"

version="3.0">

Les annotations ne sont disponibles qu'à partir de Java EE 6, il faut donc impérativement déclarer l'API Servlet 3.0 pour pouvoir les utiliser dans notre code.

**2. Spécifier un ordre de filtrage**

Pour organiser nos filtres et définir un ordre d'exécution précis, nous devons continuer à les déclarer dans notre fichier **web.xml**. Toutefois nous pouvons utiliser des annotations pour remplacer les sections <filter> et nous contenter d'écrire les sections <filter-mapping> du fichier **web.xml**.

**3. Exécuter un filtre externe**

Il existe un cas particulier dans lequel nous ne pourrons pas utiliser l'annotation @WebFilter : lorsque le filtre que nous appliquons ne fait pas partie de notre application ! Sans accès au code source, nous n’avons pas vraiment le choix.

Par exemple, dans notre projet nous utilisons le filtre natif de Tomcat nommé **Set Character Encoding**, afin de compléter la gestion de l'UTF-8. Eh bien pour ce cas précis, nous n'avons pas d'autre choix que de déclarer le filtre à la main dans le **web.xml** de notre application.

**4. Surcharger les valeurs définies dans une annotation**

Les valeurs précisées dans toutes nos annotations sont en réalité considérées par le conteneur comme des valeurs par défaut. En d'autres termes, si nous déclarons à la fois une annotation et une section dans le fichier **web.xml**, alors les données de la section seront prises en compte quoi qu'il arrive.

Cela signifie que par exemple si nous mappons en même temps une servlet sur une URL depuis une section dans le **web.xml**, et sur une autre URL depuis une annotation, alors la servlet sera en fin de compte mappée sur les deux URL.

Par ailleurs, dans le cas de paramètres à valeur unique comme les @WebInitParam, en cas de double définition c'est la valeur contenue dans le **web.xml** qui sera prise en compte, et celle contenue dans l'annotation sera ignorée.

## La persistance des données avec JPA

**Généralités**

Au sens général, il s’agit du terme utilisé pour décrire le fait de stocker des données d’une application de manière… persistante ! Autrement dit, de les sauvegarder afin qu’elles ne disparaissent pas lorsque le programme se termine.

Dans notre exemple - une très petite application - nous ne nous sommes pas inquiétés de devoir réécrire un DAO pour chaque table manipulée. Cependant, cela ne devient plus viable pour de très grosses applications, contenant des centaines de tables. Agir ainsi est une perte de temps : dans chacune de nos classes, nous utilisons un code source très similaire pour récupérer les données des tables et les transformer en objets. Concrètement, mis à part le SQL, l'affectation des paramètres sur l'objet PreparedStatement et la récupération des résultats et leur affectation sur le bean, rien ne change : la structure générale du code reste toujours la même !

La solution pour pallier à cela fournie par Java EE est le JPA. Littéralement « Java Persistence API », il s’agit d’un standard faisant partie intégrante de la plate-forme Java EE, une spécification qui définit un ensemble de règles permettant la gestion de la correspondance entre des objets Java et une base de données, ou autrement formulé la gestion de la persistance. Ce mécanisme qui gère la correspondance entre des objets d’une application et les tables d’une base de données se nomme ORM, pour « Object-Relational Mapping ».

Et Hibernate, TopLink, EclipseLink, OpenJPA… ? Ce sont des « frameworks ORM ». Lorsqu’on parle de JPA, il s’agit uniquement d’une API, c’est-à-dire une description d'un comportement à suivre, en l'occurrence pour respecter un standard en place. Hibernate etc sont des implémentations du standard JPA.

**Des EJB dans un conteneur**

Le concept mère qui se cache derrière JPA, c’est le fameux EJB ou « Enterprise JavaBean ».

**Principe général**

Les EJB sont des objets présentant une caractéristique bien particulière : **ils sont gérés par le conteneur**. Attention, quand on parle ici de conteneur il n’est plus question du simple conteneur de servlets que nous avons utilisé jusqu’à présent ! Non, il s’agit bien ici de « **conteneur EJB** », un élément dont le travail est de gérer entièrement le cycle de vie des EJB qu’il contient.

Remarque : Par défaut, tout serveur d’applications Java EE au sens strict du terme contient un conteneur EJB. En revanche, pour ce qui est des serveurs légers comme Tomcat, ce n’est pas le cas ! Ainsi, nous ne pouvons pas manipuler d’EJB depuis Tomcat sans y ajouter un tel conteneur.

**JPA, ou les EJB Entity**

Il existe deux types d’EJB : les EJB Entity et les EJB Session. Celui qui sert de pilier à JPA est le premier, et le plus simple à appréhender : l’EJB Entity. **C’est lui qui définit quelles données doivent être sauvegardées, et c’est à travers lui qu’est effectuée la correspondance entre un objet et une table d’une base de données.** En apparence, c’est un objet qui ressemble beaucoup à un simple Javabean, dans lequel on ajoute simplement quelques annotations. Ces informations sont ici utilisées pour informer le conteneur d’EJB de la manière dont l’objet devra être géré.

**Un gestionnaire d’entités**

Définir quelles données doivent être sauvegardées en construisant des entités est une première étape. Mais elle ne servirait à rien s’il n’existait pas un système définissant **comment** ces données doivent être sauvegardées.

Les méthodes permettant d’établir une connexion avec la base de données et de gérer la persistance se trouvent dans un objet particulier nommé **EntityManager**, ou gestionnaire d'entités. **Il s’agit, là encore, d’un objet dont le cycle de vie est géré par le conteneur.** Cet objet se base sur des informations que le développeur doit saisir dans un fichier de configuration externe, que nous allons découvrir prochainement et qui n’est rien d’autre qu’un simple fichier XML.

**Mise en place**

**L’ajout d’un** dossier nommé **META-INF d**ans le répertoire **src** du projet est nécessaire. Il faut y créer un fichier nommé **persistence.xml**. Voici son code de base :

**<?xml version="1.0" encoding="UTF-8"?>**

**<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence"**

**xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"**

**xsi:schemaLocation="http://java.sun.com/xml/ns/persistence**

**http://java.sun.com/xml/ns/persistence/persistence\_2\_0.xsd">**

**</persistence>**

Dans ce fichier nécessaire au bon fonctionnement de JPA, nous allons définir deux choses :

* Les informations de connexion à la base de données dont auront besoin les **EntityManager**.
* Une unité de persistance pour chacune de nos entités. Une unité de persistance est un contexte, une section qui permet de définir à quelles classes va s’appliquer un **EntityManager**, sur quelle connexion il va se baser et comment il va dialoguer avec la BDD pour persister les données.

Par exemple, pour mettre en place une unité que nous allons nommer "bdd\_sdzee\_PU" qui va s'appliquer à notre entité **Utilisateur** et qui va se baser sur la connexion nommée **jdbc/bonecp\_resource** il faut écrire :

<persistence-unit name="bdd\_sdzee\_PU" transaction-type="JTA">

<jta-data-source>jdbc/bonecp\_resource</jta-data-source>

<class>com.sdzee.entities.Utilisateur</class>

</persistence-unit>

**Création d’une entité Utilisateur**

Nous allons ici inclure quelques annotations propres à JPA dans un bean. Pour commencer, nous allons indiquer à notre serveur, plus précisément à notre conteneur, que notre bean **Utilisateur** va devenir un EJB Entity. Pour cela, nous devons l'annoter avec @Entity :

@Entity

public class Utilisateur {...}

En théorie, il faudrait également indiquer via l'annotation @Table(name = "Utilisateur") que l’entité est liée à la table nommée **Utilisateur** dans notre base. Toutefois, le comportement par défaut du conteneur, en l’absence de cette précision, est de considérer que le nom de la table est identique à celui de la classe.

Il est ensuite nécessaire de définir la correspondance entre les données de l’entité **Utilisateur** et les données qui se trouvent dans la table. Nous allons donc devoir annoter les attributs de notre entité. Là encore, par défaut le conteneur sait identifier lui-même que tel attribut correspond à tel champ, à partir du moment où les champs portent le même nom que les attributs. Lorsque ce n’est pas le cas par contre, il est nécessaire d’ajouter une annotation @Column(name = "…") :

@Column( name = "mot\_de\_passe" )

private String motDePasse;

Enfin, pour préciser qu'un attribut est associé à la clé primaire de la table, il est nécessaire de l’annoter avec @Id. En outre, pour que le conteneur puisse correctement gérer l’id auto-généré lors de la création d’un utilisateur en base, il faut ajouter l’annotation @GeneratedValue :

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY )

private Long id;

Voilà tout ce qu’il est nécessaire de modifier pour transformer notre simple JavaBean en un EJB Entity, prêt à être géré par notre conteneur.

**Création d’un EJB Session**

Un EJB Session est un objet qui donne accès aux services & méthodes qu'il contient. Il existe deux types d’EJB Session : ceux qui sont **Stateless**, et ceux qui sont **Stateful**.

**Stateless : authentification sur un site marchand**

Premier exemple, un système d'authentification à un site marchand. Dans sa version la plus simple, pour réaliser cette tâche il suffit d'un objet contenant une méthode qui compare un couple d'identifiants passés en paramètres à ceux qui sont stockés en base, et qui retourne un code de succès ou d'erreur en retour. Cet objet ne présente donc pas de risques liés aux multiples Threads, et une seule instance peut très bien être partagée par de multiples requêtes issues de clients différents.

De manière concise, voici les propriétés d'un EJB Stateless :

* Aucune donnée n'est retenue ni enregistrée, c'est-à-dire qu'aucun état n'est retenu. On dit alors que l'objet est sans état, ou Stateless.
* Aucun mécanisme ne garantit que deux appels consécutifs à une méthode d'un tel EJB visent une seule et même instance.
* Les accès concurrents sont impossibles, mais le système est threadsafe tout de même puisque le conteneur envoie les requêtes simultanées vers des instances différentes du même EJB.

**Stateful : panier sur un site marchand**

Second exemple, un système de panier sur un site marchand. Pour réaliser une telle tâche, nous avons besoin d'un objet qui soit capable de retenir les commandes effectuées par un client, et qui puisse être réutilisé par ce même client pendant sa session d'utilisation, sans risque qu'un autre client puisse y accéder. Cet objet présente donc un risque lié aux multiples Threads : une même instance ne doit surtout pas être partagée par plusieurs requêtes issues de clients différents, sans quoi le panier perd tout son intérêt.

De manière concise, voici les propriétés d'un EJB Stateful :

* Des données sont retenues dans l'objet après un appel, c'est-à-dire qu'il conserve un état. On dit alors que l'objet est à état, ou Stateful.
* L’accès à une instance de l’EJB est réservé à un seul client à la fois.
* Les accès concurrents sont impossibles, le conteneur gère une liste d’attente en cas de tentatives simultanées.

Dans l’application exemple, c’est notre DAO que nous allons transformer en EJB Session de type Stateless. Il s’agit bien là d’une classe ne contenant que des méthodes d’interaction avec JDBC. Cette fois, nous n’allons plus converser avec JDBC. Cela, c’est le conteneur qui va s’en occuper pour nous grâce à JPA. Dorénavant, nous allons simplement demander à notre EntityManager de donner des ordres à notre base de données via ses méthodes persist(), find(), remove(), etc.

@Stateless

public class UtilisateurDao {

private static final String JPQL\_SELECT\_PAR\_EMAIL = "SELECT u FROM Utilisateur u WHERE u.email=:email";

private static final String PARAM\_EMAIL = "email";

// Injection du manager, qui s'occupe de la connexion avec la BDD

@PersistenceContext( unitName = "bdd\_sdzee\_PU" )

private EntityManager em;

// Enregistrement d'un nouvel utilisateur

public void creer( Utilisateur utilisateur ) throws DAOException {

try {

em.persist( utilisateur );

} catch ( Exception e ) {

throw new DAOException( e );

}

}

// Recherche d'un utilisateur à partir de son adresse email

public Utilisateur trouver( String email ) throws DAOException {

Utilisateur utilisateur = null;

Query requete = em.createQuery( JPQL\_SELECT\_PAR\_EMAIL );

requete.setParameter( PARAM\_EMAIL, email );

try {

utilisateur = (Utilisateur) requete.getSingleResult();

} catch ( NoResultException e ) {

return null;

} catch ( Exception e ) {

throw new DAOException( e );

}

return utilisateur;

}

}

Pour préciser à notre conteneur que l'objet est un EJB de type Stateless, il suffit d'une annotation @Stateless placée avant la déclaration de la classe.

De même, il est simple d’**injecter** dans notre EJB une instance d'un **EntityManager** dépendant d'une unité de persistance, via l'annotation @PersistenceContext( unitName = "..." ). Nous reconnaitrons ici le nom de l'unité que nous avons déclarée dans le fichier **persistence.xml** : **bdd\_sdzee\_PU**.

Injecter est le terme utilisé pour décrire le fait que le cycle de vie de l’objet annoté est géré par le conteneur. En d’autres termes, cela signifie que nous n’avons plus besoin de nous occuper de la création ni de l’initialisation de l’objet, c’est le conteneur qui va le faire pour nous ! En effet comme vous pouvez le voir dans ce code, nous faisons appel à des méthodes de l'objet **em** mais à aucun moment nous ne créons ni n'initialisons une instance d'objet, nous nous contentons uniquement de sa déclaration et de son annotation !

Pour créer un objet en base, persist() est la méthode de l'**EntityManager** qui va se charger de tout pour nous. Nous avons uniquement besoin de lui transmettre notre entité **Utilisateur** et tout le reste s'effectue derrière les rideaux !

La seule requête de lecture qu'il est possible de faire par défaut, c'est la plus basique, à savoir la recherche d’un élément en se basant sur sa clé primaire. En ce qui concerne la méthode de récupération d'un utilisateur en se basant sur son adresse email, nous devons écrire un minimum de SQL pour parvenir à nos fins. Toutefois, il n’est plus nécessaire d’écrire du SQL directement via JDBC comme nous le faisions dans nos DAO. Dorénavant, JPA nous facilite la tâche en encadrant notre travail, et nous propose pour cela deux méthodes différentes : le langage JPQL, et le système Criteria. Le plus simple à prendre en mains est le JPQL, car il ressemble très fortement au langage SQL. Cependant, il présente une différence majeure : il n'est pas utilisé pour interagir avec la base de données, mais avec les entités de notre application. Et c'est bien là l'objectif de JPA : découpler l'application du système de stockage final.

Pour créer notre requête, il suffit d'appeler la méthode createQuery() de l'**EntityManager**. Analysons la syntaxe de notre requête :

SELECT u FROM Utilisateur u WHERE u.email=:email

* Dans la section FROM, nous définissons le type de l'objet ciblé par la requête, **Utilisateur**, et son alias dans la requête courante, ici **u**.
* Dans la section SELECT, nous précisons simplement que nous souhaitons récupérer l'entité **Utilisateur** en notant son alias **u**.
* Dans la clause WHERE, nous ciblons l'attribut **email** de l'entité **Utilisateur** en concaténant simplement l'alias et le nom du champ, ici **u.email**. Enfin, nous spécifions que nous allons fournir un paramètre à la requête et le nommons **email** via la notation **:email**.

Le paramètre sera fourni à la requête via la méthode setParameter(). Enfin, il faut réaliser l'appel qui va déclencher la requête JPQL via la méthode getSingleResult().Celle-ci retourne un unique résultat. L’appel à cette méthode doit être entouré d’un bloc try/catch car celle-ci retourne une **NoResultException** lorsque rien n'est trouvé (ainsi nous pouvons faire retourner null dans le bloc catch dans le cas où ce type d’exception est retourné). Il existe également la méthode getResultList() si la requête utilisée peut ramener plusieurs résultats.

**Modification de la servlet**

Auparavant, nous récupérions une instance de DAO lors de la création de la servlet, depuis une Factory qui était quant à elle initialisée au lancement de l’application. Avec JPA, nous allons pouvoir nous débarrasser de tout cet arsenal, et par conséquent nous n'allons plus avoir besoin de la méthode init() dans notre servlet. Le conteneur va créer un pool d'EJB, similaire sur le principe à notre pool de connexions. Lorsque plusieurs requêtes quasi simultanées émanant de clients différents seront redirigées vers notre servlet, le conteneur va transmettre une instance différente de notre EJB à chacune d'elles tant qu'il en restera dans son pool. Quand le pool sera vide, il gérera une file d'attente et attendra le retour des instances déjà distribuées pour servir les requêtes en attente. C'est à l'aide d’une simple annotation que nous allons **injecter** notre EJB directement dans notre servlet : @EJB. Nous ne nous occupons plus de son cycle de vie, nous déléguons entièrement ce travail au conteneur.

// Injection de notre EJB (Session Bean Stateless)

@EJB

private UtilisateurDao utilisateurDao;

//Utilisation de notre utilisateurDao

InscriptionForm form = new InscriptionForm( utilisateurDao );

**Attention :** l'injection d'EJB dans une servlet : **elle est exclusivement réservée aux EJB de type Stateless**. Il n'existe qu'une seule et unique instance de notre servlet, que le conteneur initialise au chargement de notre application. Autrement dit, les variables d'instances sont partagées entre toutes les requêtes qui sont redirigées vers notre servlet ! Voilà pourquoi il ne faut jamais procéder à l'injection d'objets Stateful dans une servlet.

**Analyse des requêtes SQL générées**

Il est possible d’analyser les requêtes SQL effectuées par un **EntityManager** en activant l'écriture de logs depuis son unité de persistance dans le fichier **persistence.xml**.

<persistence-unit name="bdd\_sdzee\_PU" transaction-type="JTA">

<jta-data-source>jdbc/bonecp\_resource</jta-data-source>

<class>com.sdzee.entities.Utilisateur</class>

<properties>

<property name="eclipselink.logging.level.sql" value="FINE"/>

<property name="eclipselink.logging.parameters" value="true"/>

</properties>

</persistence-unit>

**Aller plus loin**

Il reste un nombre conséquent d’informations importantes et d’outils très pratiques que nous n’allons pas aborder ici :

* Avec un framework ORM, il est possible de générer automatiquement le code des entités depuis les tables d’une base de données ! Encore moins de code à écrire, et donc moins de travail pour le développeur.
* Avec un framework ORM, il est possible de générer automatiquement les tables d’une base de données depuis les entités d’une application ! Eh oui, la réciproque est également possible : si un développeur préfère réaliser un modèle objet plutôt qu’un modèle de données, il lui est possible de ne pas avoir à s’occuper lui-même de la création des tables !
* Avec un framework ORM, la gestion des transactions est automatisée.

**ORM, ou ne pas ORM ?**

Hibernate, EclipseLink et consorts sont des frameworks ORM. Tentons de mettre au clair quand utiliser un ORM, et quand s’en passer.

**Génération des requêtes**

Si les requêtes créées par une solution de persistance sont parfaites pour les actions de type CUD (Create, Update, Delete), elles sont plus discutables pour les requêtes de type R (les SELECT et toutes leurs clauses et jointures éventuelles).

**Perte de contrôle sensible sur le SQL utilisé pour communiquer avec la BDD**

De par sa nature, un framework ORM masque les communications avec la BDD. S'il est possible de paramétrer finement une solution de persistance afin qu'elle produise des requêtes SQL au plus proche de ce que l'on aurait fait à la main, le jeu n'en vaut pas toujours la chandelle. Il faut parfois passer plus de temps à paramétrer le tout qu'à le faire soi-même.

**Performances attendues**

Dans des projets où la performance est la préoccupation principale, il est plus sage de ne pas utiliser d'ORM

**Possibilité de mixer avec et sans ORM**

Dans l'absolu, il est possible d'utiliser JPA pour tout ce qui est simple et barbant - les actions de type CUD et certaines actions de lecture - et du SQL fait main pour le reste. Il suffit pour cela d'une couche d'abstraction, un DAO par exemple, et il devient alors aisé de ne conserver que les avantages des deux solutions.

## Introduction aux frameworks MVC

Nous avons appris à limiter nos efforts sur la couche d'accès aux données avec JPA, nous allons maintenant découvrir comment alléger notre charge de travail sur les couches vue et contrôleur grâce aux **frameworks MVC**.

**Rappel concernant MVC**

Revenons l'espace d'un court instant sur les objectifs du pattern MVC. Il s'agit d'un modèle de conception, une bonne pratique qui décrit comment le code d'une application doit être organisé. Dans une application Java EE, il découpe littéralement le code en plusieurs couches, chacune étant chargée d'assurer des tâches bien définies. Les principaux avantages d'un tel pattern sont :

* La clarté introduite par un découpage clair et surtout standard des différentes sections d'une application permet une maintenance du code bien plus aisée que si le code ne respectait aucune règle préétablie.
* Le découpage, et donc l'isolement des différentes tâches au sein d'une application, permet une meilleure répartition du travail entre les différents profils de développeurs. Le cas le plus souvent mis en avant est celui du designer web, qui peut ainsi ne s'occuper que de la vue sans avoir à se soucier de ce qui se passe derrière, ni même de comment cela se passe.

À ces avantages s'oppose toutefois un inconvénient majeur : puisqu'il y a nécessité de délimiter clairement les couches d'une application, il faut fatalement écrire davantage de code, et la structure ainsi définie impose des répétitions de code fréquentes.

**Qu'est-ce qu'un framework MVC ?**

La brique de base de la plate-forme Java EE est la servlet. Tout passe par elle, et même nos pages JSP sont transformées en servlets derrière les rideaux. Eh bien un framework MVC n'est rien d'autre qu'une surcouche à cette technologie de base. En masquant ces rouages les plus basiques, une telle solution facilite le découpage et la gestion du code d'une application. Elle intervient sur le cycle de vie d'une requête dans l'application, et prend en quelque sorte la main sur son cheminement. Voilà pourquoi on qualifie souvent les frameworks MVC de frameworks d'inversion de contrôle, parfois abrégé en IoC.

**À quels besoins répond-il ?**

L'objectif premier d'un framework MVC est de simplifier le flot d'exécution (ou workflow) d'une application, c'est-à-dire de :

* Limiter les répétitions de code impliquées par la mise en place d'un découpage du code.
* Réaliser une partie du travail redondant à la place du développeur, en effectuant des tâches génériques derrière les rideaux, de manière transparente.
* Contraindre le développeur à respecter une organisation clairement définie. Alors qu'il reste libre d'implémenter comme bon lui semble le pattern MVC lorsqu'il travaille à la main, il doit se conformer à un format plus ou moins strict lorsqu'il fait usage d'un framework.
* Par corollaire du premier point, rendre le développement de certains aspects plus rapide, permettant ainsi au développeur de se concentrer sur le cœur de l'application.

**Quand utiliser un framework MVC, et quand s'en passer ?**

* **L'envergure du projet** : si l'application développée est relativement petite, il n'est probablement pas nécessaire de mettre en place un framework, et le développement MVC "à la main" peut convenir.
* **Le temps d'apprentissage** : si vous ou votre équipe n'avez que peu de connaissances sur un framework MVC, alors il faut peser le pour et le contre entre le temps gagné durant le développement grâce à son utilisation, et le temps perdu en amont pour apprendre et assimiler la technologie.
* **Le contexte du projet** : si des contraintes sont formulées au niveau du serveur et des composants utilisables par un projet, et éventuellement au niveau des performances ou du niveau de scalabilité attendus, alors il faut déterminer si l'utilisation d'un framework rentre dans ce cadre ou non, et vérifier qu'il respecte bien les contraintes énoncées.

**Framework MVC basé sur les requêtes**

La première grande catégorie de frameworks MVC qualifie ceux qui se basent **sur les requêtes**. On parle également de frameworks basés **sur les actions**. Ce sont des solutions qui interviennent directement sur le cycle de vie d'une requête au sein de l'application (on rejoint ici cette histoire de "prise de contrôle").

**Principe**

Il s'agit d'un framework web qui prend en entrée une requête issue d'un client, qui détermine ce que le serveur doit en faire et qui renvoie enfin au client une réponse en conséquence. Le flot d'exécution peut donc être qualifié de linéaire, et reprend ce que nous avons jusqu'à présent mis en place à la main dans nos exemples.

Ainsi, le développeur doit penser en termes d'actions, et associe naturellement ce que demande l'utilisateur (sa requête) à ce que l'utilisateur reçoit (la réponse). Concrètement, le développeur écrit des classes qui représentent des actions effectuées par l'utilisateur, comme par exemple "Passer une commande" ou "Voir les détails du client". Ces classes sont chargées de récupérer les données transmises via la requête HTTP, et de travailler dessus. En somme, avec un tel framework le développeur travaille toujours de près ou de loin sur la paire requête/réponse.

Le gros changement à noter est la mise en place d'une **servlet unique** jouant le rôle d'aiguilleur géant (ou Front Controller). Celle-ci se charge de déléguer les actions et traitements au modèle métier en se basant sur l'URL de la requête et sur ses paramètres. Le développeur peut alors travailler directement sur les objets HttpServletRequest et HttpServletResponse bruts dans le modèle, ou bien utiliser le système de mapping fourni par le framework. Ce système permet au développeur de confier au framework les tâches de regroupement, conversion et validation des paramètres de requête, et si nécessaire de mise à jour des valeurs du modèle de données, avant d'invoquer les actions métier. Enfin, il doit toujours écrire lui-même les pages - bien souvent des pages JSP - en charge de créer les réponses à renvoyer au client, et il jouit donc d'une liberté totale sur le rendu HTML/CSS/JS de chaque vue.

**Solutions existantes**

* **Spring**, solution très répandue dont le spectre s'étend de la simple gestion des requêtes à l'ensemble du cycle de vie de l'application, bien souvent couplée à Hibernate pour la persistance des données.
* **Struts**, solution éditée par Apache dont la gloire appartient au passé. Le framework a changé du tout au tout avec la sortie d'une seconde mouture qui se nomme Struts 2, mais qui à part son titre n'a rien de similaire à Struts, premier du nom. Par ailleurs, la communauté maintenant le projet semble moins active que les concurrentes.

**Framework MVC basé sur les composants**

La seconde grande catégorie de frameworks MVC qualifie ceux qui se basent non pas sur les requêtes, mais sur les composants. À la différence de ceux basés sur les requêtes, les frameworks basés sur les composants découpent logiquement le code en "composants", masquant ainsi le chemin d'une requête au sein de l'application. Ils essaient en quelque sorte d'abstraire les concepts de requête et réponse, et de traiter une application comme une simple collection de composants qui présentent leur propre méthode de rendu et des actions pour effectuer des tâches. On parle en anglais d'action-based frameworks et de component-based frameworks. De telles solutions ont pour objectif secondaire de rendre la maîtrise des technologies de présentation web habituellement requises superflues, notamment HTML, CSS et JS.

**Principe**

Dans le MVC basé sur les composants, une unique servlet jouant le rôle de Front Controller va elle-même regrouper, convertir et valider les paramètres de requête, et mettre à jour les valeurs du modèle. Le développeur n'a ainsi à se soucier que des actions métier. La façon dont le contrôleur regroupe/convertit/valide/met à jour les valeurs est définie dans un unique endroit, la vue. Puisque c'est impossible à réaliser à l'aide de simple HTML "pur", un langage similaire spécifique est requis pour y parvenir. Dans le cas de JSF, c'est un langage basé sur du XML (XHTML). Vous utilisez du XML pour définir les composants de l'interface utilisateur, qui eux contiennent des informations sur la manière dont le contrôleur doit regrouper/convertir/valider/mettre à jour les valeurs et générer lui-même le rendu HTML.

Les avantages et inconvénients doivent maintenant être clairs :

* Avec un framework MVC basé sur les requêtes, il faut écrire plus de code pour parvenir à ses fins. Cependant, nous avons un meilleur contrôle sur le processus, c'est-à-dire sur le cheminement d'une requête au sein de l'application, et sur le rendu HTML/CSS/JS.
* Avec un framework MVC basé sur les composants, nous n'avons pas besoin d'écrire autant de code nous-mêmes. Cependant, nous avons moins de possibilités de contrôle sur le processus et le rendu HTML/CSS/JS.

Donc, si nous souhaitons réaliser des choses qui s'écartent un peu de ce que décrit le standard, nous perdrons beaucoup plus de temps si vous utilisez un framework MVC basé sur les composants.

**Solutions existantes**

* **JSF**, ou "**Java Server Faces**" : il s'agit de la solution standard intégrée à Java EE 6.
* **Wicket** : solution éditée par Apache, elle est très en vogue grâce à sa relative simplicité.

**Les "dissidents"**

Il existe plusieurs solutions qui ne se classent dans aucune de ces deux catégories, mais qui sont tout de même utilisées pour développer des applications web, et qui peuvent être utilisées en suivant le pattern MVC :

* **GWT**, solution éditée par Google pour la création d'applications web de type client-serveur.
* **Play!**, un framework plus global qui se place au même niveau que Java EE lui-même, et qui permet de développer avec les langages Java ou Scala.

## Premiers pas avec JSF

JSF (Java Server Faces) est un **framework MVC basé sur les composants**. Il est construit sur l'API Servlet et fournit des composants sous forme de bibliothèques de balises ressemblant très fortement à la JSTL. Celles-ci peuvent être utilisées dans des pages JSP comme dans toute autre technologie de vue basée sur le Java, car le framework JSF ne limite pas le développeur à une technologie particulière pour la vue. Cependant, il existe une technologie relativement récente baptisée la Facelet, qui fait partie du standard et qu'il est recommandé d'utiliser dès lors que l'on travaille avec JSF.

JSF offre de grandes capacités de création de templates (ou gabarits), tels que les composants composites (composants purement XML). La création de templates est la possibilité de découper une page en plusieurs composants indépendants, assemblés ensuite pour former une page finale. JSF permet de gérer les événements et la validation des données saisies par l'utilisateur, de contrôler la navigation entre les pages, de créer des vues accessibles, de faciliter l'internationalisation des pages, etc. Puisqu'il est basé sur les composants et non sur les requêtes, il permet de gérer un état pour une vue donnée.

**Principe**

JSF propose, en guise de contrôleur unique du cycle de vie du traitement des requêtes, la FacesServlet. Il n'y a qu'une seule servlet chargée d'aiguiller l'intégralité des requêtes entrantes vers les bons composants, et ce pour l'application tout entière ! Cette architecture porte un nom, il s'agit du pattern Front Controller, qui lui-même est une spécialisation du pattern Médiateur. JSF nous évite d'avoir à écrire le code responsable du regroupement des saisies utilisateurs (paramètres de requêtes HTTP), de leur conversion & validation, de la mise à jour des données du modèle, de l'invocation d'actions métiers et de la génération de la réponse. Ainsi, nous nous retrouvons uniquement avec une page JSP ou une page XHTML (une Facelet) en guise de vue, et un JavaBean en tant que modèle. Les composants JSF sont utilisés pour lier la vue avec le modèle, et la FacesServlet utilise l'arbre des composants JSF pour effectuer tout le travail. Si nous ne devons retenir qu'une chose, c'est celle-ci : **avec JSF, nous n'avons plus besoin d'écrire de servlets !**

* JSF 2.0 est un framework web MVC qui se focalise sur la simplification de la construction d'interfaces utilisateur, autrement dit la vue. Il propose nativement plus d'une centaine de balises à cet égard, et facilite la réutilisation des composants d'une interface utilisateur à l'autre.
* Avec JSF 2, les annotations remplacent les fichiers de configuration externes.

**Structure d'une application JSF**

Étudions brièvement comment se construit une application basée sur JSF :

* La vue est généralement assurée par des pages JSP ou par des pages XHTML (on parle alors de Facelets).
* Le modèle est assuré par des entités ou des JavaBeans.
* Le contrôleur, auparavant incarné par les servlets, est décomposé en deux éléments :
  + une unique servlet mère servant de point d'entrée à toute requête, la FacesServlet
  + un JavaBean particulier, déclaré via une annotation et désigné par le terme managed-bean.
* Le tout est mis en musique par des fichiers de configuration : le classique web.xml, mais également un nouveau fichier nommé faces-config.xml.

La différence majeure qui ressort jusqu'à présent est l'absence de servlets spécifiques à chaque page, comme nous devions en écrire dans nos applications MVC faites maison.

**Facelets et composants**

Nous allons aborder la structure d'une Facelet en la comparant avec la structure d'une technologie que nous connaissons déjà : la page JSP. La différence la plus importante se situe au niveau du contenu. Alors qu'une page JSP est un objet complexe, pouvant contenir à la fois des balises JSP ou JSTL et du code Java par l'intermédiaire de scriptlets, **une Facelet est un fichier XML pur** ! Elle ne peut par conséquent contenir que des balises, et il est impossible d'y inclure des scriplets : le Java est donc définitivement banni de la vue avec cette technologie. C'est le langage XHTML qui est le plus utilisé pour créer des Facelets (XHTML étant une version de la syntaxe HTML conforme au standard XML). Toutefois, le framework JSF est markup-agnostique : cela signifie qu'il peut s'adapter à n'importe quel langage, à partir du moment où ce langage respecte la structure décrite par le standard XML. Ainsi, il est également possible de construire une Facelet avec du langage XML pur, par exemple pour créer un flux RSS, ou avec le langage XUL pour n'en citer qu'un.

Une Facelet porte en général une de ces trois extensions : **.jsf**, **.xhtml** ou **.faces**. Dans l'absolu, c'est par défaut l'extension .xml qui pourrait être utilisée...

Voici la structure à vide d'une Facelet :

<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">

...

</html>

**Bibliothèques et balises**

Il existe un moyen pour inclure les bibliothèques de balises JSF dans une Facelet. Toutefois, il faut oublier le concept même de la directive JSP : il s'agissait là littéralement d'un ordre donné au conteneur, lui précisant comment il devait gérer une page JSP et générer sa servlet Java associée. Voilà pourquoi il était non seulement possible via une directive d'inclure des bibliothèques, mais également d'importer des classes Java, d'activer ou non les sessions HTTP, les expressions EL, etc.

Dans une Facelet, une bibliothèque de balises est incluse via l'ajout d'un attribut xmlns à la balise <html> qui ouvre le corps de la page. Il s'agit là d'un namespace XML. Le framework JSF intègre nativement trois bibliothèques standard : **HTML**, **Core** et **UI**. Voici comment les inclure dans une Facelet :

<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml"

xmlns:f="http://java.sun.com/jsf/core"

xmlns:h="http://java.sun.com/jsf/html"

xmlns:ui="http://java.sun.com/jsf/facelets">

...

</html>

Traditionnellement, h: désigne la bibliothèque HTML, ui: la bibliothèque de templating et de composition, et f: la bibliothèque Core. Au passage, il ne faut pas confondre cette dernière avec la bibliothèque Core de la JSTL, nous parlons bien ici de balises JSF ! Voici une balise issue de la bibliothèque HTML :

<h:outputLabel for="confirmation">Confirmation du mot de passe <span class="requis">\*</span></h:outputLabel>

**Expressions EL**

Voici la forme d'une expression EL avec JSF :

#{inscrireBean.utilisateur.motDePasse}

Il est possible d'utiliser la syntaxe #{...} aussi depuis des pages JSP !

Dans une Facelet, les balises JSF ne sont que des appels à des composants JSF qui sont entièrement autonomes. Lorsqu'un composant JSF est appelé, il génère son propre rendu dans son état courant. Le cycle de vie des composants JSF n'a par conséquent **aucune relation avec le cycle de vie d'une page JSP** et de sa servlet auto-générée. Une Facelet est une page constituée d'une suite d'appels à des composants JSF (réalisés par l'intermédiaire de balises), et ceux-ci forment ce que l'on appelle **un arbre de composants**, ou component tree en anglais.

À la différence des JSP, les Facelets JSF conservent un état (on dit alors que la vue est stateful) : ce sont les composants autonomes appelés par l'intermédiaire des balises contenues dans les Facelets qui permettent de maintenir cet état. De la même manière que Swing et AWT, les composants JSF suivent le *pattern* de l'objet composite pour gérer un arbre de composants : en clair, cela signifie à la fois qu'un objet conteneur contient un composant, et qu'un objet conteneur est lui-même un composant. La vue lie ces composants graphiques à la page XHTML, et permet ainsi au développeur de directement lier des champs HTML d'interaction utilisateur (saisie de données, listes, etc.) à des propriétés de beans, et des boutons à leurs méthodes d'action.

**Un processus en 6 étapes**

Avec JSF, le traitement d'une requête entrant sur le serveur est découpé en six étapes :

**1. La restauration de la vue**

La requête entrante est redirigée vers l'unique servlet jouant le rôle de super-contrôleur, la FacesServlet. Celle-ci examine son contenu, en extrait le nom de la page ciblée et détermine s'il existe déjà une vue associée à cette page (avec JSF la vue conserve un état). Voilà pourquoi cette étape s'intitule "restauration de la vue" : il s'agit en réalité de restaurer les éventuels composants déjà chargés si l'utilisateur a déjà accédé à la page par le passé.

La FacesServlet va donc chercher les composants utilisés par la vue courante. Si la vue n'existe pas déjà, elle va la créer. Si elle existe déjà, elle la réutilise. La vue contient tous les composants de l'interface utilisateur intervenant dans la page. La vue (c'est-à-dire l'ensemble des composants qui y interviennent, et donc son état) est sauvegardée dans l'objet FacesContext.

**2. L'application des valeurs contenues dans la requête**  
Arrivés à cette étape, les composants de la vue courante ont tout juste été récupérés ou créés depuis l'objet **FacesContext**. Chacun d'eux va maintenant récupérer la valeur qui lui est assignée depuis les paramètres de la requête, ou éventuellement depuis des cookies ou headers.

Ces valeurs vont alors être converties. Si un champ est lié à une propriété de type Integer, alors son contenu va être converti en Integer. Si cette conversion échoue, un message d'erreur va être placé dans le **FacesContext**, et sera utilisé lors du futur rendu de la réponse.

À noter qu'à cette étape peut intervenir la "prise en charge immédiate des événements" : cela veut dire que si un composant est marqué comme tel, sa valeur va directement être convertie puis validée dans la foulée. Si aucun composant n'arbore cette propriété, alors les valeurs de tous les composants sont d'abord converties, puis intervient ensuite l'étape de validation sur l'ensemble des valeurs.

**3. La validation des données**  
Les valeurs tout juste converties vont ensuite être validées, en suivant les règles de validation définies par le développeur. Si la validation d'une valeur échoue, un message d'erreur est ajouté au FacesContext, et le composant concerné est marqué comme "invalide" par JSF. La prochaine étape est alors directement le rendu de la réponse, il n'y aura aucune autre étape intermédiaire.

Si les valeurs sont correctes vis-à-vis des règles de validation en place, alors la prochaine étape est la mise à jour des valeurs du modèle.

**4. La mise à jour des valeurs du modèle**  
Les composants peuvent être directement liés, par l'intermédiaire des balises présentes dans la vue, à des propriétés de beans. Ces beans sont qualifiés de managed-beans ou backing-beans, car ils sont gérés par JSF et la vue s'appuie sur eux. Si de tels liens existent, alors les propriétés de ces beans sont mises à jour avec les nouvelles valeurs des composants correspondants, fraîchement validées. Puisque la validation a eu lieu en premier lieu, le développeur est certain que les données enregistrées dans le modèle sont valides, au sens format du champ du formulaire. Il n'est par contre pas exclu que les données ne soient pas valides d'un point de vue de ce qu'attend le code métier de l'application.

**5. L'appel aux actions, le code métier de l'application**  
Les actions associées à la soumission du formulaire sont alors appelées par JSF. Il s'agit de l'entrée en jeu du code métier : maintenant que les données ont été converties, validées et enregistrées dans le modèle, elles peuvent être utilisées par l'application.

La fin de cette étape se concrétise par la redirection vers la vue correspondante, qui peut dépendre ou non du résultat produit par le code métier. Il s'agit donc de définir la navigation au sein des vues existantes, ce qui est réalisé directement depuis le bouton de validation dans la page, ou depuis un fichier de configuration XML externe nommé **faces-config.xml**.

**6. Le rendu de la réponse**  
La dernière étape est le rendu de la réponse. La vue définie dans la navigation est finalement affichée à l'utilisateur : tous les composants qui la composent effectuent alors leur propre rendu, dans leur état courant. La page HTML ainsi générée est finalement envoyée au client.

**De quoi avons-nous besoin ?**

JSF n'est qu'une spécification, et pour utiliser JSF il faut donc disposer d'une implémentation. Avec JPA, c'était exactement pareil : l'implémentation de référence de JPA utilisée par défaut par GlassFish était EclipseLink, mais il existait d'autres implémentations très utilisées comme Hibernate notamment. En ce qui concerne JSF, il existe deux principales implémentations :

* Oracle Mojarra, l'implémentation de référence, utilisée par défaut par GlassFish.
* Apache MyFaces, l'implémentation éditée par Apache.

**Création d’un bean**

@ManagedBean

@RequestScoped

public class BonjourBean implements Serializable {

private static final long serialVersionUID = 1L;

…

}

* Le bean implémente l'interface **Serializable**. En rendant un bean sérialisable, nous lui donnons la capacité de survivre à un redémarrage du serveur. Cela ne va pas plus loin que cela.
* Le bean contient deux annotations spécifiques à JSF :
  + @ManagedBean : permet de préciser au serveur que ce bean est dorénavant géré par JSF. Cela signifie que JSF va utiliser ce bean en tant que modèle associé à une ou plusieurs vues. Par défaut, le nom du bean correspond au nom de la classe, la majuscule en moins : en l'occurrence le nom de notre bean est donc **bonjourBean**. Si nous voulions désigner ce bean par un autre nom, par exemple **direBonjour**, alors il nous faudrait annoter le bean en précisant le nom souhaité : @ManagedBean(name="direBonjour").
  + @RequestScoped : permet de préciser au serveur que ce bean a pour portée la requête. Il s'agit en l’occurrence de la portée utilisée par défaut en cas d'absence d'annotation. Il existe autant d'annotations que de portées disponibles dans JSF : @NoneScoped, @RequestScoped, @ViewScoped, @SessionScoped, @ApplicationScoped, et @CustomScope.

Ainsi, un managed-bean ou backing-bean est un simple bean annoté pour le déclarer comme tel auprès de JSF.

**Création des facelets**

Exemples de balises JSF :

* La section header est créée par la balise <h:head>. Il s'agit d'un composant JSF qui permet d'inclure des ressources JS ou CSS dans le contenu généré entre les balises HTML <head> de manière automatisée depuis le code Java.
* Création d’un formulaire : <h:form> </h:form>. La méthode HTTP utilisée pour l'envoi des données est toujours **POST**. Pour utiliser la méthode GET, le plus simple est de ne pas utiliser le composant JSF et d'écrire directement le formulaire en HTML brut en y spécifiant l'attribut <form... method="get">. Les données sont par défaut renvoyées à la page contenant le formulaire.
* Un champ de texte JSF qui affiche la propriété **nom** du bean **bonjourBean** (notre managed-bean, géré par JSF) : <h:inputText value="#{bonjourBean.nom}" />. Son attribut **value** permet de définir deux choses :
  + La valeur contenue dans cet attribut sera par défaut affichée dans le champ texte, il s'agit ici du même principe que pour un champ de texte HTML classique <input type="text" value="..." />.
  + La valeur contenue dans cet attribut sera utilisée pour initialiser la propriété **nom** du bean **BonjourBean**, par l'intermédiaire de l'expression EL #{bonjourBean.nom}. En l'occurrence, JSF va évaluer l'expression lorsque le formulaire sera validé, c'est-à-dire lorsque le bouton sera cliqué, et va alors chercher l'objet nommé **bonjourBean**, puis utiliser sa méthode settersetNom() pour enregistrer dans la propriété **nom** la valeur contenue dans le champ texte.
* Un bouton de formulaire, chargé d'afficher la page bienvenue.xhtml lorsqu'il est cliqué : <h:commandButton value="Souhaiter la bienvenue" action="bienvenue" /> . Le composant se chargera automatiquement derrière les rideaux d'appeler la page nommée bienvenue.xhtml. Pour mettre en place une navigation un peu plus complexe, il faudra utiliser une section navigation-rule dans le **faces-config.xml**.

**Configuration de l'application**

Il faut écrire un peu de configuration dans le fichier **web.xml** :

<!-- Déclaration du contrôleur central de JSF : la FacesServlet -->

<servlet>

<servlet-name>Faces Servlet</servlet-name>

<servlet-class>javax.faces.webapp.FacesServlet</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

<!-- Mapping : association des requêtes dont le fichier porte l'extension .xhtml à la FacesServlet -->

<servlet-mapping>

<servlet-name>Faces Servlet</servlet-name>

<url-pattern>\*.xhtml</url-pattern>

</servlet-mapping>

</web-app>

Il faut déclarer la servlet mère de JSF, celle qui joue le rôle du Front Controller : la **FacesServlet**. C'est exactement le même principe que lorsque nous définissions nos propres servlets à la main. Il suffit de préciser un nom, en l'occurrence "**Faces Servlet**", et sa localisation dans l'application, javax.faces.webapp.FacesServlet.

Il faut ensuite procéder au mapping d'un pattern d'URL sur cette seule et unique **FacesServlet**. L'objectif est de rediriger **toutes les requêtes entrantes** vers elle. Dans notre exemple, nous nous contentons d'associer les vues portant l'extension **.xhtml** à la FacesServlet. Tous les développeurs n'utilisent pas l'extension **.xhtml** pour leurs vues, et il est courant dans les projets JSF existants de rencontrer quatre types d'URL différentes : /faces/\*, \*.jsf, \*.xhtml et \*.faces.

On peut ajouter au fichier **web.xml** les lignes suivantes :

<!-- Changer cette valeur à "Production" lors du déploiement final de l'application -->

<context-param>

<param-name>javax.faces.PROJECT\_STAGE</param-name>

<param-value>Development</param-value>

</context-param>

Il s'agit d'une fonctionnalité utile proposée par JSF. Lors du développement d'une application, il est recommandé d'initialiser le paramètre nommé javax.faces.PROJECT\_STAGE avec la valeur “**Development**“. Ceci va rendre disponibles de nombreuses informations de debugging et les afficher directement au sein de nos pages en cas de problème, permettant ainsi de tracer les erreurs rapidement. Lors du déploiement final, une fois l'application achevée, il suffit alors de changer la valeur du paramètre à “**Production**“, et toutes ces informations non destinées au public ne seront alors plus affichées.

Voici ce qui a changé par rapport à notre ancienne méthode, sans JSF :

* Auparavant, nous aurions dû écrire deux pages JSP, en lieu de place de nos deux simples Facelets.
* Auparavant, nous aurions dû écrire une servlet, chargée :
  + d 'afficher la page bonjour.jsp lors de la réception d'une requête GET.
  + de récupérer le contenu du champ texte du formulaire lors de la réception d'une requête POST, puis de s'en servir pour initialiser la propriété du bean BonjourBean, que nous aurions d'ailleurs dû créer, avant de transmettre le tout sous forme d'attributs à la page bienvenue.jsp pour affichage.

Ainsi, voici ce dont nous n'avons plus à nous soucier avec JSF :

* **La manipulation des objets requête et réponse** : avec JSF, nous n'avons pas conscience de ces objets, leur existence nous est masquée par le *framework*.
* **L'extraction des paramètres contenus dans une requête HTTP** : avec JSF, il nous suffit d'écrire une EL au sein d'une Facelet, et le tour est joué.
* **L'initialisation manuelle des beans** : avec JSF, le cycle de vie des beans annotés est entièrement géré par le *framework* ! À aucun moment nous n'avons initialisé un bean, et à vrai dire, à aucun moment nous n'avons créé une classe Java susceptible de pouvoir procéder à cette initialisation ! Tout ce que nous avons mis en place, ce sont deux Facelets et un bean.
* **La mise en place d'attributs dans un objet HttpServletRequest pour transmission à une page JSP** : avec JSF, il suffit d'écrire une EL dans un composant de notre Facelet, et le *framework* s'occupe du reste.
* **La redirection manuelle vers une page JSP pour affichage** : avec JSF, il suffit de préciser la page ciblée dans l'attribut **action** du composant <h:commandButton>.

**Les ressources**

Il est possible de déclarer une feuille CSS dans la section header de notre Facelet :

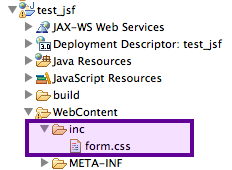
<h:head>

<title>Premier exemple JSF 2.0</title>

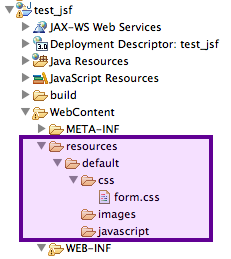
<link type="text/css" rel="stylesheet" href="inc/form.css" />

</h:head>

Nous devrions alors bien prendre soin de placer la feuille **form.css** et le dossier **/inc** à la racine du projet :



Cependant, avec JSF 2.x, une bonne pratique est de placer toutes les ressources web telles que les fichiers CSS, les images ou les fichiers JavaScript, dans un répertoire intitulé **resources** et placé directement à la racine de notre application, c'est-à-dire au même niveau que le dossier **/WEB-INF**.



Une fois ceci en place, nous pouvons maintenant utiliser un composant dédié à l'inclusion de ressources depuis nos Facelets. Par exemple, pour utiliser la feuille CSS **form.css** placée dans le dossier **/resources/default/css**, il suffit de remplacer l'inclusion traditionnelle via <link type="text/css" ... > par :

<h:outputStylesheet library="default" name="css/form.css" />

Ce composant JSF permet de cibler directement le sous-dossier du répertoire **resources** via le contenu de son attribut **library**, que nous avons donc renseigné par "default" afin d'accéder au sous-dossier de ce nom. Il cible enfin le fichier souhaité via le contenu de son attribut **name**.

**Remarque :** Le composant JSF ici utilisé est dédié à l'inclusion de feuilles CSS. Il en existe d'autres similaires pour l'inclusion des fichiers JavaScript et des images.

JSF n’est pas uniquement une alternative à HTML/CSS/JS, il prend également en compte la partie serveur (typiquement JSP/Servlets). JSF permet d'éviter tout le code passe-partout en charge du regroupement des paramètres de requêtes HTTP, de leur conversion/validation, de la mise à jour des données du modèle et de l'exécution de la bonne méthode Java pour réaliser les traitements métiers. Avec JSF, nous n'avons plus qu'une page XHTML en guise de vue, et un JavaBean en tant que modèle. Cela accélère le développement de manière significative !

## La gestion d'un formulaire avec JSF

Par défaut, le contenu d'un champ laissé vide dans nos formulaires sera considéré comme une chaîne vide. Il est possible de forcer le conteneur à considérer un tel contenu comme une valeur nulle plutôt que comme une chaîne vide, en ajoutant cette section dans le fichier **web.xml** :

<context-param>

<param-name> javax.faces.INTERPRET\_EMPTY\_STRING\_SUBMITTED\_VALUES\_AS\_NULL

</param-name>

<param-value>true</param-value>

</context-param>

**Création du backing bean**

* FacesMessage : cet objet permet simplement de un message de validation. Il existe plusieurs constructeurs, notamment un qui permet d'associer à un message un niveau de criticité, en précisant une catégorie qui est définie par FacesMessage.Severity. Les niveaux existants sont représentés par des constantes que nous pouvons retrouver sur la documentation de l'objet FacesMessage. En ce qui nous concerne, nous ne spécifions qu'un message dans le constructeur, et c'est par conséquent la criticité Severity.INFO qui est appliquée par défaut à notre message par JSF : FacesMessage message = new FacesMessage( "Succès de l'inscription !" );
* FacesContext : nous retrouvons là l'objet qui contient l'arbre des composants d'une vue ainsi que les éventuels messages d'erreur qui leur sont associés. Nous pouvons nous en servir pour mettre en place un **FacesMessage** dans le contexte courant via la méthode addMessage(), pour que la réponse puisse ensuite l'afficher : FacesContext.getCurrentInstance().addMessage( null, message );

Les méthodes de récupération et conversion des valeurs envoyées depuis le formulaire ne sont pas contenue dans le *managed-bean*. Les étapes de récupération, conversion, validation et enregistrement dans le modèle sont entièrement automatisées (cf cycle de vie JSF). C'est depuis la vue que nous allons directement effectuer les associations entre les champs du formulaire et les propriétés de notre entité.

**Création de la vue**

C’est dans nos *facelets* qu’est effectuée l’association entre les champs du formulaire et les propriétés de notre entité. Par exemple :

<h:inputSecret id="confirmation" value="#{inscrireBean.utilisateur.motDePasse}" required="true" size="20" maxlength="20" />

**Les labels**

Pour générer un label associé à un champ de formulaire, concrétisé par la balise HTML <label>, il faut utiliser le composant <h:outputLabel>. Le comportement de l'attribut **for** est identique à celui de la balise HTML, il suffit d'y préciser l'**id** du champ de saisie auquel le label fait référence.

**Les champs de saisie**

Pour générer un champ de saisie de type texte, concrétisé par la balise HTML <input type="text" ...>, il faut utiliser [le composant <h:inputText>](http://docs.oracle.com/javaee/6/javaserverfaces/2.1/docs/vdldocs/facelets/h/inputText.html). Tout comme la balise HTML, il accepte les attributs **id**, **value**, **size** et **maxlength**. Pour générer un champ de saisie de type mot de passe, concrétisé par la balise HTML <input type="password" ...>, il faut utiliser le composant <h:inputSecret>.

L’attribut **required**, qui peut prendre comme valeur true ou false, va déterminer si l'utilisateur doit obligatoirement saisir des données dans ce champ ou non. Il permettra de générer une erreur lors de la validation de la valeur du champ associé.

**Les messages**

Le composant <h:message> permet de générer un message associé à un champ de saisie en particulier. Entre autres, celui-ci accepte deux attributs utiles :

* **for**, pour cibler l'id du champ concerné.
* **errorClass**, pour permettre de donner une classe CSS particulière lors de l'affichage du message généré s'il s'agit d'une erreur.

Ce composant va donc afficher, lors du rendu de la réponse, l'éventuel message associé au champ ciblé par l'attribut **for**. S'il s'agit d'un message d'erreur, que JSF sait différencier des messages d'information qui peuvent éventuellement être placés dans le FacesContext grâce au niveau de criticité associé à un message, alors le style **erreur** défini dans notre feuille CSS sera appliqué.

Le composant <h:messages> provoque par défaut l'affichage de tous les messages disponibles dans la vue, y compris ceux qui sont déjà affichés via un <h:message> ailleurs dans la page. Toutefois, il est possible de n'afficher que les messages qui ne sont attachés à aucun composant défini, c'est-à-dire les messages dont l'**id** est null, en utilisant l'attribut optionnel globalOnly="true". L'attribut **infoClass** permet de donner à notre message global le style **info** qui est défini dans notre feuille CSS :

<h:messages globalOnly="true" />

**Le bouton d'envoi**

Pour générer un bouton de soumission de formulaire, concrétisé par la balise HTML <input type="submit" ...>, il faut utiliser le composant <h:commandButton>. Le contenu de son attribut **value** est affiché en tant que texte du bouton HTML généré, et son attribut **action** permet de définir la navigation :

<h:commandButton value="Inscription" action="#{inscrireBean.inscrire}" styleClass="sansLabel" />

Dans règle de navigation particulière précisée par le développeur, c'est la page courante qui est automatiquement utilisée comme action d'un formulaire JSF.

**Amélioration des messages affichés lors de la validation**

Les composants <h:inputText> et <h:inputSecret> possèdent un attribut intitulé **label**, qui est utilisé pour représenter un champ de manière littérale dans le message d’erreur :

<h:inputText id="email" value="#{inscrireBean.utilisateur.email}" ... label="Email" />

L'attribut **requiredMessage** des composants de saisie JSF permet de définir le message utilisé lors de la vérification de la règle définie par l'attribut **required** :

<h:inputText id="email" value="#{inscrireBean.utilisateur.email}" ... requiredMessage="Veuillez saisir une adresse email" />

Il existe justement une fonctionnalité dans JSF qui permet de définir ces messages dans un fichier externe, et d'y faire référence depuis les Facelets. Ce système s'appelle un **bundle**.

**Mise en place d'un bundle**

Un bundle n'est rien d'autre qu'un fichier de type Properties contenant une liste de messages. Il faut le placer dans les sources du projet, aux côtés du code de l'application. Il contiendra par exemple :

inscription.email = Veuillez saisir une adresse email

Pour charger le bundle depuis notre Facelet pour qu'elle puisse faire référence à son contenu, il faut utiliser le composant <f:loadBundle>. Celui-ci doit être placé dans la page **avant** les balises qui en feront usage, typiquement nous pouvons le mettre dans le header de notre page :

<f:loadBundle basename="com.sdzee.bundle.messages" var="msg"/>

Cette balise attend uniquement deux attributs :

* **basename**, qui contient le chemin complet dans lequel est placé le fichier (le package suivi du nom du fichier).
* **var**, qui permet de définir par quel nom le bundle sera désigné dans le reste de la page.

Pour utiliser ce bundle nous écrirons alors :

<h:inputText id="email" value="#{inscrireBean.utilisateur.email}" ... requiredMessage="#{msg['inscription.email']}" />

**Une inscription ajaxisée**

AJAX est l'acronyme d'**Asynchronous Javascript and XML**, ce qui en français signifie littéralement « Javascript et XML asynchrones ». Derrière cette appellation se cache un ensemble de technologies qui permettent la mise à jour d'un fragment d'une page web sans que le rechargement complet de la page web visitée par l'utilisateur ne soit nécessaire.

**L'AJAX avec JSF**

Pour utiliser l’ajax avec jsf, il existe une balise de la bibliothèque Core: <f:ajax>. Par exemple :

<h:outputLabel for="confirmation">

Confirmation du mot de passe <span class="requis">\*</span>

</h:outputLabel>

<h:inputSecret id="confirmation" value="#{inscrireBean.utilisateur.motDePasse}" required="true" size="20" maxlength="20" requiredMessage="#{msg['inscription.confirmation']}">

<f:ajax event="blur" render="confirmationMessage" />

</h:inputSecret>

<h:message id="confirmationMessage" for="confirmation" errorClass="erreur" />

Seuls deux de ses attributs nous sont utiles dans les champs de saisie :

* **event**, qui nous permet de définir l'action déclenchant l'envoi de la requête AJAX au serveur.
* **render**, qui nous permet de définir le ou les composants dont le rendu doit être effectué, une fois la requête traitée par le serveur.

Dans notre cas, ce qui nous intéresse c'est de faire valider le contenu d'un champ dès que l'utilisateur a terminé la saisie, et d'actualiser le message associé au champ en conséquence :

* l'**event** que nous utilisons pour considérer que l'utilisateur a terminé la saisie du contenu d'un champ s'intitule **blur**. Cette propriété JavaScript va permettre de déclencher une requête dès que le champ courant perd le focus.
* pour désigner quel composant actualiser, nous devons simplement préciser son id dans l'attribut **render**.

Une balise <f:ajax> placée sur un bouton de validation est un peu différente des autres : elle permet d'effectuer un envoi complet, mais là encore ajaxisé et non pas simplement un envoi classique avec rechargement de la page.

<h:commandButton value="Inscription" action="#{inscrireBean.inscrire}" styleClass="sansLabel">

<f:ajax execute="@form" render="@form" />

</h:commandButton>

Pour ce faire, il existe quatre marqueurs donnés :

* @this : désigne le composant englobant la balise <f:ajax>.
* @form : désigne le formulaire entier.
* @all : désigne l'arbre des composants de la page entière.
* @none : ne désigne aucun composant.

Nous n'utilisons plus l'attribut **event** comme nous le faisions sur les champs de saisie, car nous savons très bien que c'est un clic sur le bouton qui va déclencher l'action. Nous utilisons cette fois l'attribut **execute**, qui permet de définir la portée de l'action à effectuer : en l'occurrence, nous souhaitons bien traiter le formulaire complet.

**L'importance de la portée d'un objet**

Dans le cas de l’utilisation de requête Ajax, un bean déclaré @RequestScoped ne vit que le temps d'une requête : il est créé par JSF à l'arrivée d'une requête sur le serveur, puis détruit une fois la réponse renvoyée. Autrement dit, notre formulaire provoque la création et la destruction d'un bean sur le serveur à chaque fois que l'utilisateur change de champ ! C'est du gâchis de performances.

Pour compenser cela, il existe une portée qui se place entre la requête et la session, et qui semble donc parfaite pour ce que nous faisons ici : le scope de conversation. Il permet de conserver un objet tant qu'une même vue est utilisée par un même utilisateur. Cela signifie que tant qu'un utilisateur effectue des requêtes depuis une seule et même page, alors l'objet est conservé sur le serveur et réutilisé. Dès que l'utilisateur effectue une requête vers une autre vue, alors l'objet est finalement détruit. Pour déclarer un **backing-bean** dans cette portée, il faut l'annoter avec @ViewScoped.

**Une inscription contrôlée**

Nous nous sommes jusqu'à présent contentés d'effectuer une simple vérification sur les champs du formulaire, directement dans la vue grâce aux attributs **required** et **requiredMessage**. Nous allons déplacer ces conditions de la vue vers notre modèle, à savoir notre entité, grâce à de simples annotations ! Pour cela, il existe des annotations comme @NotNull qui permettent d'indiquer qu'une propriété ne peut être laissée vide. Il faut ajouter ces annotations dans nos EJBs Entité.

**Remarque :** Ceci ne fonctionnera que si nous avons ajouté javax.faces.INTERPRET\_EMPTY\_STRING\_SUBMITTED\_VALUES\_AS\_NULL en paramètre de contexte. Sans cette configuration, les champs du formulaire laissés vides seraient traités comme des chaînes vides par le serveur.

Ces annotations permettent également de personnaliser les messages d’erreur directement depuis notre entité :

@NotNull( message = "Veuillez saisir une adresse email" )

Quelle est la méthode de validation à préférer : depuis la vue, ou depuis l'entité ? Cela dépend principalement des contraintes du projet. Par exemple, si l'application doit pouvoir fonctionner sur un serveur léger comme Tomcat sans support des EJB ni de JPA, alors il ne faudra pas utiliser la validation depuis l'entité mais lui préférer la validation depuis la vue JSF.

**Affiner les contrôles effectués**

L'annotation @Pattern permet de vérifier un format grâce à l'utilisation d'expressions régulières :

@Pattern( regexp = "([^.@]+)(\\.[^.@]+)\*@([^.@]+\\.)+([^.@]+)", message = "Merci de saisir une adresse mail valide" )

private String email;

L'annotation @Size permet de vérifier la taille des champs :

@Size( min = 3, message = "Le mot de passe doit contenir au moins 3 caractères" )

private String motDePasse;

Pour vérifier qu’un champ entré par l'utilisateur contient au moins 8 caractères, dont au moins un chiffre, une lettre minuscule et une lettre majuscule :

@Pattern(regexp = ".\*(?=.{8,})(?=.\*\\d)(?=.\*[a-z])(?=.\*[A-Z]).\*", message = "Le mot de passe saisi n'est pas assez sécurisé")

**Ajouter des contrôles "métier"**

On les utilisera pour effectuer des contrôles sur par exemple :

* Un mot de passe et sa confirmation qui doivent être identiques.
* Un adresse email qui ne doit pas exister dans la base de données.

Ces contraintes se distinguent des simples vérifications de format que nous avons mises en place jusqu'à présent, car elles ont trait à l'aspect métier de notre application. Pour cela, JSF fournit une interface nommée javax.faces.validator.Validator qui permet de créer une classe contenant une méthode de validation. Celle-ci pourra alors être liée à un composant très simplement depuis la vue, via un attribut placé dans une balise.

Nous allons pour l’exemple mettre en place la vérification de l'existence de l'adresse email dans la base de données. La seule méthode qu'il est nécessaire de surcharger est la méthode **validate()** : c'est elle qui va contenir le code métier chargé d'effectuer le contrôle de l'existence de l'adresse dans la base :

public class ExistenceEmailValidator implements Validator {

private static final String EMAIL\_EXISTE\_DEJA = "Cette adresse email est déjà utilisée";

@EJB

private UtilisateurDao utilisateurDao;

@Override

public void validate( FacesContext context, UIComponent component, Object value ) throws ValidatorException {

// value correspond à la valeur du composant dans la vue qui appel ce validateur.

/\* Récupération de la valeur à traiter depuis le paramètre value \*/

String email = (String) value;

try {

if ( utilisateurDao.trouver( email ) != null ) {

/\*

\* Si une adresse est retournée, alors on envoie une exception propre à JSF, qu'on initialise avec un FacesMessage de gravité "Erreur" et contenant le message d'explication. Le framework va alors gérer lui-même cette exception et s'en servir pour afficher le message d'erreur à l'utilisateur.

\*/

throw new ValidatorException(

new FacesMessage( FacesMessage.SEVERITY\_ERROR, EMAIL\_EXISTE\_DEJA, null ) );

}

} catch ( DAOException e ) {

/\*

En cas d'erreur imprévue émanant de la BDD, on prépare un message d'erreur contenant l'exception retournée, pour l'afficher à l'utilisateur ensuite.

\*/

FacesMessage message = new FacesMessage( FacesMessage.SEVERITY\_ERROR, e.getMessage(), null );

FacesContext facesContext = FacesContext.getCurrentInstance();

facesContext.addMessage( component.getClientId( facesContext ), message );

}

}

}

Il faut ensuite déclarer l’objet ainsi codé auprès de JSF afin qu'il le rende accessible à notre vue. Pour cela, JSF propose l'annotation @FacesValidator qui permet de déclarer auprès du framework un objet comme étant un **Validator**, ce qui le rend accessible depuis une balise dans nos Facelets.

Malheureusement, nous n'allons pas pouvoir utiliser cette annotation. Il s'agit là d'un comportement étrange de JSF : les objets annotés avec @FacesValidator ne sont pas pris en charge par le conteneur d'injection. Formulé autrement, cela signifie qu'il n'est pas possible d'injecter un EJB avec l'annotation @EJB dans un **Validator** JSF annoté avec @FacesValidator. C'est un problème, car nous faisons usage de notre DAO Utilisateur dans notre validateur, et puisque nous travaillons avec JPA nous avons besoin de pouvoir y injecter cet EJB (ceci est vrai pour la version 2.2 de JSF, ce bug est éventuellement corrigé pour les versions supérieures).

Pour contourner cela, nous pouvons déclarer notre objet comme un simple **backing-bean.** Maintenant que notre objet est déclaré et donc accessible depuis la vue, il faut ajouter une balise à notre Facelet pour qu'elle fasse appel à ce validateur pour le champ de saisie de l'adresse email. Pour ce faire, il existe la balise <f:validator>. Elle possède un attribut **validatorId** permettant de préciser quel objet utiliser en tant que validateur. Oui mais voilà, cet attribut ne fonctionne que si notre objet est annoté avec @FacesValidator. Cependant, la balise propose un autre attribut intitulé **binding**, qui permet de préciser un **backing-bean** à utiliser en tant que validateur, à la condition que cet objet implémente l'interface **Validator** :

<h:inputText id="email" value="#{inscrireBean.utilisateur.email}" size="20" maxlength="60">

<f:ajax event="blur" render="emailMessage" />

<f:validator binding="#{existenceEmailValidator}" />

</h:inputText>

L'annotation @FacesValidator attend en paramètre le nom qui sera utilisé comme identifiant depuis la vue pour désigner ce validateur :

@FacesValidator( value = "confirmationMotDePasseValidator" )

L’appel dans la vue se fera donc via :

<f:validator validatorId="confirmationMotDePasseValidator" />

Pour vérifier l’égalité de deux champs différents du formulaire, il faut ensuite déclarer le composant en charge du champ de mot de passe **en tant qu'attribut du composant** en charge de la confirmation. L'intérêt de réaliser cette association est de rendre disponible le contenu du champ de mot de passe depuis le composant en charge du champ de confirmation. Cela se fait via [la balise <f:attribute>](http://docs.oracle.com/javaee/6/javaserverfaces/2.1/docs/vdldocs/facelets/f/attribute.html) :

<f:attribute name="composantMotDePasse" value="#{composantMotDePasse}" />

Sa propriété **name** permet de définir le nom de l'objet qui va être créé en tant qu'attribut au composant courant, et sa propriété **value** permet de définir son contenu. Nous avons donc nommé l'objet **composantMotDePasse**, et avons lié la propriété **value** directement avec sa valeur en utilisant l'expression EL #{composantMotDePasse}.

Il faut ensuite faire en sorte que la valeur du champ mot de passe soit bien affectée à la valeur de cet objet. Pour ce faire, il faut modifier la déclaration du composant en charge du mot de passe de cette manière :

<h:inputSecret id="motdepasse" value="#{inscrireBean.utilisateur.motDePasse}" binding="#{composantMotDePasse}" size="20" maxlength="20">

</h:inputSecret>

En ajoutant un attribut **binding** et en le liant à la valeur de l'objet via l'expression EL #{composantMotDePasse}, nous nous assurons alors que la valeur saisie dans le champ de mot de passe sera affectée à l'objet déclaré en tant qu'attribut du composant de confirmation !

Maintenant que nous vérifions l'égalité entre les deux champs, pour rendre l'expérience utilisateur plus intuitive et logique, il faut faire en sorte que la validation AJAX de l'un entraîne la validation de l'autre, et vice-versa :

<h:inputSecret id="motdepasse" value="#{inscrireBean.utilisateur.motDePasse}" binding="#{composantMotDePasse}" size="20" maxlength="20">

<f:ajax event="blur" execute="motdepasse confirmation" render="motDePasseMessage confirmationMessage" />

</h:inputSecret>

<h:inputSecret id="confirmation" value="#{inscrireBean.utilisateur.motDePasse}" size="20" maxlength="20">

<f:validator validatorId="confirmationMotDePasseValidator" />

<f:attribute name="composantMotDePasse" value="#{composantMotDePasse}" />

<f:ajax event="blur" execute="motdepasse confirmation" render="motDePasseMessage confirmationMessage" />

</h:inputSecret>

Pour finir, il faut trouver un moyen pour récupérer le contenu du champ mot de passe depuis le composant de validation « confirmationMotDePasseValidator » lié au champ confirmation. L['objet UIComponent](http://docs.oracle.com/javaee/6/api/javax/faces/component/UIComponent.html) utilisé dans les validators contient une méthode [getAttributes()](http://docs.oracle.com/javaee/6/api/javax/faces/component/UIComponent.html#getAttributes%28%29) qui permet de récupérer une liste des attributs du composant. En effet, nous avons déclaré le composant mot de passe comme attribut du composant confirmation dans notre Facelet, par le biais de la balise <f:attribute>. Les éléments de cette liste sont accessibles par leur nom dans l’objet métier de validation. Il suffira donc d’écrire dans le validator :

UIInput composantMotDePasse = (UIInput) component.getAttributes().get( "composantMotDePasse" );

String motDePasse = (String) composantMotDePasse.getValue();

## L'envoi de fichiers avec JSF