UNIVERSITE HASSAN 1ER – SETTAT
FACULTE DES SCIENCES ET TECHNIQUES

J2EE - Java 2 Enterprise Edition

Filière : Génie Informatique

Pr. Youssef Balouki

Département de Mathématiques et
d'Informatique

Année universitaire 2019/2020

I. Introduction au Java EE

I.1 Introduction au Java Framework

Le «**Java Framework**» (*Java 2 Platform*) est composé de trois éditions, destinées à des usages différents :

- **J2ME** : *Java 2 Micro Edition* est prévu pour le développement d'applications embarquées, notamment sur des assistants personnels et terminaux mobiles ;
- **J2SE** : *Java 2 Standard Edition* est destiné au développement d'applications pour ordinateurs personnels ;
- **J2EE** : *Java 2 Enterprise Edition*, destiné à un usage professionnel avec la mise en oeuvre de serveurs.

Chaque édition propose un environnement complet pour le développement et l'exécution d'applications basées sur Java et comprend notamment une machine virtuelle Java (Java virtual machine) ainsi qu'un ensemble de classes.

I.2 Introduction à J2EE

La plate-forme Java Enterprise Edition (Java EE) est un ensemble de spécifications d'API conçues pour fonctionner ensemble lors du développement d'applications Java d'entreprise côté serveur, multi-niveaux, basées sur des composants. Java EE est un standard; il existe plusieurs implémentations des spécifications Java EE. Cela empêche le fournisseur d'être bloqués, car le code développé conformément à la spécification Java EE peut être déployé sur tout serveur d'applications compatible Java EE avec peu ou pas de modifications.

Java EE est développé sous le **Java Community Process** (**JCP**), une organisation responsable du développement de la technologie Java. Les membres de JCP incluent Oracle (l'intendant actuel de la plate-forme Java) et la communauté Java dans son ensemble.

La plate-forme J2EE désigne généralement l'ensemble des services (API) offerts et de l'infrastructure d'exécution. J2EE comprend notamment :

- Les spécifications du **serveur d'application**, c'est-à-dire de l'environnement d'exécution : J2EE définit finement les rôles et les interfaces pour les applications ainsi que l'environnement dans lequel elles seront exécutées. Ces recommandations permettent ainsi à des entreprises tierces de développer des serveurs d'application conformes aux spécifications ainsi définies, sans avoir à redévelopper les principaux services.
- Des services, au travers d'**API**, c'est-à-dire des extensions Java indépendantes permettant d'offrir en standard un certain nombre de fonctionnalités. *Sun* fournit une implémentation minimale de ces API appelée **J2EE SDK** (*J2EE Software Development Kit*).

Dans la mesure où J2EE s'appuie entièrement sur le Java, il bénéficie des avantages et inconvénients de ce langage, en particulier une bonne portabilité et une maintenabilité du code.

De plus, l'architecture J2EE repose sur des composants distincts, interchangeables et distribués qui permet un découpage de l'application et donc une séparation des rôles lors du développement ce qui offre plusieurs avantages :

- La simplicité d'étendre l'architecture ;
- un système reposant sur J2EE peut posséder des mécanismes de haute-disponibilité, afin de garantir une bonne qualité de service ;
- La facilité de la maintenance des applications ;
- la possibilité de s'interfacer avec le système d'information existant grâce à de nombreuses API : JDBC, JNDI, JMS, JCA ...
- la possibilité de choisir les outils de développement et le ou les serveurs d'applications utilisés qu'ils soient commerciaux ou libres.

I.3 Les API de J2EE

Les API de J2EE peuvent se répartir en trois grandes catégories :

- Les **composants**. On distingue habituellement deux familles de composants :
  - Les composants web : **Servlets** et **JSP** (Java Server Pages). Il s'agit de la partie chargée de l'interface avec l'utilisateur (on parle de *logique de présentation*).
  - Les composants métier : EJB (Enterprise Java Beans). Il s'agit de composants spécifiques chargés des traitements des données propres à un secteur d'activité (on parle de *logique métier* ou de *logique applicative*) et de l'interfaçage avec les bases de données.
- Les **services**, pouvant être classés par catégories :
  - Les **services d'infrastructures** : il en existe un grand nombre, définis ci-dessous :
    - **JDBC** (*Java DataBase Connectivity*) est une API d'accès aux bases de données relationnelles.
    - **JNDI** (*Java Naming and Directory Interface*) est une API d'accès aux services de nommage et aux annuaires d'entreprises tels que DNS, NIS, LDAP, etc.
    - **JTA**/**JTS** (*Java Transaction API*/*Java Transaction Services*) est un API définissant des interfaces standard avec un gestionnaire de transactions.
    - **JCA** (*J2EE Connector Architecture*) est une API de connexion au système d'information de l'entreprise, notamment aux systèmes dits «Legacy» tels que les ERP.
    - **JMX** (*Java Management Extension*) fournit des extensions permettant de développer des applications web de supervision d'applications.
  - Les **services de communication** :
    - **JAAS** (*Java Authentication and Authorization Service*) est une API de gestion de l'authentification et des droits d'accès.

- **JavaMail** est une API permettant l'envoi de courrier électronique.
- **JMS** (*Java Message Service*) fournit des fonctionnalités de communication asynchrone (appelées *MOM* pour *Middleware Object Message*) entre applications.
- **RMI-IIOP** est une API permettant la communication synchrone entre objets.

I.4 L'architecture J2EE

L'architecture J2EE permet une grande flexibilité dans le choix de l'architecture de l'application en combinant les différents composants. Ce choix dépend des besoins auxquels doit répondre l'application mais aussi des compétences dans les différentes API de J2EE. L'architecture d'une application se découpe idéalement en au moins trois tiers :

- la partie présentation (cliente) : c'est la partie qui permet le dialogue avec l'utilisateur. Elle peut être composée d'une application standalone, d'une application web ou d'applets
- la partie métier : c'est la partie contenant l'essentiel des traitements de données en se basant dans la mesure du possible sur des API existantes, elle encapsule les traitements (dans des EJB ou des JavaBeans)
- la partie donnée : la couche de données correspondant aux informations de l'entreprise stockées dans des fichiers, dans des bases de données relationnelles ou XML, dans des annuaires d'entreprise ou encore dans des systèmes d'information complexes.

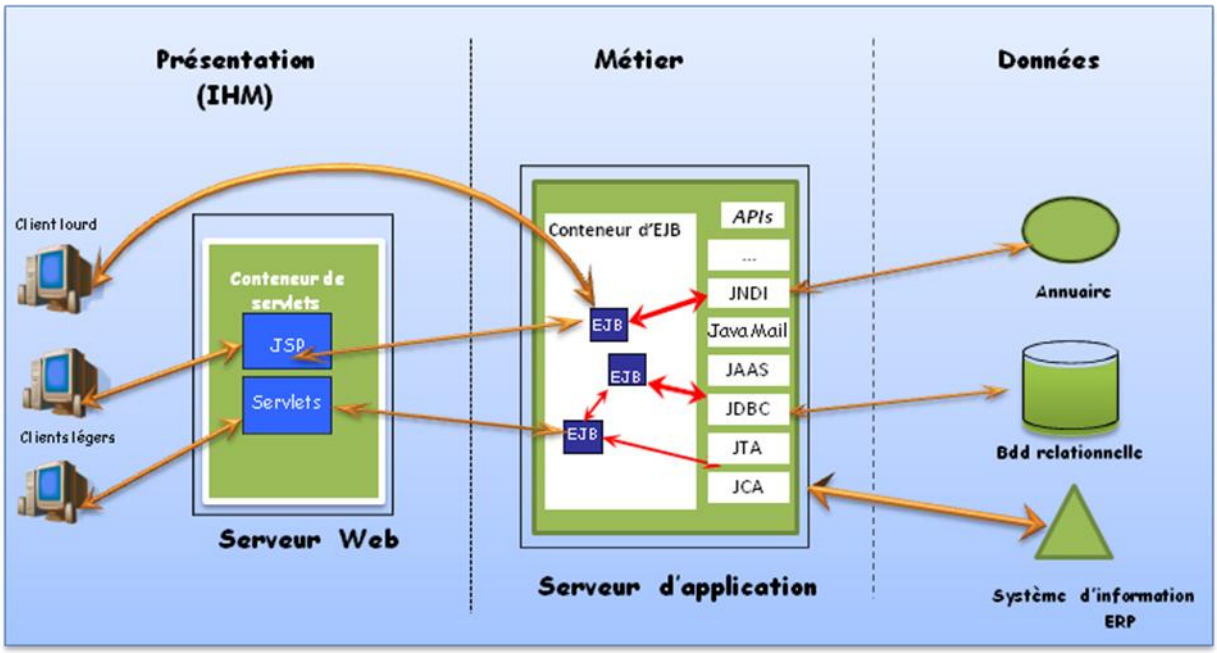


Figure 1 : L'architecture J2EE

J2EE (Java 2 Entreprise Edition) est une plate-forme de développement d'application s'appuyant sur le langage Java. Les architectures J2EE sont utilisées essentiellement pour l'élaboration d'applications présentant une architecture complexe. Ainsi, la mise en place de ce type d'application nécessite l'intervention de plusieurs acteurs et l'utilisation de plusieurs

composants. D'où une exigence accrue en matière d'interopérabilité et l'interconnexion qui s'avère souvent difficile à mettre en place. Ce type d'applications nécessite également des données hétérogènes (souvent de différents formats) pouvant provenir de différentes applications externes.

Le développement d'une telle application doit tenir compte des composants logiciels préexistants mais également des évolutions futures envisageables (changement de base de données, autres types de clients, changement de logique métier, etc). Les applications J2EE sont typiquement utilisées dans le cadre des architectures distantes (type client/serveur). Nous étudierons dans ce qui suit l'architecture des applications J2EE réparties et leurs différents composants.

I.4.1. Architecture multicouche

I.4.1.1. Présentation et organisation d'une architecture multicouche

Les applications J2EE actuelles possèdent une architecture complexe où plusieurs composantes applicatives interviennent. Une telle application est organisée sous forme de plusieurs couches interconnectées que l'on appelle « tiers ».Ce type d'architecture met en place des interconnexions entre des entités différentes génériques et/ou spécialisées dans un traitement particulier réparties, une nouvelle contrainte est à prendre en considération : il s'agit de la composante communication (réseaux et protocoles d'échange).

La figure suivante présente l'architecture J2EE d'une application Web avec la vue « tiers » :

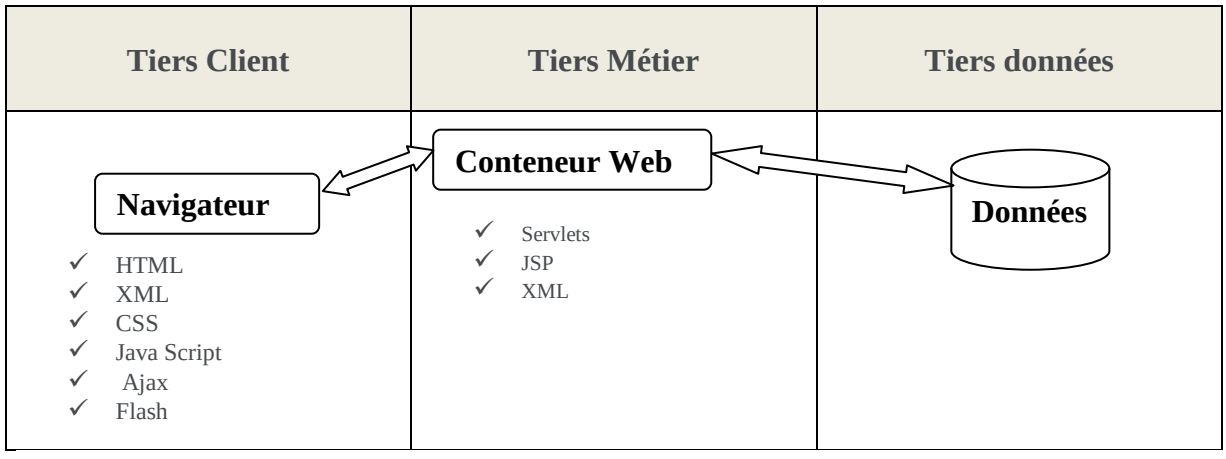


Figure 2 : Architecture J2EE d'une application Web

L'exemple d'architecture décrit au dessus représente une architecture de type 3-tiers. Celle-ci est la plus utilisée dans les petits et moyens projets. Elle permet de séparer le client, le serveur d'application et le réservoir de données.
Dans le cas d'une application plus complexe (applications d'intégration, gestion métier poussée, communication entre différentes applications existantes, etc), l'architecture 3-tiers peut s'avérer insuffisante. Ceci est du à une évolution difficile de ce type d'application à

cause de la concentration des services au niveau d'un seul tiers.
Pour pallier aux divers problèmes cités ci-dessus, une architecture dotée d'une granularité plus fine s'avère plus appropriée.

La figure suivante illustre une telle architecture dite « n-tiers » ou multi-couches :

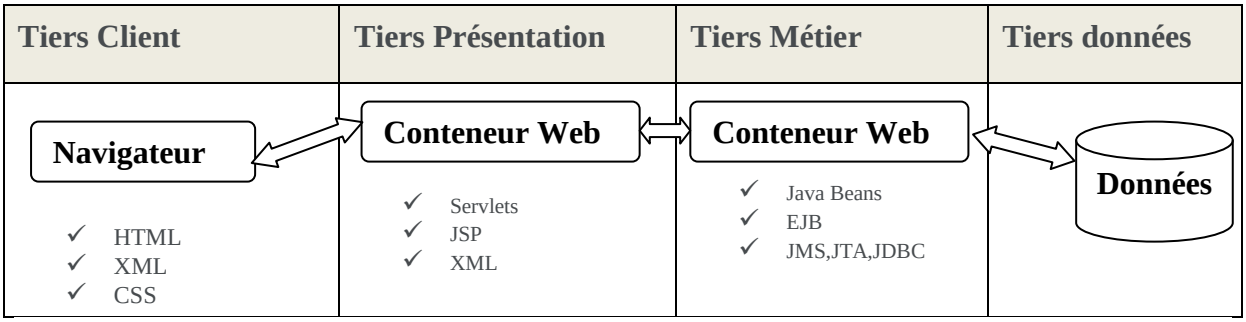


Figure 3 : Architecture de type n-tiers

I.4.1.2. Avantages et inconvénients d'une architecture multicouches:

Les principaux avantages que procure l'élaboration d'une architecture « multi-couches » sont:

- Structure de l'application adaptée à l'architecture de déploiement ciblée,
- Modularité de l'application, - Evolution facilitée de l'application,
- Factorisation du code avec possibilité d'exploiter un framework ou des composants génériques (gain de temps et de performances).

Les inconvénients majeurs sont :

- L'augmentation de la complexité lors de rajout de plusieurs services,
- Nécessité des connaissances tant technique que théorique.

I.4.2. Architecture J2EE Standard

L'architecture J2EE Standard est composée de plusieurs couches. Elle nécessite un travail de personnalisation en fonction du projet à développer avant de permettre la mise en œuvre de l'applicatif à mettre en place. Les couches de base qui la composent sont les suivantes :

- ✓ Couche présentation
- ✓ Couche application
- ✓ Couche métier
- ✓ Couche accès aux données

I.4.2.1. Couche présentation

La couche présentation correspond à la partie de l'application visible et interactive avec les utilisateurs. Elle relaie les requêtes de l'utilisateur à destination de la couche métier, et en retour, lui présente les informations renvoyées par les traitements de cette couche. Il s'agit

donc ici d'un assemblage de services métiers et applicatifs offerts par la couche métier.

I.4.2.2. Couche métier

La couche métier correspond à la partie fonctionnelle de l'application, qui est responsable de l'implémentation de la « logique », décrivant les opérations que l'application opère sur les données en fonction des requêtes des utilisateurs, effectuées au travers de la couche présentation.
Les différentes règles de gestion et de contrôle du système sont mises en œuvre dans cette couche.
La couche métier offre des services applicatifs et métiers à la couche présentation. Pour fournir ces services, elle s'appuie sur les données du système, accessibles au travers des services de la couche d'accès aux données. En retour, elle renvoie à la couche présentation les résultats qu'elle a calculés.

I.4.2.3. Couche d'accès aux données

Elle consiste en la partie gérant l'accès aux de données du système. Ces données peuvent être propres au système, ou gérées par un autre système. La couche métier n'a pas à s'adapter à ces deux cas, ils sont transparents pour elle, et elle accède aux données de manière uniforme.

I.4.2.4. Couche application

La couche application sert de médiateur entre la couche présentation et la couche métier et contrôle l'enchaînement des tâches.

I.5. L'approche MVC

La plupart des plateformes de développement des applications web y compris Java EE n'imposent aucun rangement particulier du code, c'est-à-dire qu'on peut développer n'importe comment. Le problème c'est que si on développe n'importe comment notre code va être mal organisé et il devient très vite difficile de retrouver un bout de code ou une fonction qu'on veut modifier. Pour éviter ce problème, les développeurs ont aujourd'hui tendance à utiliser les bonnes pratiques de développement qu'on appelle les **Patrons de conception** ou **Design Pattern** (en anglais). Un design pattern ou un modèle de conception est en quelque sorte une ligne de conduite qui permet de décrire les grandes lignes d'une solution. Le modèle MVC est un modèle de conception logicielle largement répandu, fort et utile. Néanmoins il faut retenir que c'est un modèle de conception, il est donc indépendant du langage de programmation et fortement recommandé comme modèle pour la plate-forme J2EE.

Le MVC est un modèle d'architecture qui repose sur la volonté de séparer les données, les traitements et la présentation. Ainsi l'application se retrouve segmentée en trois composants essentiels :

- ***La vue :*** C'est la partie de votre code qui s'occupera de la présentation des données à l'utilisateur, elle retourne une vue des données venant du modèle, en d'autres termes

c'est elle qui est responsable de produire les interfaces de présentation de votre application à partir des informations qu'elle dispose, il peut s'agir d'une interface HTML/JSP, mais n'importe quel composant graphique peut jouer ce rôle.

- ***Le contrôleur***, quant à lui, se charge d'intercepter les requêtes de l'utilisateur, d'appeler le modèle puis de retourner une réponse avec l'aide de la couche Modèle et Vue. Il ne doit faire aucun traitement. Il ne fait que de l'interception et de la redirection.

- ***Le modèle*** : C'est la partie de votre code qui exécute la logique métier de votre application. Ceci signifie qu'elle est responsable de récupérer les données, de les convertir selon les concepts de la logique de votre application tels que le traitement, la validation, l'association et tout autre tâche concernant la manipulation des données. Les données peuvent être liées à une base de données, des EJBs, des services Web, etc.

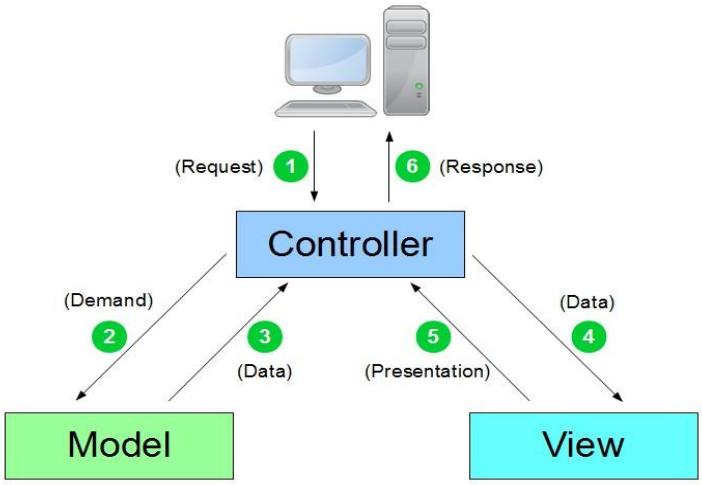


Figure 4 : Modèle MVC

Il est important de noter que les données sont indépendantes de la présentation. En d'autres termes, le modèle ne réalise aucune mise en forme. Ces données pourront être affichées par plusieurs vues. De ce fait le code du modèle est factorisé : il est écrit une seule et unique fois puis réutilisé par chaque vue.

I.5.1. MVC1 vs MVC2

Le MVC1 très pratique, peut se révéler lourd à mettre en place. Ceci à cause de la multitude de contrôleurs à implémenter, car comme nous l'avons expliqué précédemment, chaque vue possède son propre contrôleur(figure5).

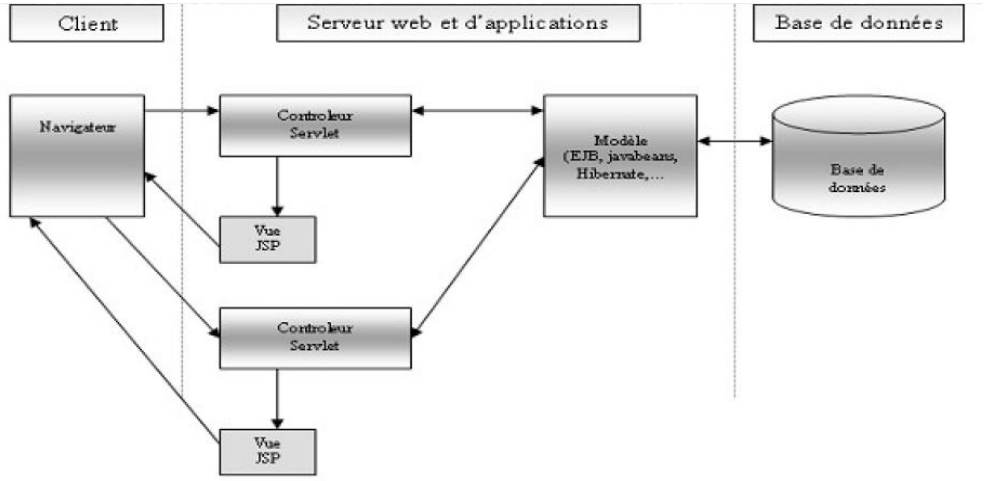


Figure 5 : Modèle MVC1

Dans l'organisation logicielle MVC2, le Servlet est unique, en classe et en instance. Il garantit l'unicité du point d'entrée de l'application. Il prend en charge une partie du contrôle de l'application.
Les contrôleurs MVC se retrouvent alors partiellement déportés dans l'entité « dynamique de l'application » qui assure le contrôle de la dynamique de l'application et qui gère les relations entre les objets métier et la présentation. Les contrôleurs deviennent essentiellement des contrôleurs du dialogue entre l'utilisateur et les objets métiers.

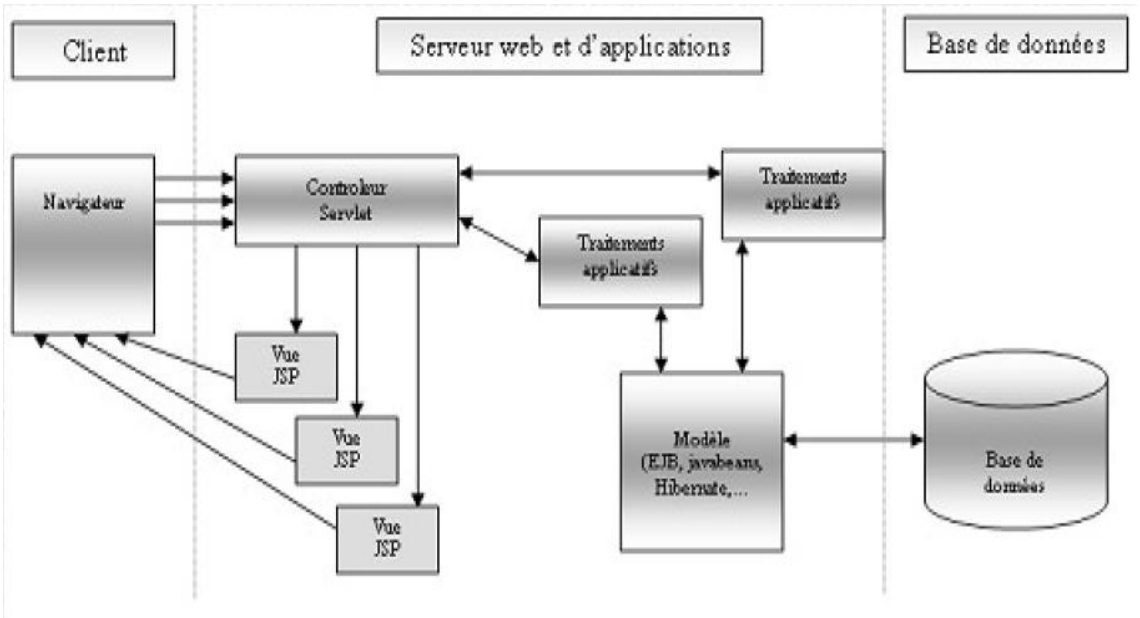


Figure 6 : Modèle MVC2

Dans ce modèle, le cycle de vie d'une requête est le suivant :

1. Le client envoie une requête à l'application. La requête est prise en charge par le Servlet d'entrée.

2. Le Servlet d'entrée analyse la requête et réoriente celle-ci vers un contrôleur adapté.

3. Le contrôleur sélectionné par le Servlet d'entrée est responsable de l'exécution des traitements nécessaires à la satisfaction de la requête. Il sollicite les objets métiers lorsque nécessaire.

4. Les objets métiers fournissent des données au contrôleur.

5. Le contrôleur encapsule les données métiers dans des JavaBeans, sélectionne la JSP qui sera en charge de la construction de la réponse et lui transmet les JavaBean contenant les données métier.

6. La JSP construit la réponse en faisant appel aux JavaBeans qui lui ont été transmis et l'envoie au navigateur.

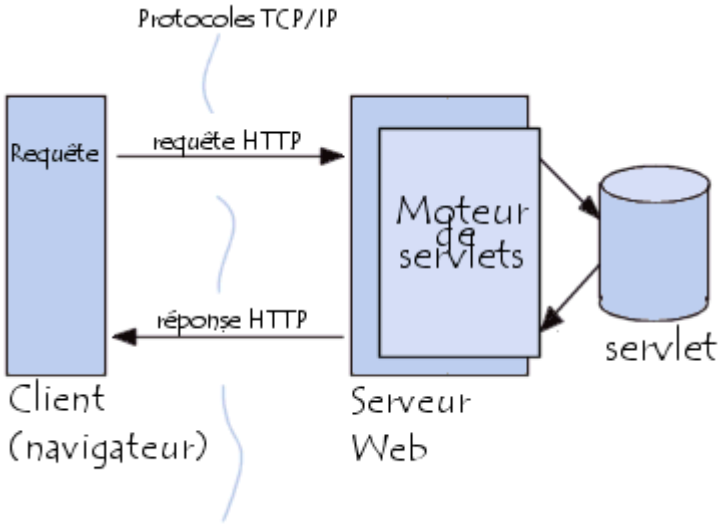
7. Lorsque nécessaire, pour le traitement d'erreurs essentiellement, le Servlet d'entrée traite la requête directement, sélectionne la JSP de sortie et lui transmet par JavaBean les informations dont elle a besoin.

II. Servlets

II.1 Introduction aux servlets :

Les servlets (on dit généralement *une servlet*) sont au serveur Web ce que les applets sont au navigateur pour le client. Les servlets sont donc des applications Java fonctionnant du côté serveur au même titre que les CGI(**Common Gateway Interface**) et les langages de script côté serveur tels que ASP ou bien PHP. Les servlets permettent donc de gérer des requêtes HTTP et de fournir au client une réponse HTTP dynamique (donc de créer des pages Web dynamiques).

Les servlets ont de nombreux avantages par rapport aux autres technologies côté serveur. Tout d'abord, étant donné qu'il s'agit d'une technologie Java, les servlets fournissent un moyen d'améliorer les serveurs web sur n'importe quelle plateforme, d'autant plus que les servlets sont indépendantes du serveur web (contrairement aux modules apache ou à l'API *Netscape Server*). En effet, les servlets s'exécutent dans un **moteur de servlet** utilisé pour établir le lien entre la servlet et le serveur web. Ainsi le programmeur n'a pas à se soucier de détails techniques tels que la connexion au réseau, la mise en forme de la réponse http. On appelle **conteneur de servlet** une classe permettant de manipuler la servlet.



D'autre part les servlets sont beaucoup plus performantes que les scripts, car il s'agit de pseudo-code (code java compilé), chargé automatiquement lors du démarrage du serveur ou bien lors de la connexion du premier client. Les servlets sont donc actives (résidentes en mémoire) et prêtes à traiter les demandes des clients grâce à des threads, tandis qu'avec les langages de script traditionnels un nouveau processus est créé pour chaque requête HTTP. Cela permet donc une charge moins importante au niveau du processeur du serveur (d'autant plus qu'un système de cache peut permettre de stocker les calculs déjà accomplis), ainsi que de prendre une place moins importante en mémoire.

L'un des principaux atouts des servlets est la réusabilité (réutilisation), permettant de créer des composants encapsulant des services similaires, afin de pouvoir les réutiliser dans des applications futures.

Enfin une servlet étant une application Java, peut utiliser toutes les API Java afin de communiquer avec des applications extérieures, se connecter à des bases de données, accèder aux entrée-sorties (fichiers par exemple).

JSDK (Java Servlet Development Kit)

Le **JSDK** (*Java Servlet Development Kit*) est un package contenant l'ensemble des classes et des interfaces nécessaires au développement d'une application java en générale et de servlets en particulier.

De plus le JSDK de Sun contient un serveur web et un moteur de servlets permettant de tester vos créations. Le moteur de servlet founi dans le JSDK est basique (mais gratuit). Il existe de nombreux autres moteurs de servlet beaucoup plus robustes et pouvant s'interfacer avec les principaux serveurs web du marché (les meilleurs sont toutefois payant).

Ou trouver le JSDK

Le JSDK de Sun est disponible gratuitement en téléchargement sur le site de Sun http://java.sun.com/products/servlet.

Les versions du JSDK sont souvent disponibles sur le site des développeurs Java (*Java Developer Connection*), http://developer.java.sun.com, disponible après inscription (gratuite).

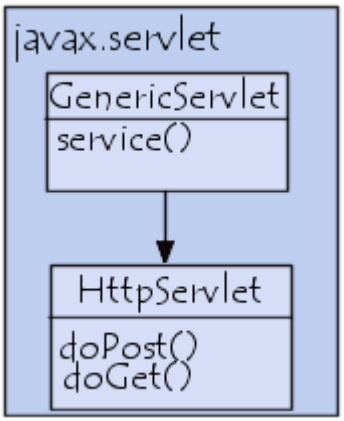
II.2 Architecture du package servlet.jar

Les servlets sont des classes Java implémentant des classes et des interfaces provenant des packages :

- *javax.servlet*, un package générique indépendant du protocole utilisé
- *javax.servlet.http*, un package spécifique au protocole HTTP 1.0
- (accessoirement le package *java.io* doit être importé, notamment pour gérer les exceptions)

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
```

Le package *javax.servlet* est fourni dans le JSDK (*Java Servlet Development Kit*, c'est-à-dire l'API Java Servlet) de Sun, disponible sur http://java.sun.com/products/servlets.



Ainsi toutes les servlets implémentent directement ou indirectement l'interface Servlet, en dérivant une classe qui l'implémente: c'est-à-dire généralement la classe *HTTPServlet*, elle-même issue de *GenericServlet*.

La classe GenericServlet (*javax.servlet.GenericServlet*) définit une classe abstraite (implémentation de base de l'interface *Servlet*).

II.3 Fonctionnement d'une servlet

Lorsqu'une servlet est appelée par un client, la méthode *service()* est exécutée. Celle-ci est le principal point d'entrée de toute servlet et accepte deux objets en paramètres:

- l'objet *ServletRequest* encapsulant la requête du client, c'est-à-dire qu'il contient l'ensemble des paramètres passés à la servlet (informations sur l'environnement du client, cookies du client, URL demandée, ...)
- l'objet *ServletResponse* permettant de renvoyer une réponse au client (envoyer des informations au navigateur). Il est ainsi possible de créer des en-têtes HTTP (headers), d'envoyer des cookies au navigateur du client, ...

II.4 Développer une servlet

Afin de développer un servlet fonctionnant avec le protocole HTTP, il suffit de créer une classe étendant *HttpServlet* (qui implémente elle-même l'interface *Servlet*).

La classe *HttpServlet* (dérivant de *GenericServlet*) permet de fournir une implémentation de l'interface *Servlet* spécifique à HTTP. La classe *HttpServlet* surcharge la méthode *service* en lisant la méthode HTTP utilisée par le client, puis en redirigeant la requête vers une méthode appropriée. Les deux principales méthodes du protocole HTTP étant GET et POST, il suffit de surcharger la méthode adéquate afin de traiter la requête :

- Si la méthode utilisée est GET, il suffit de redéfinir la méthode :
  - public void **doGet**(HttpServletRequest req, HttpServletResponse res);
- Si la méthode utilisée est POST, il suffit de redéfinir la méthode :
  - public void **doPost**(HttpServletRequest req, HttpServletResponse res);
-

```
import javax.servlet.*;
import javax.servlet.http.*;
public class ServletName extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
       throws ServletException {

       // lecture de la requete
       // traitements
       // envoi de la reponse
  }
  public void doPost(HttpServletRequest req, HttpServletResponse
res)
       throws ServletException {

       // lecture de la requete
       // traitements
       // envoi de la reponse
  }

}
```

Lire la requete :

A l'intérieur de la méthode *DoXXX()* (*Doget()* ou *DoPost()* selon la méthode invoquée), la requête de l'utilisateur est passée en paramètres sous forme d'objet (ou plus exactement l'interface) *HttpServletRequest* .

Afin de comprendre son fonctionnement, il est essentiel de connaître la façon selon laquelle les requêtes sont transmises du client au serveur par le protocole HTTP.

Voici les différentes méthodes de l'objet *HttpServletRequest*

Méthode	Description
String getMethod()	Récupère la méthode HTTP utilisée par le client
String getHeader(String Key)	Récupère la valeur de l'attribut Key de l'en-tête
String getRemoteHost()	Récupère le nom de domaine du client
String getRemoteAddr()	Récupère l'adresse IP du client
String getParameter(String Key)	Récupère la valeur du paramètre Key (clé) d'un formulaire. Lorsque plusieurs valeurs sont présentes, la première est retournée
String[] getParameterValues(String Key)	Récupère les valeurs correspondant au paramètre Key (clé) d'un formulaire, c'est-à-dire dans le cas d'une sélection multiple (cases à cocher, listes à choix multiples) les valeurs de toutes les entités sélectionnées
Enumeration getParameterNames()	Retourne un objet *Enumeration* contenant la liste des noms des paramètres passés à la requête
String getServerName()	Récupère le nom du serveur
String getServerPort()	Récupère le numéro de port du serveur

Créer une réponse :

De la même façon, la réponse à fournir à l'utilisateur est représentée sous forme d'objet *HttpServletResponse*.

Voici les différentes méthodes de l'objet *HttpServletResponse*

Méthode	Description
String setStatus(int StatusCode)	Définit le code de retour de la réponse
void setHeader(String Nom, String Valeur)	Définit une paire clé/valeur dans les en-têtes
void setContentType(String type)	Définit le type MIME de la réponse HTTP, c'est-à-dire le type de données envoyées au navigateur
void setContentLength(int len)	Définit la taille de la réponse
PrintWriter getWriter()	Retourne un objet *PrintWriter* permettant d'envoyer du texte au navigateur client. Il se charge de convertir au format approprié les caractères Unicode utilisés par Java
ServletOutputStream getOutputStream()	Définit un flot de données à envoyer au client, par l'intermédiaire d'un objet *ServletOutputStream*, dérivé de la classe *java.io.OutputStream*
void sendredirect(String location)	Permet de rediriger le client vers l'URL *location*

II.5 Cycle de vie d'une servlet

Le cycle de vie d'une servlet est assuré par le conteneur de servlet. Ainsi afin d'être à même de fournir la requête à la servlet, récupérer la réponse ou bien tout simplement démarrer/arrêter la servlet, cette dernière doit possèder une interface (un ensemble de méthodes prédéfinies) déterminée par le JSDK afin de suivre le cycle de vie suivant :

1. le serveur crée un pool de threads auxquels il va pouvoir affecter chaque requête
2. La servlet est chargée au démarrage du serveur ou lors de la première requête
3. La servlet est instanciée par le serveur
4. La méthode *init()* est invoquée par le conteneur
5. Lors de la première requête, le conteneur crée les objets Request et Response spécifiques à la requête
6. La méthode *service()* est appelée à chaque requête dans un nouveau thread. Les objets *Request* et *Response* lui sont passés en paramètre
7. Grâce à l'objet *Request*, la méthode *service()* va pouvoir analyser les informations en provenance du client
8. Grâce à l'objet *Response*, la méthode *service()* va fournir une réponse au client
9. La méthode *destroy()* est appelée lors du déchargement de la servlet, c'est-à-dire lorsqu'elle n'est plus requise par le serveur. La servlet est alors signalée au *garbage collector* .

Interface d'une servlet :

Pour créer une servlet il est indispensable de mettre en oeuvre l'interface *javax.servlet.**Servlet*** permettant au conteneur d'assurer le cycle de vie de la servlet. La mise en place de l'interface (généralement réalisée en étendant *javax.servlet.GenericServlet* ou *javax.servlet.HttpServlet*) fait appel à cinq méthodes :

- la méthode *init()*
- la méthode *service()*
- la méthode *getServletConfig()*
- la méthode *getServletInfo()*
- la méthode *destroy()*

La méthode *init() :*

La méthode *init()* est appelée par le conteneur à chaque instanciation de la servlet :

- public void init(ServletConfig config) throws ServletException

Lors de l'instanciation, le conteneur de servlet passe en argument à la méthode *init()* un objet *ServletConfig* permettant de charger des paramètres de configuration propres à la servlet.

En cas d'anomalie lors de l'appel de la méthode *init()*, celle-ci renvoie une exception de type *ServletException* et la servlet n'est pas initialisée.

La méthode *getServletConfig() :*

La méthode *getServletConfig()* : public ServletConfig getservletConfig()

La méthode *service() :*

La méthode *service()* est appelée à chaque requête d'un client :

- public void service(ServletRequest req, ServletResponse res) throws ServletException, IOException

Les objets *ServletRequest* et *ServletResponse* sont automatiquement passés en paramètre à la méthode *service()* par le conteneur.

La méthode *destroy() :*

La méthode *destroy()* est appelée par le conteneur lors de l'arrêt du serveur Web.

- public void destroy()

La méthode *destroy()* attend la fin de tous les threads lancés par la méthode *service()* avant de mettre fin à l'exécution de la servlet.

Fournir des informations sur la servlet :

Certaines applications (comme *Java Web Server Administration Tool*) permettent d'obtenir des informations sur la servlet et de les afficher.

La méthode *getServletInfo()* lorsqu'elle est surchargée permet de retourner cette information sous forme d'objet String (elle retourne *null* par défaut).

- public String getServletInfo()

1- Premiere servlet

Voici un exemple simple de servlet dont le seul but est d'afficher du texte sur le navigateur du client :

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class PremiereServlet extends HttpServlet {

  public void init() {
  }

  public void doGet(HttpServletRequest req, HttpServletResponse res)
       throws ServletException, IOException {

       res.setContentType("text/html");
       PrintWriter out = res.getWriter();

       out.println("<HTML>");
       out.println("<HEAD><TITLE> Titre </TITLE></HEAD>");
       out.println("<BODY>");
       out.println("Ma première servlet");
       out.println("</BODY>");
       out.println("</HTML>");
       out.close();
  }

}
```

La première étape consiste à importer les packages nécessaires à la création de la servlet, il faut donc importer *javax.servlet*, *javax.servlet.http* et *javax.io*

Afin de mettre en place l'interface *Servlet* nécessaire au conteneur de servlet, il existe plusieurs possibilités:

- Dériver la classe *GenericServlet* et redéfinir les méthodes dont on a besoin
- Dériver la classe *HttpServlet* et redéfinir les méthodes dont on a besoin

Dans la servlet ci-dessus, la classe *HttpServlet* a été étendue

Lorsque la servlet est instanciée, il peut être intéressant d'effectuer des opérations qui seront utiles tout au long du cycle de vie de la servlet (se connecter à une base de données, ouvrir un fichier, ...). Pour ce faire, il s'agit de redéfinir la méthode *init()* de la servlet :public void init() {}

A chaque requête, la méthode *service()* est invoquée. Celle-ci détermine le type de requête dont il s'agit, puis transmet la requête et la réponse à la méthode adéquate (*doGet()* ou *doPost*). Dans notre cas, on ne s'intéresse qu'à la méthode GET, c'est la raison pour laquelle la méthode *doGet()* a été surchargée

```
public void doGet(HttpServletRequest req, HttpServletResponse res)
       throws ServletException, IOException {
}
```

L'objet *HttpServletRequest* permet de connaître les éventuels paramètres passés à la servlet (dans le cas d'un formulaire HTML par exemple), mais l'exemple ci-dessus n'en a pas l'utilité.

Par contre l'objet *HttpServletResponse* permet de renvoyer une page à l'utilisateur. La première étape consiste à définir le type de données qui vont être envoyées au client. Généralement il s'agit d'une page HTML, la méthode *setContentType()* de l'objet *HttpServletResponse* doit donc prendre comme paramètre le type MIME associé au format HTML (*text/html*) :

- `res.setContentType("text/html");`

Ensuite la création d'un objet *PrintWriter* grâce à la méthode *getWriter()* de l'objet *HttpServletResponse* permet d'envoyer du texte formatté au navigateur (pour envoyer un flot de données, il faudrait utiliser la méthode *getOutputStream()*)

- `PrintWriter out = res.getWriter();`

Enfin il faut utiliser la méthode *println()* de l'objet *PrintWriter* afin d'envoyer les données textuelles au navigateur, puis fermer l'objet *PrintWriter* lorsqu'il n'est plus utile avec sa méthode *close()*

```
out.println("<HTML>");
out.println("<HEAD><TITLE> Titre </TITLE></HEAD>");
out.println("<BODY>");
out.println("Ma première servlet");
out.println("</BODY>");
out.println("</HTML>");
out.close();
```

II.6 Déploiement d'une servlet :

Une servlet n'est pas une classe Java comme les autres, il s'agit d'un composant Java EE qui va être pris en charge par le serveur d'application. Le serveur d'application a besoin de savoir pour quelle(s) URL cette servlet sera responsable de traiter les requêtes et de fournir la réponse.

La méthode la plus simple pour configurer le déploiement d'une servlet consiste à utiliser l'annotation **@WebServlet** sur la classe.

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;


@WebServlet("/hello")
public class HelloServlet extends HttpServlet {


 @Override
  protected void doGet(HttpServletRequest req, HttpServletResponse resp)
              throws ServletException, IOException {
    String name = req.getParameter("name");
   resp.setContentType("text/html");
    resp.setCharacterEncoding("utf-8");
    resp.getWriter().write("Hello " + name + "!");}
}
```

Pour la servlet ci-dessus, l'annotation @WebServlet précise le motif de l'URL (URL pattern) pour lequel la servlet devra être sollicitée (dans cet exemple « /hello »). Une fois l'application déployée dans un serveur de test en local, une requête de la forme :

```
https://localhost:8080/[nom de l'application]/hello?name=Youssef
```

répond : Hello Youssef !

Chemin absolu d'URL dans une application Web

Le motif d'URL dans l'exemple précédent est « /hello ». Le / est obligatoire est dénote donc un chemin absolu. Néanmoins dans une servlet, un chemin absolu commence non pas à la racine du serveur mais à la racine de l'application.

Ainsi pour une application déployée dans le contexte racine **« /monappli »**, une servlet dont le motif d'URL est **« /hello »** sera accessible par le chemin **« /monappli/hello »** et non pas « /hello ».

Cette astuce est très pratique car elle dispense les servlets de connaître le contexte racine d'une application. Cela peut néanmoins entraîner une certaine confusion chez les développeurs entre les URL qui seront effectivement retournées au client (comme les liens dans une page Web par exemple) et les URL manipulées côté serveur.

Motif d'URL d'une Servlet

Comme nous l'avons vu dans la section précédente, une servlet pour être déployée a besoin d'un ou plusieurs motifs d'URL indiquant le chemin des requêtes qu'elle prend en charge. Il existe plusieurs syntaxes qui sont toutes équivalentes :

```
@WebServlet("/hello")
@WebServlet({"/hello"})
@WebServlet(urlPatterns={"/hello"})
```

Il est possible de donner plusieurs motifs d'URL indiquant que la même servlet peut être sollicitée à partir de chemins différents.

```
@WebServlet({"/hello", "/bonjour"})
@WebServlet(urlPatterns={"/hello", "/bonjour"})
```

Enfin, il est possible d'utiliser le caractère générique *. Par contre son utilisation est limitée car il ne peut apparaître que **comme premier ou dernier** élément d'un motif :

```
// Toutes les URL se terminant par .html
@WebServlet("*.html")
// Toutes les URL commençant par /hello/
@WebServlet("/hello/*")
```

Utilisation du fichier de déploiement web.xml

Nous avons vu que l'annotation ***@WebServlet*** permet d'indiquer comment une servlet doit être déployée dans le serveur. S'il préfère, le développeur a la possibilité de spécifier ces informations dans le fichier de déploiement ***web.xml*** plutôt que d'utiliser une annotation.

Les annotations n'ont été introduites dans le langage Java que depuis la version 5. Pour J2EE, le recours au fichier de déploiement ***web.xml*** était la seule façon de déclarer les servlets. Ce fichier reste donc encore aujourd'hui très utilisé par les développeurs, particulièrement pour déclarer des servlets provenant de frameworks et de bibliothèques tiers. Pour déclarer une servlet dans une fichier ***web.xml***, il suffit d'associer un identifiant avec le nom de la classe de la servlet. Ensuite, on précise un ou des motifs d'URL pour cette servlet de la façon suivante :

```
<web-app
 …..
  <!-- la déclaration de la servlet -->
  <servlet>
    <servlet-name>nomLogiqueDeLaServlet </servlet-name>
    <!-- le nom de la classe implémentant la servlet (précédé du nom du package) -->
    <servlet-class>le.nom.complet.de.la.classe.de.la.Servlet</servlet-class>
  </servlet>

  <!-- l'association de la servlet avec un motif d'URL -->
  <servlet-mapping>
    <servlet-name>nomLogiqueDeLaServlet</servlet-name>
    <!-- le motif d'url (par exemple *.html ou /servlet) -->
    <url-pattern>/ma-servlet</url-pattern>
  </servlet-mapping>
</web-app>
```

Remarque : Java EE est une plate-forme pour laquelle les développeurs d'applications implémentent des **composants** (Web, métier, …). Pour fournir les informations de déploiement de ces composants, nous verrons qu'il est toujours possible d'utiliser des annotations ou des descripteurs de déploiement (des fichiers XML). L'utilisation d'annotations offre l'avantage de déclarer les informations au plus près du code. Au contraire, le descripteur de déploiement centralise l'ensemble des informations pour une application. Il permet une plus grande souplesse au détriment de la verbosité et de la nécessité de maintenir un fichier XML.

II.7 Les cookies

Les cookies représentent un moyen simple de stocker temporairement des informations chez un client, afin de les récupérer ultérieurement. Concrètement il s'agit de fichiers texte stockés sur le disque dur du client après réception d'une réponse HTTP contenant des champs appropriés dans l'en-tête.

Les cookies font partie des spécifications du protocole HTTP. Le protocole HTTP permet d'échanger des messages entre le serveur et le client à l'aide de requêtes HTTP et de réponses HTTP.

Les requêtes et réponses HTTP contiennent des en-têtes permettant d'envoyer des informations particulières de façon bilatérale. Un de ces en-têtes est réservé à l'écriture de fichiers sur le disque: les cookies.

L'en-tête HTTP réservé à l'utilisation des cookies s'appelle Set-Cookie, il s'agit d'une simple ligne de texte de la forme:

Set-Cookie : NOM=VALEUR; domain=NOM_DE_DOMAINE; expires=DATE

Il s'agit donc d'une chaîne de caractères commençant par *Set-Cookie* : suivie par des paires clés-valeur sous la forme CLE=VALEUR et séparées par des virgules.

L'API servlet de Java propose un objet permettant de gérer de façon quasi-transparente l'usage des cookies, il s'agit de l'objet ***Cookie*.**

L'objet Cookie

La classe *javax.servlet.http.Cookie* permet de créer un objet *Cookie* encapsulant toutes les opérations nécessaires à la manipulation des cookies.

Ainsi, le constructeur de la classe *Cookie* crée un cookie avec un nom et une valeur initiaux passés en paramètre. Il est toutefois possible de modifier la valeur de ce cookie ultérieurement grâce à sa méthode *setValue()*.

Conformément à la norme HTTP 1.1, le nom du cookie doit être une chaîne de caractères ne contenant aucun caractère spécial défini dans la RFC 2068 (Il vaut mieux donc utiliser des caractères alphanumériques uniquement). Les valeurs par contre peuvent inclure tous les caractères hormis les espaces ou chacun de ces caractères : [ ] ( ) = , " / ? : ;

Envoi du cookie

L'envoi du cookie vers le navigateur du client se fait grâce à la méthode *addCookie()* de l'objet *HttpServletResponse*.

```
void AddCookie(Cookie cookie)
```

Remarque :

Étant donné que les cookies sont stockés dans les en-têtes HTTP, et que celles-ci doivent être les premières informations envoyées, la création du cookie doit se faire avant tout envoi de données au navigateur (le cookie doit être créé avant toute écriture sur le flot de sortie de la servlet)

```
Cookie MonCookie = new Cookie("nom", "valeur");
response.addCookie(MonCookie);
```

Récupération des cookies du client

Pour récupérer les cookies provenant de la requête du client, il suffit d'utiliser la méthode *getCookies()* de l'objet *HttpServletRequest*

Cookie[] getCookies()
Cette méthode retourne un tableau contenant l'ensemble des cookies présents chez le client. Il est ainsi possible de parcourir le tableau afin de retrouver un cookie spécifique grâce à la méthode *getName()* de l'objet *Cookie()*.

Récupération de la valeur d'un cookie

La récupération de la valeur d'un cookie se fait grâce à la méthode *getValue()* de l'objet *Cookie*

```
String Valeur = Cookie.getValue()
```

L'exemple suivant permet de récupérer la valeur d'un cookie spécifique (dont le nom est *"LeCookieQueJeCherche"*) et d'afficher sa valeur sur le navigateur du client :

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.net.*;

public class AfficheMonCookie extends HttpServlet {
public void doGet (HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException    {
       //...

     Cookie[] cookies = request.getCookies();
     for(i=0; i < cookies.length; i++) {
       Cookie MonCookie = cookies[i];
      if (MonCookie.getName().equals("LeCookieQueJeCherche")) {
         String Valeur = cookies[i].getValue();
      }
}

// ecriture de la reponse

       response.setContentType("text/html");
```

```
        PrintWriter out = response.getWriter();
        out.println("<html><head>");
        out.println("<title>Mon Cookie</title>");
        out.println("</head><body>");
        out.println("Voici la valeur de mon cookie : " + Valeur);
        out.println("</body></html>");
}
}
```

Ajouter des paramètres à un cookie

Voici l'ensemble des méthodes publiques de l'objet *Cookie* permettant d'obtenir des informations sur le cookie ou bien de préciser certains paramètres :

Méthode	Description
Cookie(String Name, String Value)	Constructeur créant un objet Cookie de nom *Name* et de valeur *Value*)
String getDomain()	Retourne le domaine pour lequel le cookie est valide
void setDomain(String Domain)	Définit le domaine pour lequel le cookie est valide
int getMaxAge()	Retourne la durée de validité du cookie (en secondes)
void setMaxAge(int duree)	Définit la durée de validité du cookie (en secondes)
String getName()	Retourne le nom du cookie
String getPath()	Retourne le chemin pour lequel le cookie est valide
void setPath(String Chemin)	Définit le chemin pour lequel le cookie est valide
boolean getSecure()	Retourne *true* si le cookie doit être envoyé uniquement sur ligne sécurisée, sinon *false*
boolean setSecure(Boolean flag)	Définit si le cookie doit être envoyé sur une ligne sécurisée (SSL)
String getValue()	Retourne la valeur du cookie
void setValue(String Valeur)	Redéfinit la valeur du cookie
int getVersion()	Retourne la version du cookie
void setVersion(int Version)	Définit la valeur du cookie

II.7 Les Sessions

L'objet **HttpSession** permet de mémoriser les données de l'utilisateur, grâce à une structure similaire à une table de hachage, permettant de relier chaque ***id*** de session à l'ensemble des informations relatives à l'utilisateur.

Ainsi en utilisant un mécanisme tel que les cookies (permettant d'associer une requête à un id) et l'objet HttpSession (permettant de relier des informations relatives à l'utilisateur à un id), il est possible d'associer facilement une requête aux informations de session !

L'objet *HttpSession* s'obtient grâce à la méthode *getSession()* de l'objet *HttpServletRequest*.

Gérer les sessions

La gestion des sessions se fait de la manière suivante :

- Obtenir l'ID de session
  - Si GET: en regardant dans la requête
  - Si POST: en regardant dans les en-têtes HTTP
  - Sinon dans les cookies
- Vérifier si une session est associé à l'ID
  - Si la session existe, obtenir les informations
  - Sinon
    - Générer un *ID de Session*
      - Si le navigateur du client accepte les cookies, ajouter un cookie contenant l'ID de session
      - Sinon ajouter l'ID de session dans l'URL
    - Enregistrer la session avec l'ID nouvellement créé

Obtenir une session

La méthode *getSession()* de l'objet *HttpServletRequest* permet de retourner la session relative à l'utilisateur (l'identification est faite de façon transparente par cookies ou réécriture d'URL):

HttpSession getSession(boolean create)
L'argument *create* permet de créer une session relative à la requête lorsqu'il prend la valeur *true*.

Remarque :

Etant donnée que les cookies sont stockés dans les en-têtes HTTP, et que celles-ci doivent être les premières informations envoyées, la méthode *getSession()* doit être appelée avant tout envoi de données au navigateur (la méthode doit être invoquée avant toute écriture sur le flot de sortie de la servlet)

L'exemple suivant récupère la session associée à la requête de l'utilisateur et la crée si elle n'existe pas déjà :

```
public class Caddie extends HttpServlet {
public void doGet (HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException        {

                    //*** Recupere la session****
  HttpSession session = request.getSession(true);

              ...
              //*** Ecrit la reponse
              out = response.getWriter();
              ...
 }
}
```

Obtenir des informations d'une session

Pour obtenir une valeur précédemment stockée dans l'objet *HttpSession*, il suffit d'utiliser la méthode *getAttribute()* de l'objet *HttpSession* (celle-ci remplace dans la version 2.2 de l'API servlet la méthode *getValue()* qui était utilisée dans les versions 2.1 et inférieures).

Object getAttribute("***cle***")

La méthode *getAttribute()* retourne un objet (de type *Object*), il faut donc effectuer un surtypage pour obtenir un type élémentaire de données (un entier sera renvoyé sous forme d'objet *Integer* qu'il faudra convertir en *int*). Si l'attribut passé en paramètre n'existe pas, la méthode *getAttribute()* retourne la valeur *null*.

public class Caddie extends HttpServlet {

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

                      //*** Recupere la session
HttpSession session = request.getSession(true);

                  //*** Recupere l'age de l'utilisateur
int Age = (int)session.getAttribute("Age");
if (Age != null) {

                         //***faire quelque chose
                        //*** Ecrit la réponse
      out = response.getWriter();
 }

else {

       Age = new Integer(...);
       // *** faire quelque chose d'autre
       //*** Ecrit la reponse
       out = response.getWriter();
}
}
```

```
}
```

Stocker des informations dans une session

Le stockage d'informations dans la session est similaire à la lecture. Il suffit d'utiliser la méthode *setAttribute()* (*putvalue()* pour les versions antérieures à la 2.2) en lui fournissant comme attributs la clé et la valeur associée.

```
Object setAttribute("cle","valeur")
```

Invalider une session

D'une manière générale, il est préférable de laisser une session se terminer seule (par expiration par exemple). Cependant, dans certains cas il peut être utile de supprimer manuellement une session (si l'utilisateur achète le contenu du caddie par exemple). Pour supprimer une session, il suffit de faire appel à la méthode *invalidate()* de l'objet *HttpSession*

```
public class Caddie extends HttpServlet {
public void doGet (HttpServletRequest request, HttpServletResponse
response)throws ServletException, IOException        {

            // Recupere la session
 HttpSession session = request.getSession(true);
session.invalidate();

}
}
```

Autres méthodes de HttpSession :

Méthode	Description
long getCreationTime()(	Retourne l'heure de la création de la session
Object getAttributes(String Name)	Retourne l'objet stocké dans la session sous le nom *Name*, *null* s'il n'existe pas
String getId()	Génère un identifiant de session
long getCreationTime()(	Retourne la date de la requête précédente pour cette session
String[] getValueNames()(	Retourne un tableau contenant le nom de toutes les clés de la session
Object getValue(String Name)	Retourne l'objet stocké dans la session sous le nom *Name*, *null* s'il n'existe pas
void invalidate()	Supprime la session
boolean isnew()(	Retourne *true* si la session vient d'être créée, sinon *false*
void putValue(String Name, Object Value)	Stocke l'objet *Value* dans la session sous le nom *Name*

void removeValue(String Name)	Supprime l'élément *Name* de la session
void setAttribute(String Name, Object Value)	Stocke l'objet *Value* dans la session sous le nom *Name*
int setMaxInactiveInterval(int interval)	Définit l'intervalle de temps maximum entre deux requête avant que la session n'expire
int getMaxInactiveInterval(int interval)	Retourne l'intervalle de temps maximum entre deux requête avant que la session n'expire

II.8 Notion de variable d'environnement

Les variables d'environnement sont, comme leur nom l'indique, des données stockées dans des variables permettant au programme d'avoir des informations sur son environnement. L'environnement, dans le cas d'une servlet est:

- Le serveur
- Le client

Java permet d'accéder à ces variables grâce à des méthodes de l'objet *HttpServletRequest*.

Méthodes spécifiques des servlets :

Java fournit une méthode correspondant à chaque variable d'environnement. Voici la liste des méthodes de l'objet *HttpServletRequest* permettant de récupérer des variables d'environnement :

Variable d'environnement	Méthode associée
AUTH_TYPE	getAuthType()
CONTENT_LENGTH	getContentLength()
CONTENT_TYPE	getContentType()
HTTP_ACCEPT_LANGUAGE	getHeader("Accept")
HTTP_REFERER	getHeader("Referer")
HTTP_USER_AGENT	getHeader("User-Agent")
PATH_INFO	getPathInfo()
PATH_TRANSLATED	getPathTranslated()
QUERY_STRING	getQueryString()
REQUEST_METHOD	getRequestMethod()
REMOTE_ADDR	getRemoteAddr()
REMOTE_HOST	getRemoteHost()
REMOTE_USER	getRemoteUser()
SCRIPT_NAME	getScriptName()
SERVER_NAME	getServerName()
SERVER_PROTOCOL	getServerProtocol()
SERVER_PORT	getServerPort()

III. Java Server Pages

III. 1- Introduction

Comme nous l'avons vu, jusqu'à présent, les servlets Java sont très intéressantes afin de fournir des traitements, exécutés coté serveur, sur les pages HTML à renvoyer au client. Cela permet, notamment, de renvoyer des pages dont le contenu provient de bases de données. Mais peut-être avez vous trouvé certains aspects de cette technologie, ennuyeux. Plus précisément, je pense à l'intégration du code HTML au sein de la classe Java.

En effet, chaque tag Java est renvoyé au client, par l'intermédiaire de *println*. Cela n'est pas franchement d'une lisibilité absolue. De plus, imaginez que vous ayez un infographiste dans votre équipe. Lui demander d'intervenir au niveau d'une classe de code Java n'est peut-être pas la meilleure idée.

En réponse à ce besoin de simplification de l'intégration des deux langages (HTML et Java), SUN Microsystems vous propose une seconde technologie : les JSP (Java Server Pages). Une page JSP ressemble plus à une page HTML qu'à une classe Java. Elle comporte malgré tout des bouts de code. Un autre avantage sérieux à utiliser cette technologie réside dans le fait qu'elle est automatiquement transformée par le serveur d'applications en servlet Java et déployée sur le serveur.

De plus, différentes constructions syntaxiques vous sont fournies pour répondre aux différentes exigences que vous pourriez avoir. Nous allons maintenant, reprendre ces différents points un à un dans le détail.

III. 2- Votre première JSP

Afin de comprendre comment marche une page JSP, nous allons commencer notre étude par un petit exemple simple. Cette page JSP va afficher six titres de niveaux différents via les tags HTML *<H1>* à *<H6>*. Mais attention, ces tags ne vont pas explicitement être tapés, mais au contraire, générés via une boucle.

Nous verrons aussi comment le serveur d'applications va prendre en charge ce nouvel élément. En fait les choses sont bien plus simples que pour les servlets.

Aspects syntaxiques élémentaires

Une page JSP se présente donc un peu comme une page HTML. Souvent, il y a plus de code HTML que de code Java. Mais des constructions nouvelles apparaissent. On peut facilement les reconnaître : elles commencent par les deux caractères *<%* et se terminent par *%>*. Ainsi, ce premier exemple (voir quelques lignes plus bas) commence par une construction indiquant quel est le langage à utiliser pour traiter la page ainsi qu'une demande d'utilisation de package. Cette première ligne est plus précisément une déclarative (elle commence par <%@).

D'autres bouts de code Java sont insérés dans la page, afin d'opérer la boucle sur le six niveaux de titres. Noter aussi que la date et l'heure exacte sont insérer dans la page via la construction *<%=*. En fait cette construction revient à taper *<% out.println(new Date()); %>*, mais elle est plus concise. En conséquence, on est bien d'accord qu'il ne faudra jamais utiliser un point-virgule terminal dans ce type d'expressions.

```
<%@ page language="Java" import="java.util.*" %>
<HTML>
    <HEAD>
        <TITLE>First.jsp</TITLE>
    </HEAD>
    <BODY>
        <H1 Align="center">Time is : <%= new Date() %></H1>

        <% for(int i=1; i<=6; i++) { %>
          <H<%=i%> align="center">Heading <%=i%></H<%=i %>>
        <% } %>
        <HR>
    </BODY>
</HTML>
```

Histoire de mieux comprendre les choses, notons que nous pouvons réécrire cette page JSP de différentes manières. En effet, comme expliqué précédemment, la construction *<%= i %>* peut s'écrire différemment : *<% out.println(i); %>*. Du coup, le bloc *<H<%= i %> align="center">* pourrait s'écrire *<% out.println("<H" + i + " align=\"center\">"); %>*. Et ainsi, de proche en proche, on pourrait envisager plusieurs versions sensiblement différentes telles que celle-ci. Dans tous les cas, le comportement restera le même.

```
<%@ page language="Java" import="java.util.*" %>
<HTML>
    <HEAD>
        <TITLE>First.jsp</TITLE>
    </HEAD>
    <BODY>
        <H1 Align="center">Time is : <%= new Date() %></H1>

        <%
            for(int i=1; i<=6; i++) {
out.println("<H" + i + " align=\"center\">Heading " +
                        i + "</H" + i + ">");
            }
        %>
        <HR>
    </BODY>
</HTML>
```

Et si l'on pousse le vis jusqu'au bout, l'ensemble du code de cette page pourrait être généré par un bloc de code Java. C'est ce que va, en gros, faire le serveur d'applications pour exécuter votre page JSP.

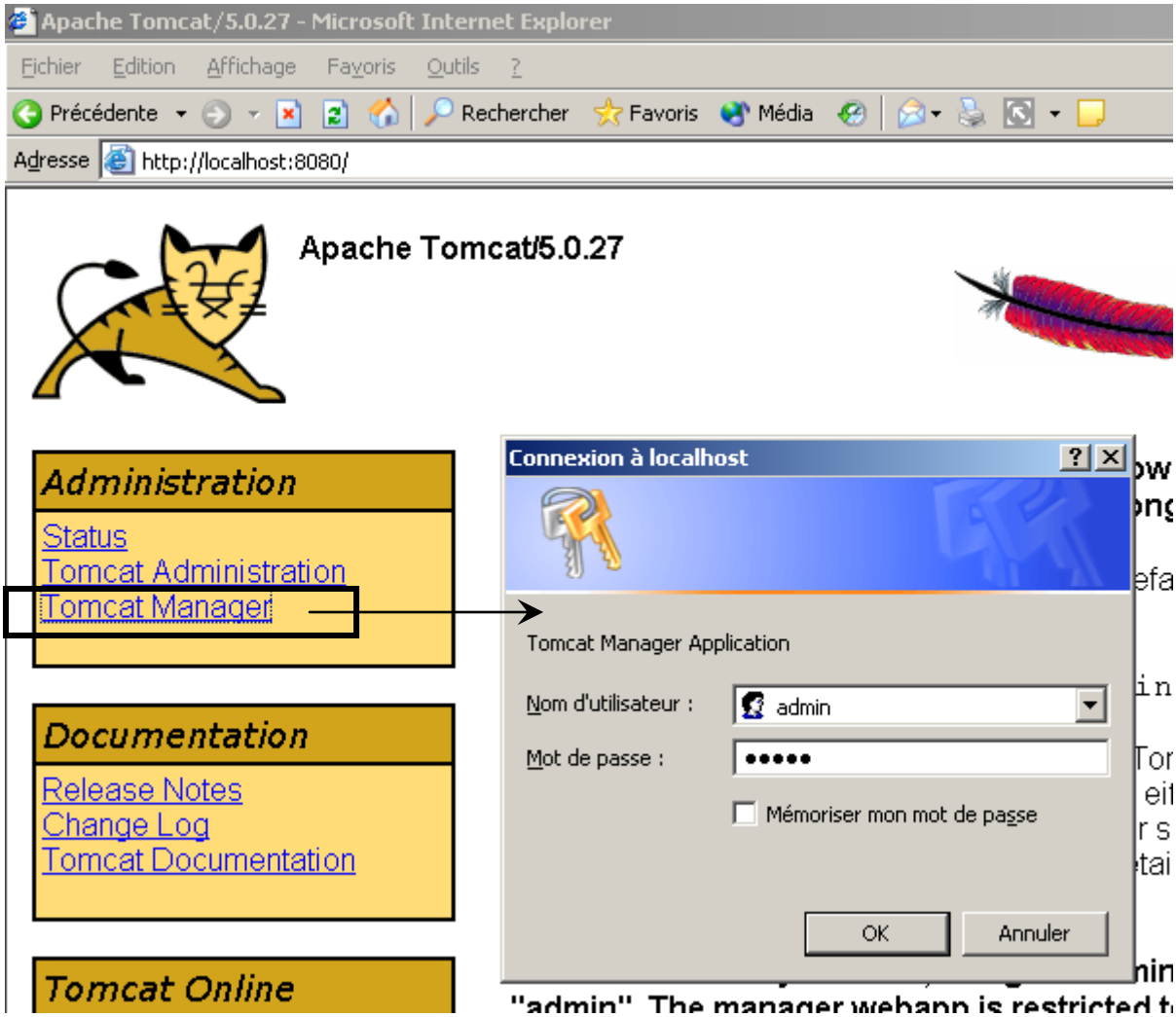
Transformation implicite en Servlet et déploiement

Pour tester votre page JSP, les choses sont très simples : il vous suffit de placer le fichier d'extension ".jsp" sur le répertoire contenant les pages HTML du serveur Web fournit avec le serveur d'applications.

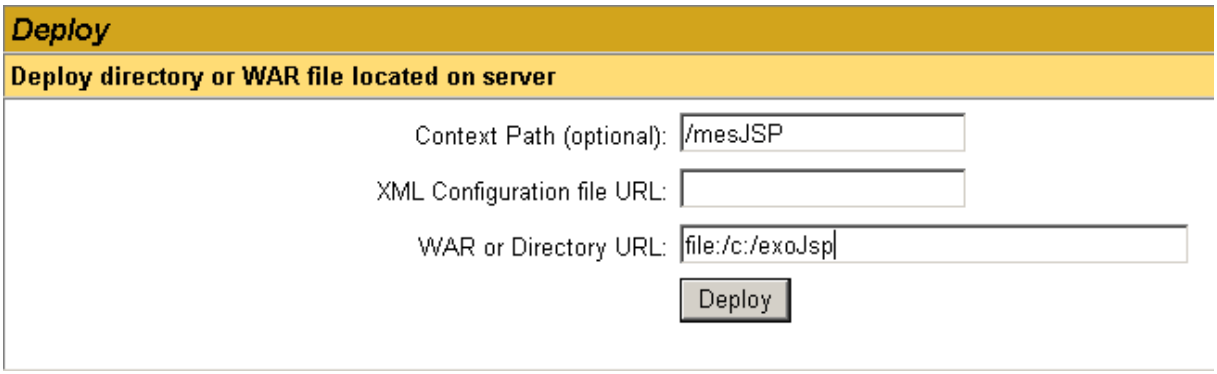
Si vous utilisez le serveur J2EE de SUN Microsystem (en version 3.1), placez vos fichiers dans le répertoire *"C:\j2sdkee1.3.1\public_html"*.

Si vous utilisez Tomcat, Placer ce fichier dans un répertoire quelconque exemple «c:\exoJsp » puis déployer votre application en utilisant la page d'administration de TOMCAT.

- A près avoir démarré Tomcat, lancer le navigateur, puis taper http://localhost:8080/
- Cliquez sur le lien Tomcat Manager.
- Introduire le nom d'utilisateur et le mot de passe.



- Taper le nom du contexte Path (Alias) ici : /mesJSP
- Taper le chemin complet de votre dossier dans la zone WAR or Directoty URL
- Valider en cliquant sur le bouton Deploy.

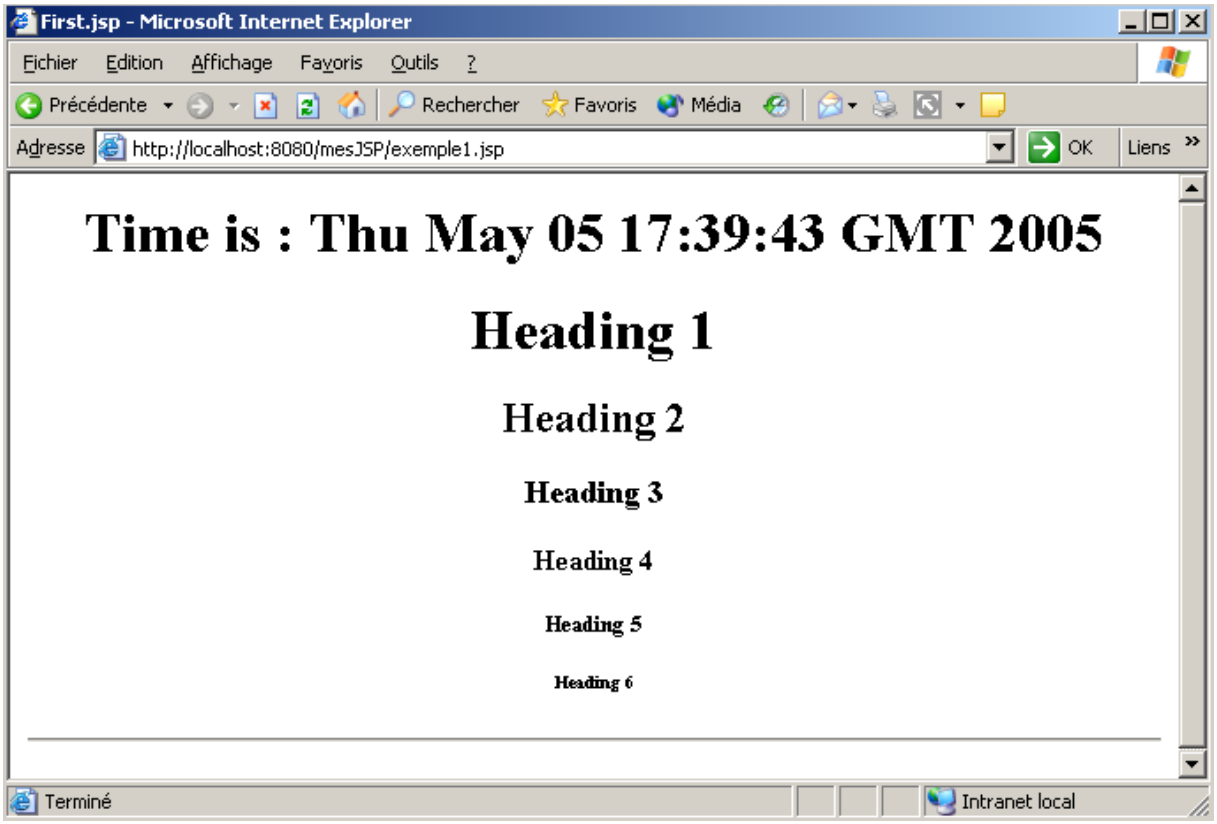


C'est finit ! Vous avez déployé votre JSP. Bien entendu, il est aussi possible de stoker vos JSP dans une archive Web (*.War*) : dans ce cas, le déploiement de l'application dans son intégralité sera simplifiée. Si vous avez plusieurs applications Web différentes, alors vos pages JSP seront stockées

dans les sous-répertoires associés à chaque application Web, bien entendu. Dans tous les cas, vérifiez, bien entendu, que votre serveur soit bien démarré.

Pour tester votre page JSP demandez l'URL suivante à votre navigateur : http://localhost:8080/mesJSP/exemple1.jsp

Normalement, vous devriez voir le résultat suivant associé au code HTML ci-contre.



Code source coté client:

```
<html>
    <head>
        <title>First.jsp</title>
    </head>
    <body>
        <h1 Align="center">Time is : Mon Aug 25 19:17:11 CEST 2003</h1>
            <h1 align="center">Heading 1 </h1>
            <h2 align="center">Heading 2 </h2>
            <h3 align="center">Heading 3 </h3>
            <h4 align="center">Heading 4 </h4>
            <h5 align="center">Heading 5 </h5>
            <h6 align="center">Heading 6 </h6>
        <hr>
    </body>
</html>
```

Nous sommes bien d'accord : au final, le serveur d'applications n'a renvoyé uniquement que du code HTML

III.3- Les différentes constructions syntaxiques d'une JSP

Une page JSP peut être formée par les éléments suivants :

- Les expressions
- Les déclarations
- Les directives
- Les scriptlets
- Les actions

Les expressions

Les expressions JSP sont, comme leur nom l'indique, des expressions Java qui vont être évaluées à l'intérieur d'un appel de méthode *print* (voir l'exemple précédent). Une expression commence par les caractères *<%=* et se termine par les caractères *%>*. Comme l'expression est placée dans un appel de méthode, il est interdit de terminer l'expression via un point-virgule. Sans quoi une erreur de compilation vous sera retournée lors de la première invocation de votre JSP. Revoici un petit exemple d'expression JSP.

```
Time is : <%= new Date() %>
```

Les déclarations

Dans certains cas, un peu complexe, il est nécessaire d'ajouter des méthodes et des attributs à la servlet qui va être générée (en dehors de la méthode de service). Une construction JSP particulière permet de répondre à ces besoins. Elle commence par les caractères *<%!* et se termine, vous vous en doutez, par les caractères *%>*. Voici un petit exemple d'utilisation.

```
<%@ page language="java" %>
<HTML>
    <HEAD>
        <TITLE>Exemple d'utilisation de déclarations JSP</TITLE>
        <%! private int userCounter = 0; %>
    </HEAD>
    <BODY>
        <H1 align="center">Exemple d'utilisation de déclarations JSP</H1>

     <P>Vous êtes le <%= ++userCounter %><SUP>ième</SUP> client du site</P>
    </BODY>
<HTML>
```

Les directives

Une directive permet de spécifier des informations qui vont servir à configurer et à influer sur le code de la servlet générée. Ce type de construction se repère facilement étant donné qu'une directive commence par les trois caractères *<%@*. Notons principalement deux directives : *<%@ page ... %>* et *<%@ include ... %>*. Voyons de plus près quelques unes des possibilités qui vous sont offertes.

La directive <%@ page .. %> permet donc, notamment, de pouvoir spécifier des informations utiles pour la génération et la compilation de la servlet. En fait, cette directive accepte de nombreux paramètres. Le tableau suivant vous présente sommairement, les principaux paramètres.

language="java"	Permet d'indiquer quel est le langage utilisé pour la génération de page. Dans le cas de page JSP, c'est Java qui est tous naturellement mentionné.
import="package|class"	Permet d'injecter dans la servlet générée, des inclusions de package. ou de classes. Vous pouvez indiquez plusieurs localisation en utilisant la virgule comme séparateur. <%@page include="java.util.*" %> <%@page include="java.util.Date" %> <%@page include="java.util.*,java.io.*" %>
isThreadSafe="true|false"	Permet d'indiquer si le modèle *SingleThreadModel* est utilisé ou non. Rappelez-vous (nous en avons parlé dans les chapitres précédents) que cela impacte sur le nombre d'instances de la classe de servlets que va utiliser le conteneur J2EE.
session="true|false"	Cet attribut permet de spécifier si un objet de session peut être maintenu ou non pour cette page JSP.
extends="class"	Vous pouvez ainsi indiquer qu'elle est la classe mère de la servlet générée. Cela peut être extrêmement utile pour séparer le code de la partie HTML. Il suffit juste de créer une servlet qui contienne le comportement souhaité, puis d'en faire dériver votre JSP qui elle utilisera ce comportement, afin de générer le flux HTML. Cela peut aussi être pratique si plusieurs pages JSP cherchent à partager un même comportement.
errorPage="url"	En cas de problème, une exception sera normalement remontée à la classe mère de votre JSP, sauf si vous la capturez dans votre code. Si vous ne traitez pas cette éventuelle exception, la classe mère est codée de manière à intercepter cette dernière. Dans ce gestionnaire d'exception, il y a un test sur l'existence de cet attribut *errorPage*. S'il est fournit, la servlet invoquera l'url mentionnée pour le retour HTML qui informera le client d'une erreur, mais de manière plus adaptée.

isErrorPage="true|false"	Cet attribut permet d'indiquer au conteneur J2EE, que la servlet générée sert de page d'erreur. Si cet attribut est fixé à *true*, c'est certainement que d'autres pages auront l'attribut *errorPage* pointant sur la servlet considérée.
...	

La directive *<%@ include ... %>* est très utile si plusieurs pages se doivent de partager une même ensemble d'information. C'est souvent le cas avec les entêtes et les pieds de pages. Dans ce cas, codez ces parties dans des fichiers séparés et injectez les, via cette directive, dans tous les autre fichiers qui en ont besoin. Voici un petit exemple d'utilisation de cette directive.

```
<%@ page language="java" %>
<HTML>
    <HEAD>
        <TITLE>Exemple d'inclusion de fichier</TITLE>
    </HEAD>
    <BODY>
        <H1 align="center">Exemple d'inclusion de fichier</H1>

        <%@ include file="header.jsp" %>
        <!-- Contenu de la page à générer -->

        <%@ include file="footer.jsp" %>
    </BODY>
<HTML>
```

Les scriptlets

Les scriptlets correspondent aux blocs de code introduit par le caractères *<%* et se terminant par *%>*. Ils servent à ajouter du code dans la méthode de service. N'oubliez surtout pas que tous le code HTML d'une servlet équivaut à coder un gros scriptlet.

Le code Java du scriptlet est inséré tel quel dans la servlet générée : la vérification, par le compilateur, du code aura lieu au moment de la compilation totale de la servlet équivalent. L'exemple complet de JSP présenté précédemment, comportait quelques scriptlets :

```
<%
for(int i=1; i<=6; i++) {
out.println("<H" + i + " align=\"center\">Heading " +
                        i + "</H" + i + ">");
            }
%>
```

Les actions

Les actions constituent une autre façon de générer du code Java à partir d'une page JSP. Les actions se reconnaissent facilement, syntaxiquement parlant : il s'agit de tag XML ressemblant à <jsp:tagName ... />. Cela permet d'indiquer que le tag fait partie du namespace (espace de noms) *jsp*. Le nom du tag est près établi. Enfin, le tag peut, bien entendu comporter plusieurs attributs.

Il existe plusieurs actions différentes. Les principales sont reprises dans le tableau qui suit.

<jsp:include>	Cette action permet d'injecter dans la JSP une autre JSP, une servlet ou une page HTML. Nous avons déjà vue précédemment une autre façon de faire avec la directive *<%@ include %>*. A vous de choisir une technique ou une autre. <jsp:include page="footer.html" />
<jsp:forward>	Permet de rediriger le traitement sur une autre page. On peut aussi obtenir ce comportement, via la méthode *sendRedirect* sur l'objet *response*. <jsp:forward page="url" />
<jsp:useBean>	Cette méthode permet d'instancier un objet Java (un JavaBean), en indiquant sa visibilité (la page, la session, ...). Cet objet va être nommé, ce qui permettra aux actions *<jsp:setProperty ...>* et *<jsp:getProperty ...>* de le manipuler. Ces actions permettent de réduire le nombre de lignes de code dans la JSP au profit d'un modèle déclaratoire. <jsp:useBean id="jbName" class="TheClass" scope="session" />
<jsp:setProperty>	Cette action, permet de modifier une propriété sur un objet créé via l'action *<jsp:useBean ...>*. En fait cette action va générer un appel à une méthode setXXX (ou XXX est le nom de la propriété) sur le composant JavaBean. <jsp:setProperty name="jbName" property="XXX" value="<%= javaExpression %>" />
<jsp:getProperty>	Enfin, cette action est l'inverse de la précédente : elle permet de retourner dans le flux HTML, la valeur de la propriété considérée. <jsp:getProperty name="jbName" property="XXX" />

Exemples :

• Inclusions dynamiques avec

```
<jsp:include page="url" flush="true" />
```

Soit la page bannière.jsp suivante :

```
<p><b>
Titre : <%= request.getParameter("titre") %>
</b></p>
```

et son utilisation:

```
<html><body>
<h1>Essai de jsp:include</h1>
```

```
<jsp:include page="banniere.jsp" flush="true">
  <jsp:param name="titre" value="Premier titre"/>
</jsp:include>

<jsp:include page="banniere.jsp" flush="true">
  <jsp:param name="titre" value="Deuxième titre"/>
</jsp:include>

</body></html>
```

• Redirection

```
<jsp:forward page="relative_url" />
```

ou bien

```
<jsp:forward page="banniere.jsp">
  <jsp:param name="titre" value="Premier titre"/>
</jsp:forward>

<p>paragraphe ignoré !</p>
```

- **Afficher des JavaBeans**

Soit le JavaBean `fr.massat.Produit`:

```
package fr.massat;

public final class Produit {
    String nom = null;   String prix = null;
    String desc = null;

    public String getNom()  { return nom; }
    public void setNom(String s) { nom = s; }
}
```

La page suivante réalise son affichage (le `scope` peut être `page`, `request` ou `application`):

```
<jsp:useBean
  id="produit"
  scope="session"
  class="fr.massat.Produit"
  />

<p>Nom: <%= produit.getNom() %></p>
<p>Prix: <%= produit.getPrix() %></p>
<p>Desc: <%= produit.getDesc() %></p>
```

Une autre façon de réaliser l'affichage d'un Bean avec la page « `affiche_produit.jsp` »:

```
<jsp:useBean
  id="produit"
  scope="session"
  class="fr.massat.Produit"
  />
```

```
<p>Nom:
   <jsp:getProperty
      name="produit" property="nom"/> </p>
<p>Prix:
   <jsp:getProperty
      name="produit" property="prix"/> </p>
<p>Desc:
   <jsp:getProperty
      name="produit" property="desc"/> </p>
```

- **Modifier des JavaBeans**

• Il existe trois façons de modifier un JavaBean dans une page JSP:

```
<jsp:setProperty
  nom="produit"
  property="nom"
  value="Voiture" />
```

En récupérant la valeur d'un paramètre:

```
<jsp:setProperty
  nom="produit"
  property="prix"
  param="prix" />
```

En récupérant tous les paramètres:

```
<jsp:setProperty
  nom="produit"
  property="*" />
```

• Un exemple d'affectation des JavaBeans:

```
<jsp:useBean
  id="produit"
  scope="session"
  class="fr.massat.Produit" />

<jsp:setProperty name="produit" property="*"/>

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

Quelques essais

```
affect_produit.jsp
affect_produit.jsp?nom=voiture&prix=200
affect_produit.jsp
affect_produit.jsp?desc=blablabla
```

III.4- JSP ou Servlet ?

Pour clore ce chapitre, que pensez-vous être le plus utile : les servlets Java ou bien les pages JSP ? C'est à cette question que nous allons tenter de répondre.

En fait, il n'y a pas une technologie qui est, de manière absolue, meilleur qu'une autre. Elles ont, toutes les deux des points forts et des inconvénients. Dans les deux cas, l'inconvénient majeur est qu'on mixe deux types de code : du code HTML de description de la page, et du code Java implémentant le comportement souhaité. Or, on peut au moins dire qu'il y a deux types d'informaticiens pouvant intervenir sur le développement d'un site Web : les infographistes et les développeurs. Chacun d'eux cherchant au maximum à éviter l'autre type de code. Le mieux est donc de chercher à séparer ces deux types d'informations. Dans ce but, on peut dire qu'il y a deux solutions intéressantes.

Vous pouvez utiliser le modèle MVC (Modèle Vue Contrôleur). Chaque URL invoque une servlet. Les servlets ont pour but de construire les objets JavaBeans chargés de prendre en charge la logique métier. Ensuite, le contrôleur (la servlet) donne la main à la vue (la JSP) qui devra se charger de construire la réponse HTML. Pour ce faire, la JSP utilisera les JavaBeans, par l'intermédiaire des actions JSP, dont nous avons parlé il y a quelques instants.

Souvent, en programmation orientée objets, il y a plusieurs solutions pour arriver à un résultat (avec parfois, certaines meilleurs que les autres). Bien souvent, nous avons ainsi le choix entre l'utilisation de l'agrégation ou de l'héritage. Dans le cas présent, on peut retrouver un certaines analogies. Les modèle MVC, peut être assimilé à de l'agrégation ; c'est la servlet qui connaît sa JSP associée.

L'autre solution, pourrait donc consister à utiliser l'héritage. En effet, nous savons qu'au final, la JSP est transformée en une servlet. Il est donc envisageable de coder une servlet contenant toute la logique métier, ou bien manipulant du code de construction de JavaBeans. Ensuite, il suffit d'indiquer, dans la déclarative *<%@ page* de la JSP, que cette dernière dérive de la classe de servlet que vous avez codé. Ainsi, l'utilisateur du site n'invoque que des JSP, mais par héritage, celle-ci possède du code métier (héritage de la servlet).

Dans les deux cas proposés, la séparation de la logique métier et de la logique de présentation est bien réalisée. Mais notez bien que dans ces deux cas, la communication entre les deux couches impose que les deux couches soient contenues dans une même JVM, ce qui, en terme de maîtrise de la montée en charge du système, peut s'avérer être problématique. C'est pour cela que les composants EJB (Enterprise Java Beans) ont été conçus, mais ceci est un autre sujet.

III.4-Conclusion

Au terme de ce chapitre, nous avons présenté la technologie JSP. Il en résulte qu'une JSP est transformée en une servlet, et déployée, tous cela implicitement par le conteneur de servlet que vous utilisez. Ce qui, d'un certain point de vue, est sympathique comparé aux servlets.

Paradoxalement, alors que les servlets mettent l'accent sur le code Java, les JSP, elles, favorisent le code HTML. De nombreuses constructions permettent malgré tout d'influer précisément sur le code Java auto-généré.

En fait, il ne faut pas dire que les JSP sont mieux que les servlets. Ni même l'inverse. En réalité, SUN Microsystems à développé ces deux technologies dans le but d'être utilisée conjointement.