Apprenez à créer des applications web dynamiques avec JEE

Informations sur le tutoriel

Ajouter à mes tutoriels favoris (62 fans)

Auteur : cysboy
Difficulté :
Visualisations : 111 368 364
Licence :

Plus d'informations

Historique des mises à jour

- *Le 02/01/2010 à 17:20:50*
  Correction orthographique,
- *Le 30/10/2009 à 12:13:28*
  Publication sous licence CC-BY-NC-SA
- *Le 29/10/2009 à 13:11:18*
  Correction orthographique mineure

Bonjour à toutes et tous !
Avant de commencer à lire ce tuto, vous devez avoir lu les premières parties du tuto sur la programmation en Java ainsi que la totalité du tuto de M@teo sur XHTML / CSS.

Ceci fait, nous allons pouvoir nous concentrer sur ce qui nous intéresse ici : JEE.
Pour simplifier nous pouvons aussi dire : **développement d'application WEB en Java**.


Je ne prétends pas faire un cours magistral sur le sujet, mais les Zéros qui connaissent un peu JEE savent que beaucoup de choses rentrent en ligne de compte et, par conséquent, que ça nécessite patience, rigueur et débrouillardise.

Nous allons, dans un premier temps, rappeler quelques notions sur le fonctionnement du web.
Ensuite, nous verrons ce qu'est le coeur d'une application JEE, les servlets.
Après avoir appris à manipuler ces objets, nous verrons que, pour avoir une application respectant certains standards de programmation JEE, nous allons devoir coupler ces dernières avec ce qu'on appelle des JSP.
Toute application web dite dynamique nécessite une base de données ainsi que des objets qui iront manipuler ces données. La suite nous amènera à l'interaction avec les bases de données.

J'espère que ce programme vous plaît...

Petite précision :
En fait, la version actuelle (JEE) est un diminutif de Java 5 Enterprise Edition : Le 5 signifiant en fait 1.5.
Par contre, dans ce tuto, nous partirons de la version 1.4 qui s'appelle, elle, J2EE : le 2 couvre en fait jusqu'à la version 1.4 !

Je vois que vous êtes impatients de commencer, alors allons-y !

Ce cours est composé des parties suivantes :

Partie 1 : Prologue : les bases

Dans cette partie nous poserons les bases de connaissance ainsi que quelques rappels afin de pouvoir développer aisément !
Vous apprendrez peut-être certaines choses sur le fonctionnement du web et, pour ceux qui ne connaissent pas J2EE,

vous verrez comment débuter dans cette aventure !
Oui, ce que vous allez vivre à partir de maintenant sera une véritable aventure .

Bon, nous avons réussi tant bien que mal à poser les bases de la plateforme JEE.

Vous avez appris à créer un projet, faire des servlets, combiner ces dernières avec des JSP tout en utilisant des objets métiers !
Maintenant, vu que vous vous êtes sûrement posé beaucoup de questions sur le fonctionnement de tout ceci, le moment est venu d'apporter quelques éléments de réponses...

- ## Partie 2 : Ce que sait faire le conteneur

Dans la partie précédente, nous avons fait un tour rapide de la plateforme JEE.
Ceci dans le sens où nous avons créé une servlet liée à une JSP tout en utilisant un objet métier.

Par contre, bon nombre de points doivent vous sembler obscurs.

Ce que je vous propose dans cette partie n'est rien d'autre que de faire la lumière sur ce qu'il se passe dans notre conteneur.

 Vous vous sentez prêts ?

Alors go ! 

- 1) Paramètres de servlets

  - Paramètres d'initialisation
  - Tout dépend du contexte...
  - Des objets en paramètres

- 2) Cycle de vie d'une servlet

  - Initialisation de la servlet
  - Utilisation de la servlet
  - Le retour des listeners
  - Des listeners, encore des listeners

Partie encore en cours d'édition...
Revenez y faire un tour.

Partie 1 : Prologue : les bases

Dans cette partie nous poserons les bases de connaissance ainsi que quelques rappels afin de pouvoir développer aisément !
Vous apprendrez peut-être certaines choses sur le fonctionnement du web et, pour ceux qui ne connaissent pas J2EE, vous verrez comment débuter dans cette aventure !
Oui, ce que vous allez vivre à partir de maintenant sera une véritable aventure  .

Rappels

Le titre de ce chapitre n'est pas très évocateur, je vous le concède... 
En fait, vous auriez pu remplacer ce dernier par : "**tout ce que vous avez toujours voulu savoir sur le web sans jamais avoir osé le demander !**".

Je sais qu'en bons Zéros que vous êtes, vous surfez régulièrement sur le net ; mais savez-vous réellement ce qu'il se passe lorsque vous saisissez une URL ou que vous suivez un lien ? Savez-vous ce qu'est un serveur web ? Connaissez-vous la différence entre une requête POST et une requête GET ?

Non ? Alors suivez le guide !

Internet : qui ? quoi ? qu'est-ce ?

Je ne vais pas vous faire un historique sur la naissance du web tel que nous le connaissons maintenant, je vais juste vous rappeler le fonctionnement de celui-ci.
Cependant, si certaines personnes souhaitent tout de même en savoir plus sur l'histoire d'Internet, elles peuvent suivre ce lien.

Pour faire court, **ne confondez pas Internet avec le web !**

Internet est un assemblage de multiples réseaux, tous connectés entre eux. Cet amas de câbles, de fibres optiques... de matériels, pour faire simple, constitue Internet, aussi appelé "**le réseau des réseaux**".

Le **Web** est un système de fichiers présent sur des machines (serveurs) transitant par un protocole particulier, consultable grâce à des navigateurs web et fonctionnant **SUR** Internet ! Le web est donc un système de fichiers que toute personne possédant un ordinateur (ou un téléphone, maintenant...) connecté à Internet peut consulter (avec un abonnement d'un FAI, bien sûr...).

En fait, consulter les fichiers présents sur le web est chose courante, surtout pour vous !
Eh oui ! Surfer sur le web, aller sur le Site du Zéro, consulter vos mails chez votre FAI... tout ceci est en fait de la consultation de fichiers présents sur Internet.
Vous n'êtes pas sans savoir que, dans la majeure partie des cas, on surfe sur le web avec un navigateur tel que Firefox, Internet Explorer, Safari... Ne vous êtes-vous jamais demandés comment les navigateurs savent aller au bon endroit ? Comme, par exemple, aller sur le SdZ ?

Votre navigateur vous demande une URL saisie en "**1**" et, une fois cette adresse validée, votre navigateur vous renvoie ce qui se trouve à cette adresse (oui, c'est une adresse), le Site du Zéro, en "**2**".
Il faut bien sûr que l'adresse existe et qu'il y ait quelque chose à cette adresse, sinon :

Pourquoi certaines adresses nous renvoient des pages web et d'autres des erreurs ?

Pour ceux qui ne le sauraient pas, tout ordinateur actuel possède une adresse sur un réseau : son adresse IP.
C'est grâce à cette adresse qu'un ordinateur, ou un serveur, peut s'identifier sur un réseau. Voyez ça comme sa carte d'identité. 
Par exemple, chez moi, je suis connecté à ma box (fournie par mon FAI) qui me donne accès à Internet.
Sur Internet, cette box a une adresse qui lui est propre et celle-ci ressemble à quelques choses comme ça **242.231.15.123** : on appelle ces adresses des "adresses IP".

Lorsque vous demandez une page web à votre navigateur, vous lui demandez, de façon tacite, d'aller chercher ce qui se trouve à l'adresse demandée !

 Eh ! Si les ordinateurs ont des adresses pour se reconnaître sur les réseaux, comment se fait-il qu'en tapant un nom comme "**google.com**" les navigateurs sachent où chercher ?

Partez du principe que toute adresse de site internet pointe vers un serveur (ou plusieurs) qui a une adresse. Par exemple, taper "http://www.google.fr" dans votre navigateur revient à saisir "**http://74.125.19.147**" (adresse d'un serveur Google sur Internet) : essayez, vous verrez !
Vous êtes d'accord sur le fait que cette suite de nombres n'est pas des plus faciles à retenir...
Il est bien plus simple de mémoriser ***google.fr***. 
Je ne m'éterniserai pas sur le sujet mais sachez qu'il y a une machine qui fait le lien entre les adresses de serveurs (suite de nombres) et les adresses littérales (google.fr) : les DNS. Voyez ces machines comme de gigantesques annuaires téléphoniques, mais pour les sites internet. 

 Et qu'est-ce que c'est que le "**http://**" ?

Si vous relisez bien ce que j'ai dit plus haut, vous devez voir que nous avons vu qu'avec l'URL que vous renseignez, vous spécifiez une machine à interroger, donc des fichiers à lire, il ne nous manque plus que le protocole.
Ici, il s'agit du protocole http.
C'est grâce à ce protocole que le navigateur envoie des "***requêtes***" (nous y reviendrons) aux serveurs que vous sollicitez. Il en existe d'autres comme le FTP, le SMTP...
Inutile de nous apesantir sur le sujet (c'est un tuto de programmation, pas de réseau, non mais)...

Au final, une URL peut se décomposer comme suit :

Je pense que ce schéma est assez explicite... 
Il y a toutefois un petit détail qu'il serait bon que vous sachiez. Dans les URL, il y a un paramètre facultatif : le numéro de port utilisé par le protocole.
En fait, chaque protocole de transfert utilise un port sur le serveur, voyez ça un peu comme une porte affectée à une personne. Par exemple, lorsque vous rentrez dans une gendarmerie, vous prenez l'entrée principale (sauf si vous vous êtes fait coffrer...), seules les personnes autorisées ont le droit de prendre l'entrée de service.

C'est la même chose pour les protocoles de transfert, chacun a un port attribué :

- **HTTP** : port 80 ;
- **FTP** : port 20 ou 21 ;
- **SMTP** : port 25 ;
- ...

Si nous ajoutons le numéro de port à notre URL présente dans le schéma, nous aurions ceci :

http://www.monsite.com:80/dossier/fichier.html

Mais ceci est facultatif puisque le protocole http utilise le port 80 par défaut !

Maintenant que la lumière est plus ou moins faite sur la façon dont le web fonctionne, nous allons voir comment les serveurs nous retournent des pages web.

Les en-têtes en fête !

Nous avons vu que, lorsque vous saisissez une URL dans votre navigateur, que vous validez cette dernière, votre navigateur envoie une "**requête**" au serveur concerné afin qu'il nous renvoie une page web.
Tout d'abord, on nomme vulgairement l'échange de données entre votre navigateur et le serveur qui fournit les pages web un échange **client / serveur**.
Le client représente votre navigateur et le serveur... Enfin, vous avez deviné.

Le moment est venu de vous en apprendre un peu plus.
Voici ce qu'il se passe :

- le client émet une **requête http** vers le serveur ciblé ;
- le serveur reçoit les éléments de la requête ;
- celui-ci les interprète ;
- il renvoie la page demandée en émettant une réponse http ;
- le client reçoit la page au format HTML ;
- le client affiche la page.

Nous pouvons résumer ce qu'il se passe avec ce schéma :

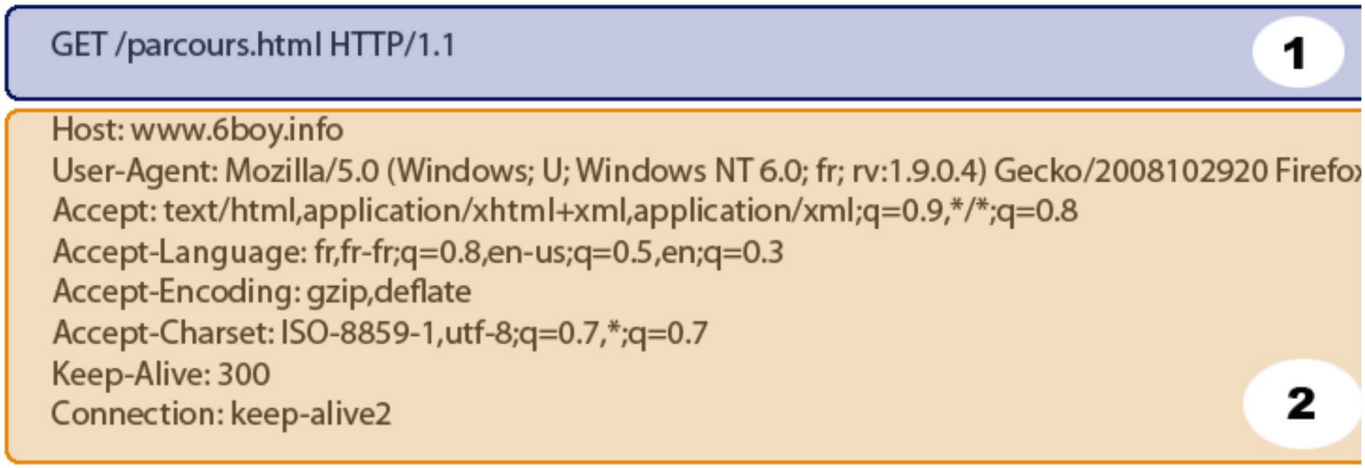
Bon, on a compris que le navigateur et le serveur s'échangent des "requêtes http". Mais c'est quoi ?

Voici un exemple d'en-tête HTTP correspondant à cette adresse : http://www.6boy.info/parcours.html.

Code : Autre

```
GET /parcours.html HTTP/1.1
Host: www.6boy.info
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.0; fr; rv:1.9.0.4) Gecko/2008102920 Firefo
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: fr,fr-fr;q=0.8,en-us;q=0.5,en;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
```

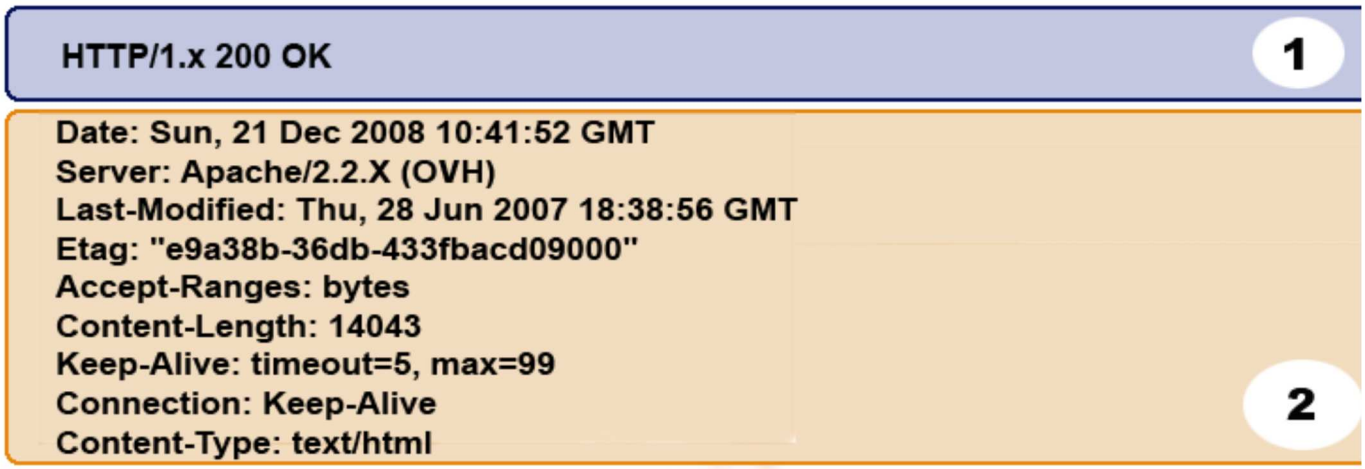
Je me doute que ceci doit vous faire peur...
Ne vous en faites pas trop, nous n'allons pas décortiquer tout ça : nous allons juste voir ce que cela signifie. Je vous ai fait une petite image qui encadre les blocs principaux. Je sais, je suis trop bon...



Dîtes-vous que ce qui correspond au premier point est en fait ce que vous demandez au serveur.
Serveur spécifié dans le paramètre "**Host**" du deuxième bloc (nous ne parlerons pas des autres paramètres du deuxième bloc... le but n'est pas là).
Vous remarquez aussi qu'il y a un type de requête envoyée au serveur ; ici, il s'agit d'une requête de type "**GET**".
Vous pouvez comprendre la première ligne de la requête comme une discussion entre votre navigateur et le serveur spécifié, un peu comme ça : "***Donne-moi la page suivante s'il te plaît***".

Il existe d'autres types de requêtes HTTP, nous y reviendrons le moment venu...

Maintenant, voyons ce que vous répond le serveur :



Le plus intéressant se trouve dans la première zone.

- **HTTP/1.x** : le protocole utilisé pour la réponse
- **200** : ce que le traitement de la requête sur le serveur a retourné comme type de réponse.
- **OK** : la traduction de 200 en langage humain, ici le traitement de la requête a été un succès !

Il y a toutefois une chose importante dans ces entêtes HTTP, aussi bien dans la demande que dans la réponse :

Content-Type: text/html

Ceci signifie que le navigateur demande une page HTML et que le serveur retourne une page HTML.
Vous devriez savoir ce que c'est si vous avez suivi le tuto de M@teo.

Vous n'êtes pas sans savoir que le HTML est le langage de base pour toute page web **STATIQUE**.
Par là entendez que les pages sont toutes identiques et donc que leurs contenus ne s'adaptent pas aux utilisateurs.
Vous n'aimeriez pas vous connecter sur le SdZ et vous retrouver avec un autre compte, ou pire, vous connecter sur votre boîte mail de votre FAI et avoir les mails de votre petite soeur !

Pour réussir à rendre les pages web dynamiques, vous devrez utiliser un langage de programmation, autre que HTML, que votre serveur pourra interpréter !
Sachez qu'un serveur ne sait utiliser et renvoyer que des pages HTML statiques !
Si vous voulez que votre site s'adapte aux utilisateurs, vous aller devoir faire deux choses :

- apprendre un langage permettant de faire ce genre de personnalisation...
- ajouter un serveur d'application capable d'interpréter un langage.

Il y a beaucoup de langages qui permettent de faire ce genre de choses.
Beaucoup de tutos sur ce genre de langage existent sur le SdZ et sur le web en général. Ce genre de langage s'appelle un langage côté serveur : un langage *interprété* par un serveur.
Les plus connus sont PHP, Java, .Net mais il en existe bien d'autres.

Voyons un peu ce qu'il se passe lorsque vous demandez une page contenant ce genre de code !

Soyez dynamiques

Vous allez voir, en fin de compte, c'est très simple...
Je vous ai dit plus haut que, par défaut, votre serveur ne sait manipuler que des fichiers .html, donc des pages web statiques.
Afin que notre serveur sache interpréter du code dans des pages web, nous allons devoir lui installer un "***environnement***".
Pour simplifier les choses, les pages contenant du code à interpréter ont une extension particulière :

- **.php** : page contenant du code PHP ;
- **.aspx** : pages contenant du code .NET ;
- **.jsp** : pages contenant du code Java ;
- ...

Si vous demandez ce genre de page à un serveur sans que celui-ci sache que faire avec ce qui s'y trouve, vous aurez de mauvaises surprises...

Attends deux minutes ! À quoi ressemble le fameux code dont tu parles depuis tout à l'heure ?

En fait, il dépend du langage que vous utilisez...

Voici une page contenant du code PHP :

Code : PHP

```
<html>
 <body>
  <?php echo "Bonjour toi...<br />";?>
 </body>
</html>
```

Voici une page contenant du code Java :

Code : JSP

```
<html>
 <body>
  <% out.println("Bonjour toi...<br />");%>
 </body>
</html>
```

Vous pouvez voir qu'il y a des différences entre ces deux codes, même si le résultat sera le même.

D'accord, on voit déjà mieux à quoi ressemble ton fameux code côté serveur.
Mais, du coup, comment le serveur sait que faire avec ce code ?

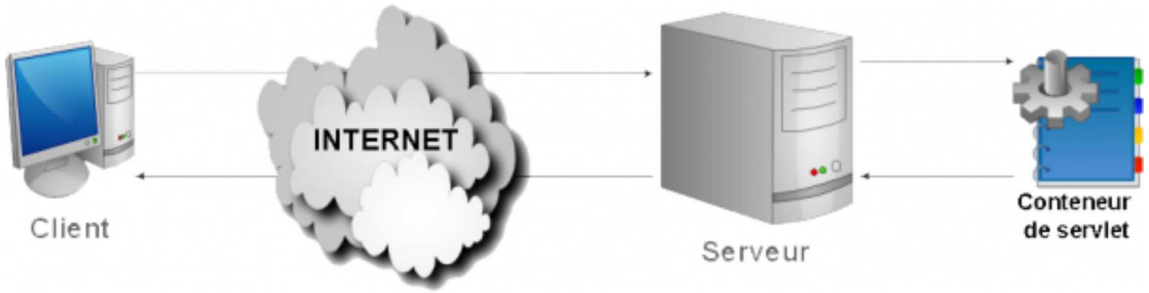
J'allais y venir.
Dites-vous bien que sans le fameux "***environnement***" dont je vous parlais plus haut, le serveur ne saurait pas quoi faire.
En fait, votre serveur va, dès qu'il aperçoit un type d'extension de fichier demandé, laisser l'environnement triturer la page. Ce dernier, après avoir terminé sa cuisine de son côté, retourne une page ne contenant que du code HTML de base au serveur qui, lui, va nous retourner une réponse HTTP avec la page ! 

Nous ne parlerons pas de PHP ou de .NET puisque c'est de Java dont il s'agit dans ce tuto.
Donc, afin que votre serveur sache interpréter du code Java dans les pages web, vous allez avoir besoin d'un logiciel spécifique que l'on nomme vulgairement "**conteneur de servlets**".

Ne vous inquiétez pas, vous allez savoir ce qu'est une servlet très bientôt...

Pour information, sachez qu'il existe plusieurs conteneurs de servlets, chacun avec des caractéristiques différentes.
Nous en parlerons dans le prochain chapitre.

Revenons à nos moutons : je vous disais que le serveur, selon les requêtes émises, demandait de l'aide à notre conteneur.
Pour que vous puissiez mieux visualiser, voici un petit schéma fait par mes soins :



Vous pouvez déjà deviner ce dont nous allons avoir besoin pour travailler.
Nous aborderons tout ceci dans le chapitre prochain ; pour le moment, vu que vous venez de voir dans les grandes lignes comment fonctionne la génération de pages web dynamiques, le temps est venu de faire un petit QCM avant de poursuivre...


Les Zéros qui ne connaissaient pas ces notions ont dû apprendre plein de chose rigolotes ici...
Dîtes-vous bien que ce que nous allons faire, tout au long de ce tuto, n'est qu'une autre façon de créer des programmes pour le

web !
Sur le SdZ, vous pouvez déjà voir que M@teo a fait deux tutos sur la création de sites.

Bon, je me doute que si vous avez lu ce chapitre, c'est que vous êtes intéressés par la façon de créer des sites internet avec Java. Alors continuons...

Votre boîte à outils

Nous avons vu précédemment comment les pages web transitent sur Internet ainsi que la nécessité d'avoir un conteneur sur notre serveur, ceci pour que ce dernier sache interpréter les pages dynamiques.

Dans ce chapitre nous allons mettre en place notre environnement de développement :

- le logiciel que nous allons utiliser pour coder nos pages web ;
- l'environnement d'exécution de ces pages.


Je vous rappelle que, pour suivre aisément ce tuto, vous devez avoir lu les premières partie du tuto Java (au moins les deux premières parties).

Par conséquent, je pars du principe que vous avez un environnement Java installé sur votre machine.
Bien ! Nous allons maintenant mettre en place tout ce dont nous allons avoir besoin pour travailler.

Environnement d'exécution


Attends ! Il va falloir qu'on achète un serveur pour suivre le tuto ?

Non, heureusement !
Nous allons juste utiliser le même logiciel que les serveurs web... Et le serveur sera... votre machine !
Oui, vous avez bien entendu !
Nous allons faire croire à votre ordinateur qu'il est un serveur. Du coup, pas besoin d'héberger nos pages sur Internet pour faire nos tests...

Bon, si vous vous rappelez bien ce que je vous ai dit, il existe plusieurs logiciels qui permettent de contenir des pages web dynamiques écrites en Java.
Vous pourrez trouver :

- Apache Tomcat ;
- JBoss ;
- WebSphere ;
- GlassFish ;
- ...

La liste est longue. Chacun de ces serveurs d'applications permet à une machine de comprendre des pages web contenant du code Java.

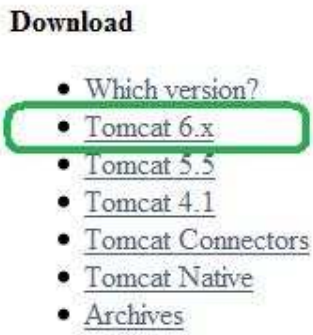

Pourquoi tu les appelles serveurs d'applications ?

Parce que c'est le nom que les développeurs leur donnent. Le serveur physique (la machine) est communément appelée serveur web et l'application qui permet de générer des pages HTML depuis des pages dynamiques s'appelle le serveur d'applications !

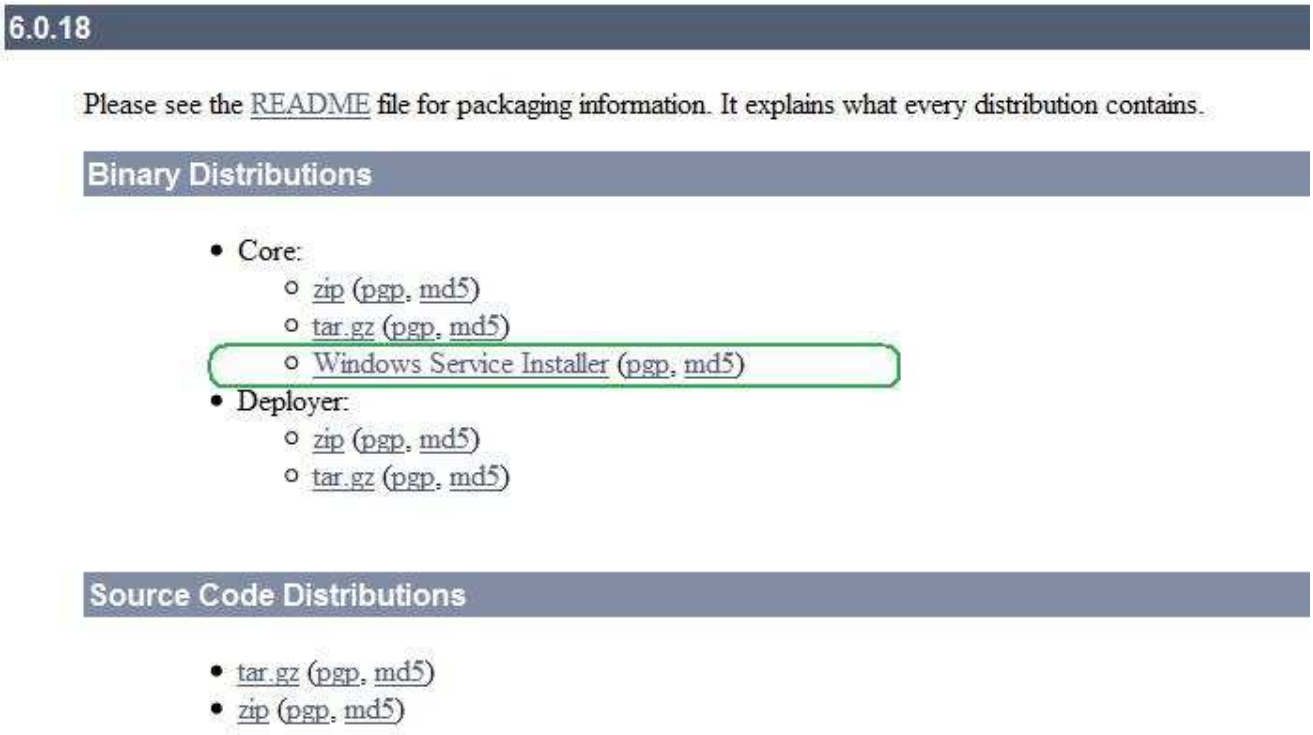
Tous ces serveurs d'applications ont des spécifications, tous sont des conteneurs de servlets (bientôt, bientôt...) mais seulement certains d'entre eux peuvent contenir ce qu'on appelle des EJB. Nous aurons l'occasion d'en parler, mais pour l'heure, sachez seulement que ce sont des objets Java qui permettent de gérer ce qu'on nomme **la persistance des données !**
Je ne vais pas vous en parler maintenant alors que vous n'avez même pas encore installé votre environnement de travail...

Pour l'instant, nous allons utiliser Tomcat.

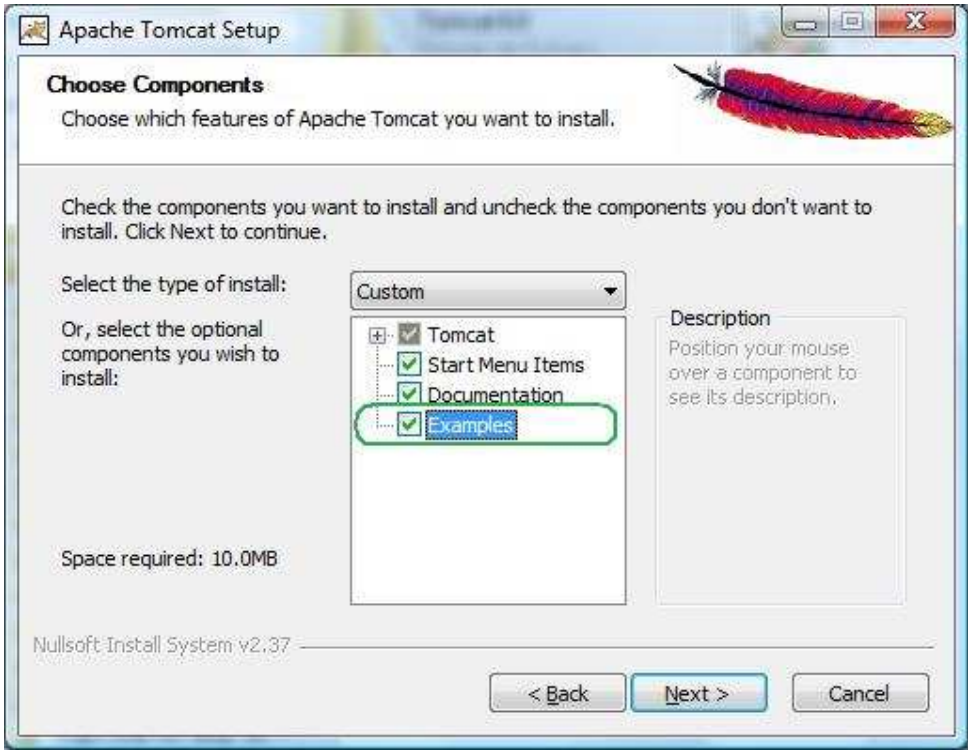
Je vous invite donc à le télécharger sur ce site.
Au moment ou j'écris ce tutoriel, Tomcat en est à sa version 6, je vous invite donc à le télécharger :



Il ne vous reste plus qu'à choisir ce que vous voulez comme type d'installation ; vu que je suis sous Windows, je prends celle-ci :

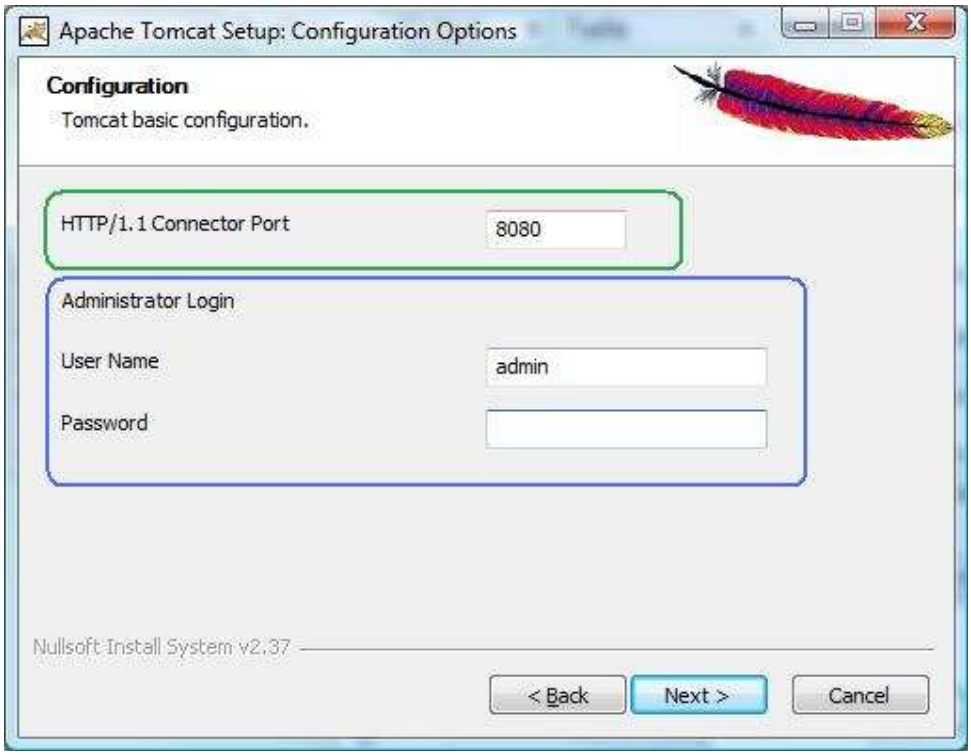


Une fois que le fichier est téléchargé, il ne vous reste plus qu'à l'installer.
Double cliquez sur l'exécutable et laissez-vous guider.
Je vous conseille vivement d'installer les exemples : comme ça, vous pourrez jeter un coup d'oeil... 

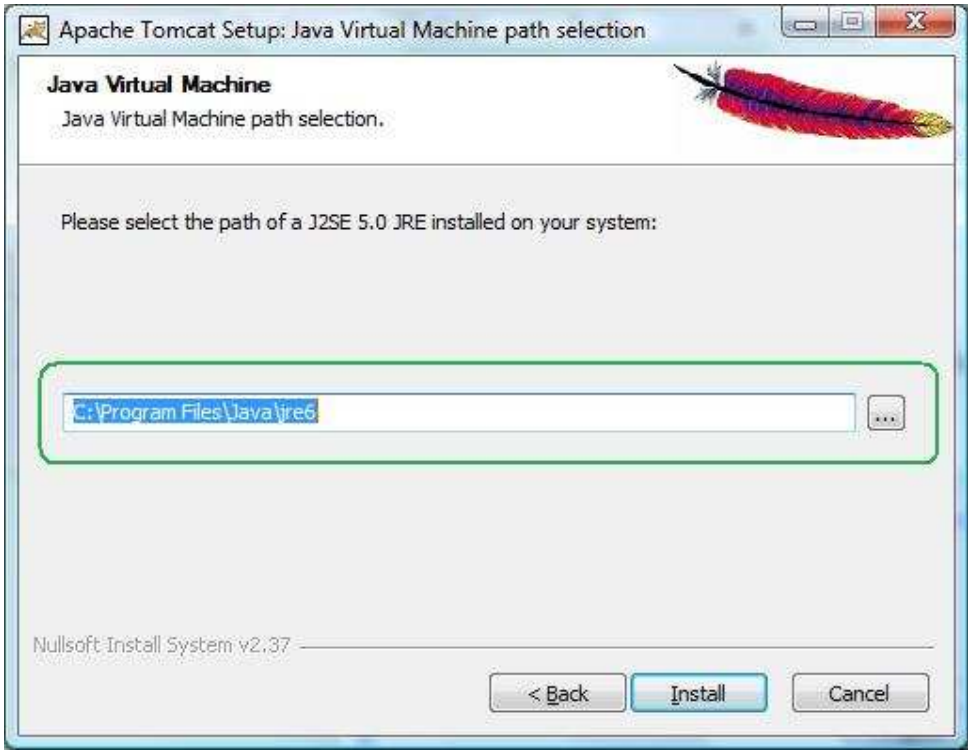


À un moment donné de l'installation, Tomcat va vous demander sur quel port vous voulez utiliser le serveur, le mieux est encore de laisser la valeur par défaut.
Dans la même fenêtre, on vous demande de saisir un nom et un mot de passe pour accéder à l'espace d'administration du serveur. Pour ceci, je vous laisse juges. Je n'ai pas de données sensibles sur mon serveur perso, donc je laisse tel quel... Mais si un jour vous devez utiliser un serveur destiné à être connecté à Internet, je vous conseille vivement de mettre un bon couple login-mot de passe !

Voici ladite fenêtre :



Enfin, avant de terminer l'installation, on vous demandera de spécifier l'endroit où se trouve le JRE installé :



Voilà ; une fois installé, on vous demande si vous souhaitez lancer le serveur : allez-y. Nous allons un peu en faire le tour...
Une fois que Tomcat a lancé ses services, vous devriez avoir cette icône dans la barre des tâches :

Si vous faites un clic droit sur cette icône, vous devriez voir ceci :

Nous ne nous attarderons pas sur l'aspect configuration pour le moment, sachez que vous pouvez arrêter Tomcat en cliquant sur "**Stop service**" ou carrément quitter l'application en cliquant sur "**Exit**".

Génial, mais comment on se sert de Tomcat ?

C'est exactement ce que nous allons voir...

Tomcat pour les intimes !

Bon vous avez installé Tomcat, mais comment s'en sert-on ?
Rien de plus facile !
Rappelez-vous que c'est une application qui émule un serveur sur votre machine. Il suffit donc d'aller l'interroger pour voir ce qu'il y a dedans...

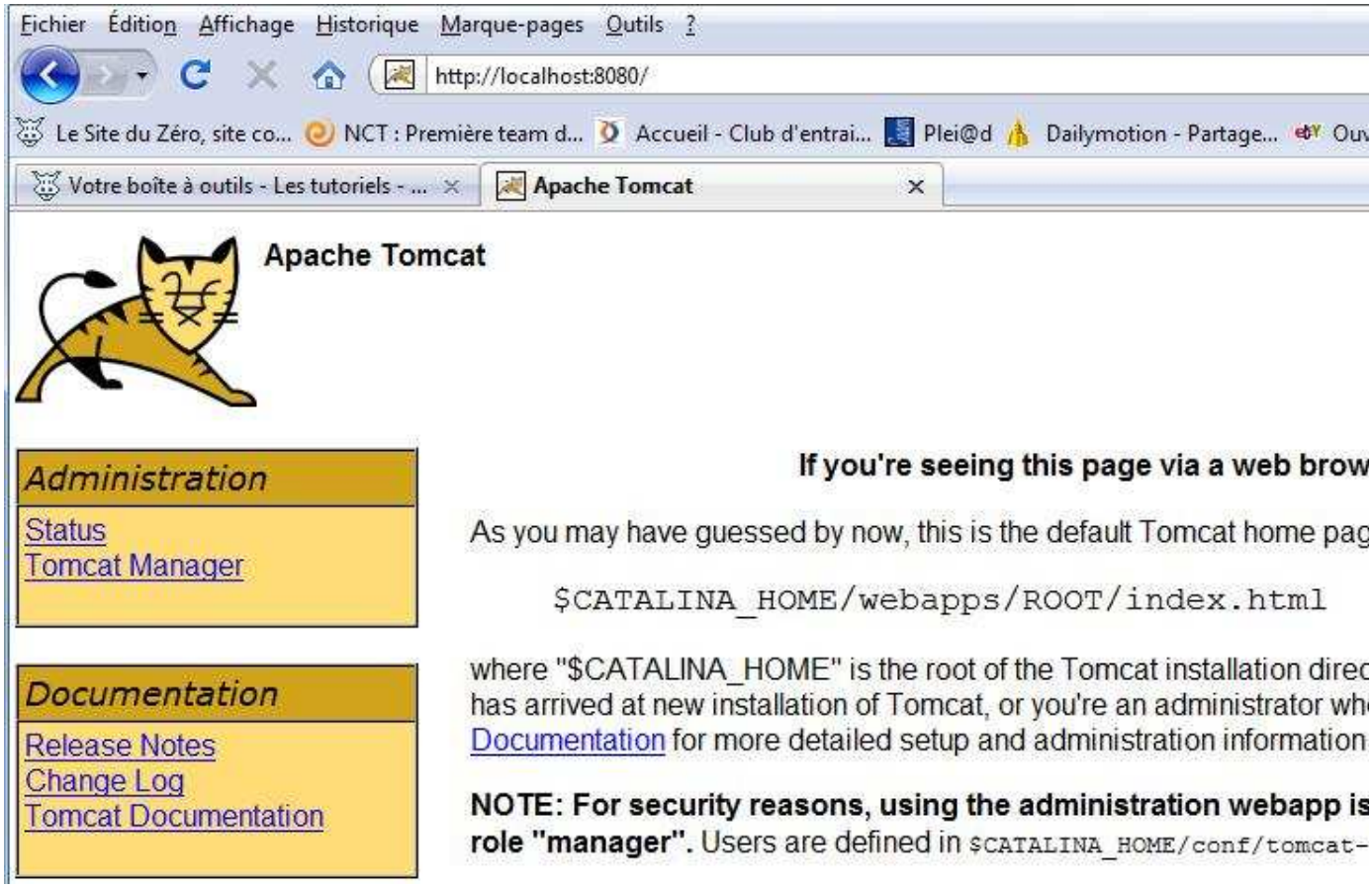
Oui, mais COMMENT ON FAIT ? ? ? ?

Souvenez-vous que pour interroger un serveur, il faut un navigateur et son adresse, l'adresse de notre serveur est "**localhost**", traduisez par "***hôte local***".
Donc si vous tapez ceci dans votre navigateur à l'emplacement de l'URL, vous aurez :

Hein ?

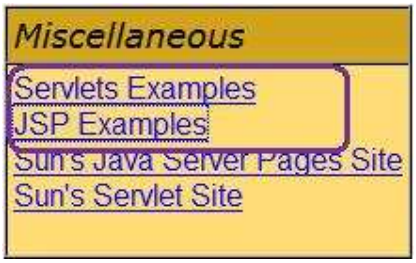
Ah oui... Vous vous rappelez que, pour le protocole HTTP on se sert par défaut du port 80. Cependant, ici, nous avons spécifié à Tomcat qu'il allait utiliser le port 8080 ; il fallait donc taper ceci dans l'URL : "**localhost:8080**".

Voici ce que vous devriez obtenir :



Vous voilà sur la page d'accueil de votre serveur Tomcat ! 

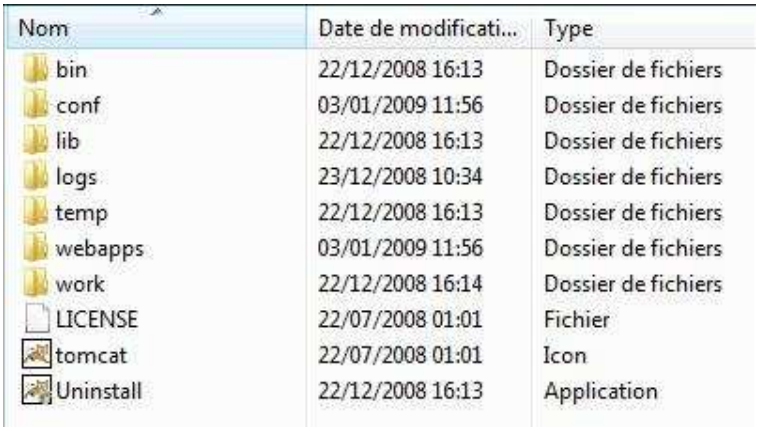
Je vous invite à faire un tour du côté des exemples que vous avez dû installer... Pour ceux qui seraient un peu perdus, c'est par là :



Vous pouvez voir les exemples ainsi que les codes sources générant ces derniers !

 Mais où se trouve tout ça ?

Dans le répertoire d'installation de Tomcat, vous avez plusieurs dossiers :



Nom	Date de modificati...	Type
bin	22/12/2008 16:13	Dossier de fichiers
conf	03/01/2009 11:56	Dossier de fichiers
lib	22/12/2008 16:13	Dossier de fichiers
logs	23/12/2008 10:34	Dossier de fichiers
temp	22/12/2008 16:13	Dossier de fichiers
webapps	03/01/2009 11:56	Dossier de fichiers
work	22/12/2008 16:14	Dossier de fichiers
LICENSE	22/07/2008 01:01	Fichier
tomcat	22/07/2008 01:01	Icon
Uninstall	22/12/2008 16:13	Application

Les exemples se trouvent dans le dossier "**examples**" qui se trouve dans le dossier "**webapps**" !

Vous le découvrirez assez tôt, mais c'est dans le dossier "**webapps**" que nous allons mettre tous nos projets JEE !

Vous pouvez voir aussi qu'il y a des dossiers comme "**conf**" ou "**logs** " dans lesquels vous trouverez respectivement les fichiers de configuration de votre serveur Tomcat, y compris les couples "utilisateurs / mots de passe" dans "**conf/tomcat-users.xml**" (bon à savoir si vous perdez votre mot de passe d'accès à l'administration) et les fichiers que Tomcat génère en cas d'erreur...

Nous aurons l'occasion de reparler de tout ceci.

Nous ne tarderons pas à utiliser l'interface d'administration, mais pour le moment, il nous manque encore quelque chose pour réaliser des applications web : **un éditeur de code !**

Eclipse : le retour

Afin de pouvoir créer des applications JEE, il va nous falloir l'environnement de développement JEE, téléchargeable ici.
Il s'agit de la version d'Eclipse permettant de réaliser des projets JEE.

Il vous suffit de suivre le lien entouré ci-dessous :

Si la version J2SE d'Eclipse ne vous permet pas d'effectuer d'application JEE, le contraire est vrai ! Avec la version JEE d'Eclipse, vous pourrez très bien faire des applications Java comme vu dans le tuto : programmation en Java.

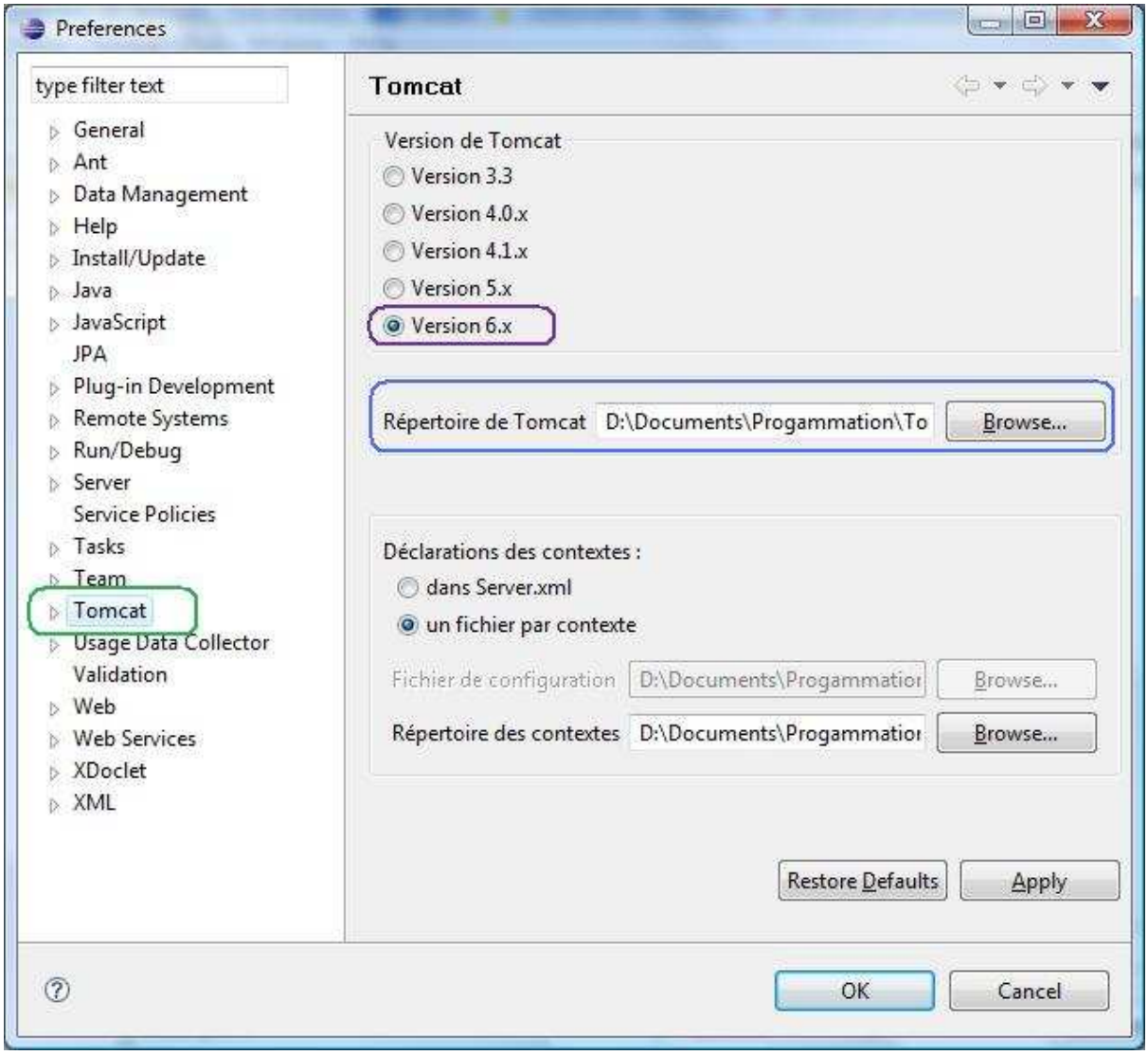
Une fois Eclipse téléchargé, nous avons ce qu'il nous faut pour créer des applications web en Java.
Toutefois, le fait que nous devions lancer Tomcat, puis lancer Eclipse afin de tout faire fonctionner semble un peu fastidieux.

Il existe un petit plug-in bien utile pour Eclipse, permettant à ce dernier de piloter le lancement, l'arrêt ou le redémarrage de Tomcat.
Je vous invite donc à le télécharger ici.
Choisissez la version la plus récente :

Version	File	Date	Comment
3.2.1	tomcatPluginV321.zip	10 May 2007	releaseNotesV321.txt Works with Eclipse 3.1, 3.2 and 3.3M7 Fix a problem with HTTPS

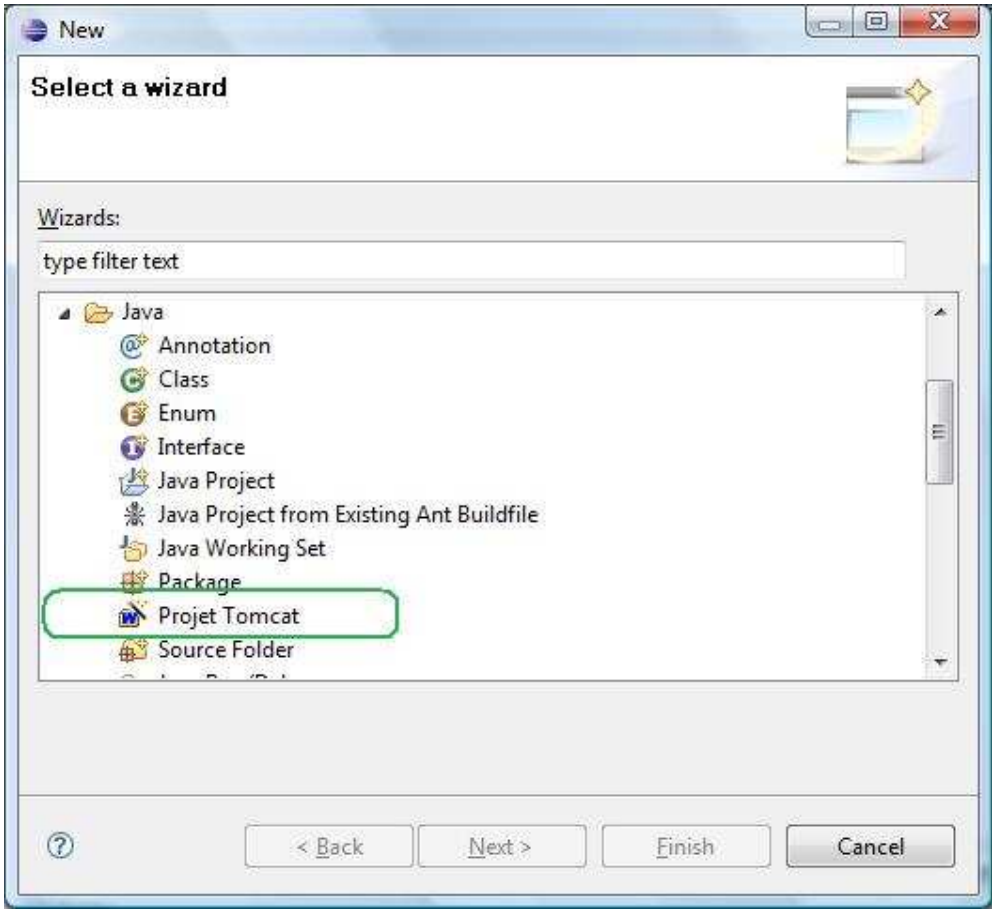
Une fois le plug-in téléchargé, il vous suffit de dézipper l'archive et de coller le dossier que celle-ci contenait dans le dossier "**plugin**" présent dans le dossier d'installation d'Eclipse !
Une fois ceci fait, vous pouvez lancer Eclipse. Vous devriez voir trois boutons avec une icône familière :

Il ne nous reste plus qu'à paramétrer Tomcat dans Eclipse et le tour sera joué !
Pour cela, vous devez aller dans "**Window**", choisissez le point de menu "**Preferences**" ; de là, allez dans les paramétrages de Tomcat.
Il ne vous reste plus qu'à spécifier quelle version de Tomcat vous utilisez et où vous l'avez installé :



Voilà ! Vous pouvez désormais piloter Tomcat depuis Eclipse : vous verrez très bientôt que ceci est très utile.
Je pense que vous avez compris que le bouton de gauche lance Tomcat, celui à sa droite l'arrête et le dernier relance le serveur... Rien de compliqué ! 

Il y a une autre nouveauté qu'apporte ce plug-in : la possibilité de créer un projet Tomcat.



Enfin, nous avons tout ce dont nous avons besoin pour travailler ! Nous pouvons commencer et ça ne sera pas une partie de plaisir... Préparez-vous...
Nous allons voir tout ceci dans le prochain chapitre avec votre première application web en Java !

Voilà. Les bases sont posées !
Vous avez tout ce qu'il faut pour vous lancer dans l'aventure J2EE. Je vous propose donc de commencer tout de suite avec un chapitre somme toute assez touffu...

Premiers pas

Maintenant que nous avons nos outils, nous allons pouvoir rentrer dans le vif du sujet !

Rappelez-vous comment fonctionnent les applications web : requête, traitement, réponse.
Nous allons voir comment bien commencer dans l'aventure JEE, mais progressivement car, avant de commencer à programmer, vous allez avoir besoin de précisions sur la façon dont Tomcat va gérer nos applications !

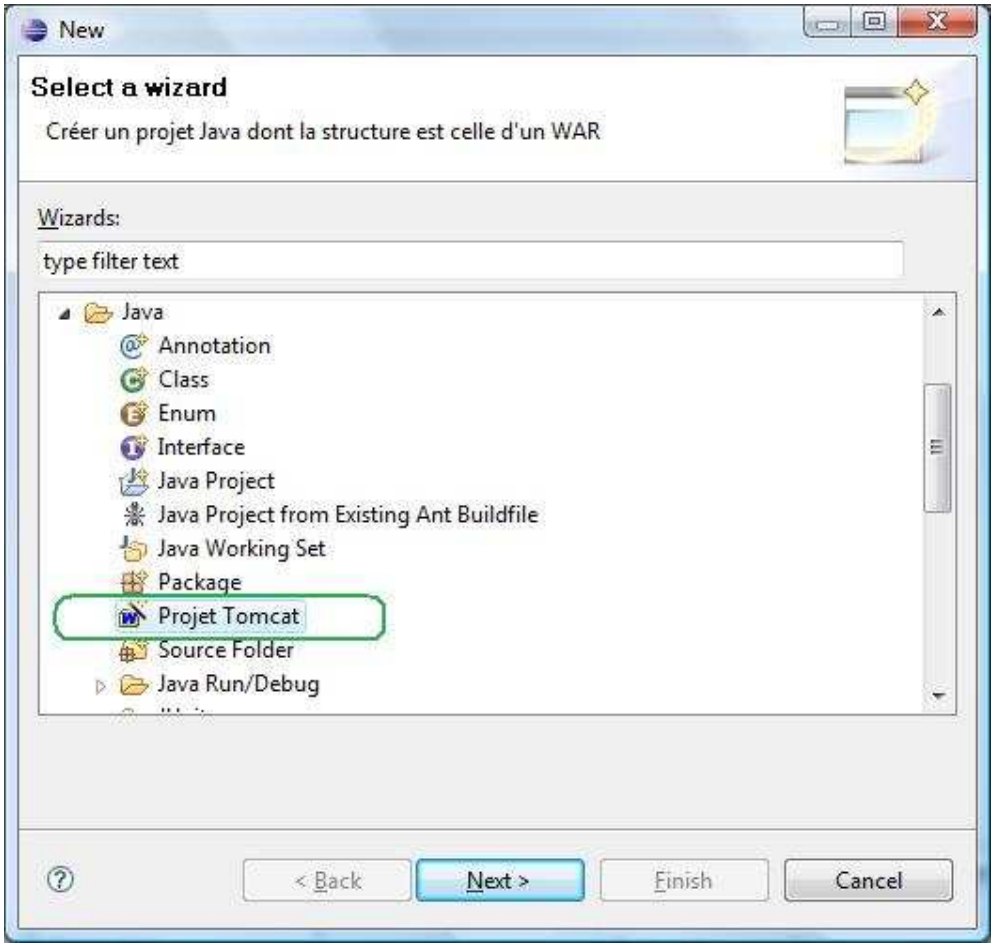
Bon ben, qu'est-ce qu'on attend ?

Création

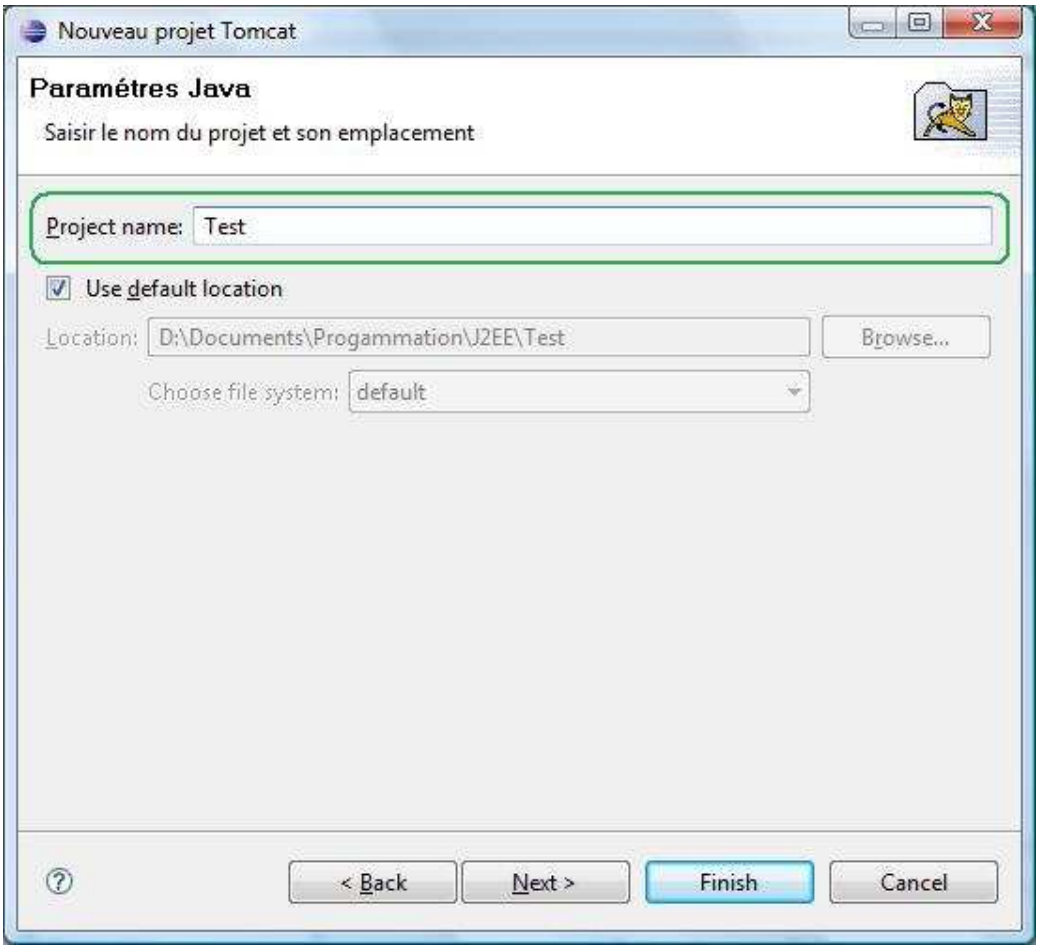
Nous allons faire simple (comme d'habitude...).
Avant de commencer à coder, nous allons tout d'abord voir comment créer un projet JEE - ceci n'est pas anodin - et ensuite, comment mettre notre application sur notre serveur (dans Tomcat !), tout ça pour pouvoir admirer notre travail.

Lors du dernier chapitre, je vous avais montré que le plug-in Tomcat permettait de créer un "**projet Tomcat**" : c'est exactement ce que nous allons faire !

Faîtes "**File > new > Other**", déroulez le menu "**Java**" et choisissez un "**Projet Tomcat**" :

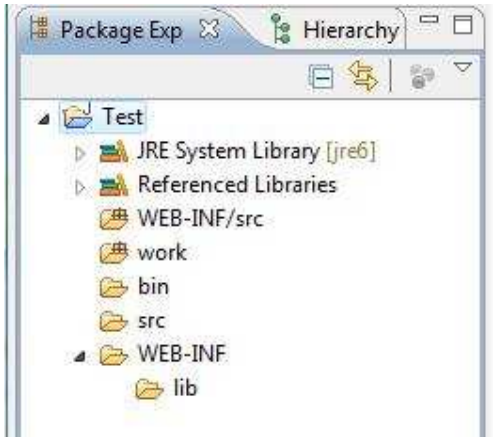


Nommez-le "**Test**" :

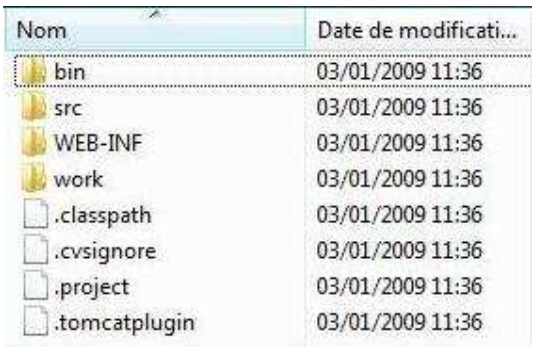


Cliquez sur "**Finish**" pour terminer la création du projet.

Wahou ! Qu'est-ce que c'est que tout ça ?



Regardez dans votre espace de travail, vous devriez avoir un dossier "**Test**" avec ceci dedans :



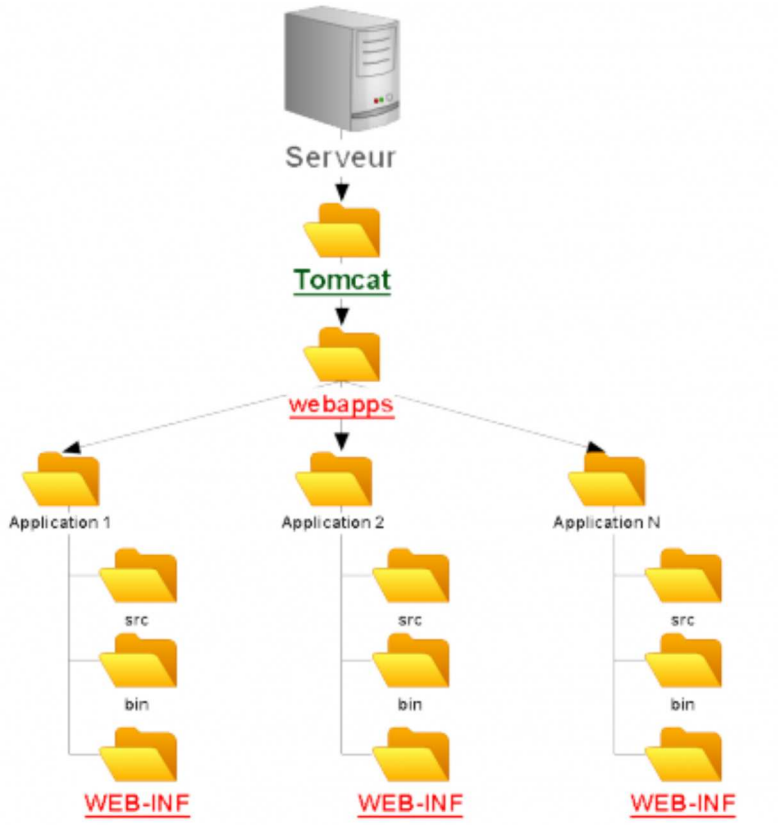
En fait, les plus curieux ont sûrement jeté un rapide coup d'oeil aux dossiers se trouvant dans le répertoire "**webapps**" dans Tomcat...

Ceux-ci ont dû s'apercevoir que les dossiers présents n'étaient pas tous les mêmes, à l'exception du dossier "**WEB-INF**". En effet, ce dernier est le coeur d'une application JEE ! Tout repose sur lui et sur son contenu...

Les autres dossiers sont facultatifs et peuvent changer selon les besoins du programmeur. Dans notre cas nous avons des dossiers "**src**" et "**bin**", dossiers qui doivent vous être familiers...

Eh oui, un dossier pour les sources Java et un pour les .class !

Pour schématiser un peu ce que contient un serveur d'application, voici un petit schéma :




Concernant le contenu du dossier "**WEB-INF**", qui doit bien vous intriguer maintenant, nous allons y arriver très vite ! Dès le chapitre prochain en fait...

Bon, vous venez de créer votre premier projet Tomcat. Nous allons voir maintenant comment le déployer !

Déploiement


Quoi ?

Je savais que ce mot allait vous perturber !
"Déployer" une application JEE veut seulement dire que nous allons la mettre en place dans notre conteneur de servlets : Tomcat !
Après cette manipulation, nous pourrons aller interroger notre serveur d'applications pour voir le rendu de nos pages web...

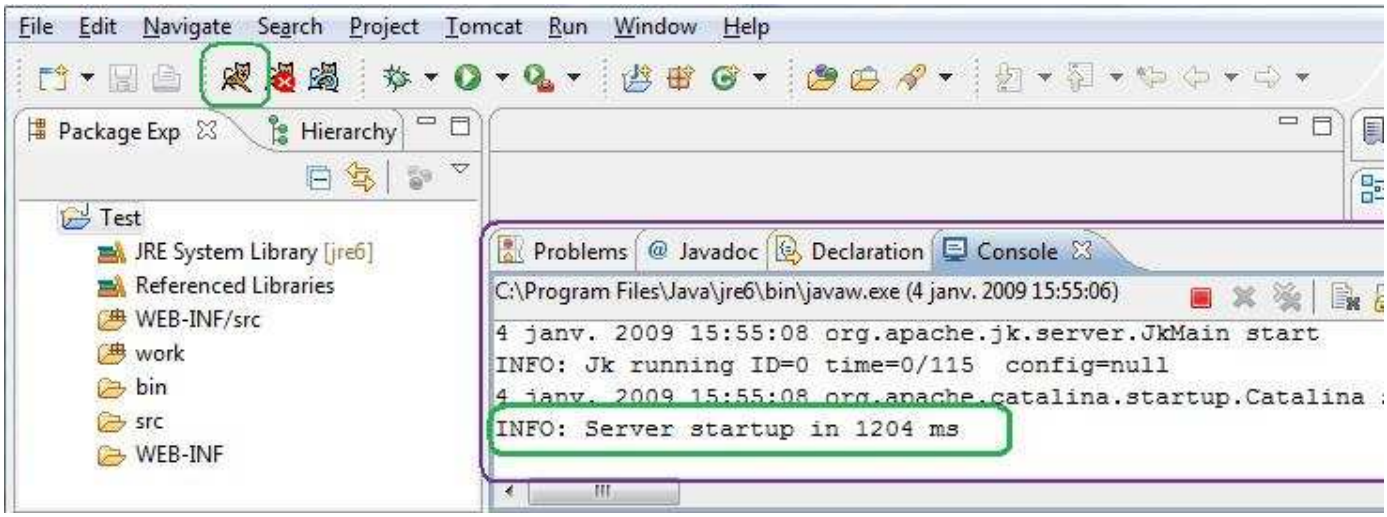

Tu n'arrêtes pas d'appeler Tomcat un "***conteneur de servlets***"... Tu ne pourrais pas expliquer d'avantage ?

Je pourrais, mais je préfère vous faire faire une servlet avant de vous expliquer ceci... Vous comprendrez mieux, je pense.

Vous allez voir que mettre en place une application JEE est tout simple : un simple copier-coller suffit !

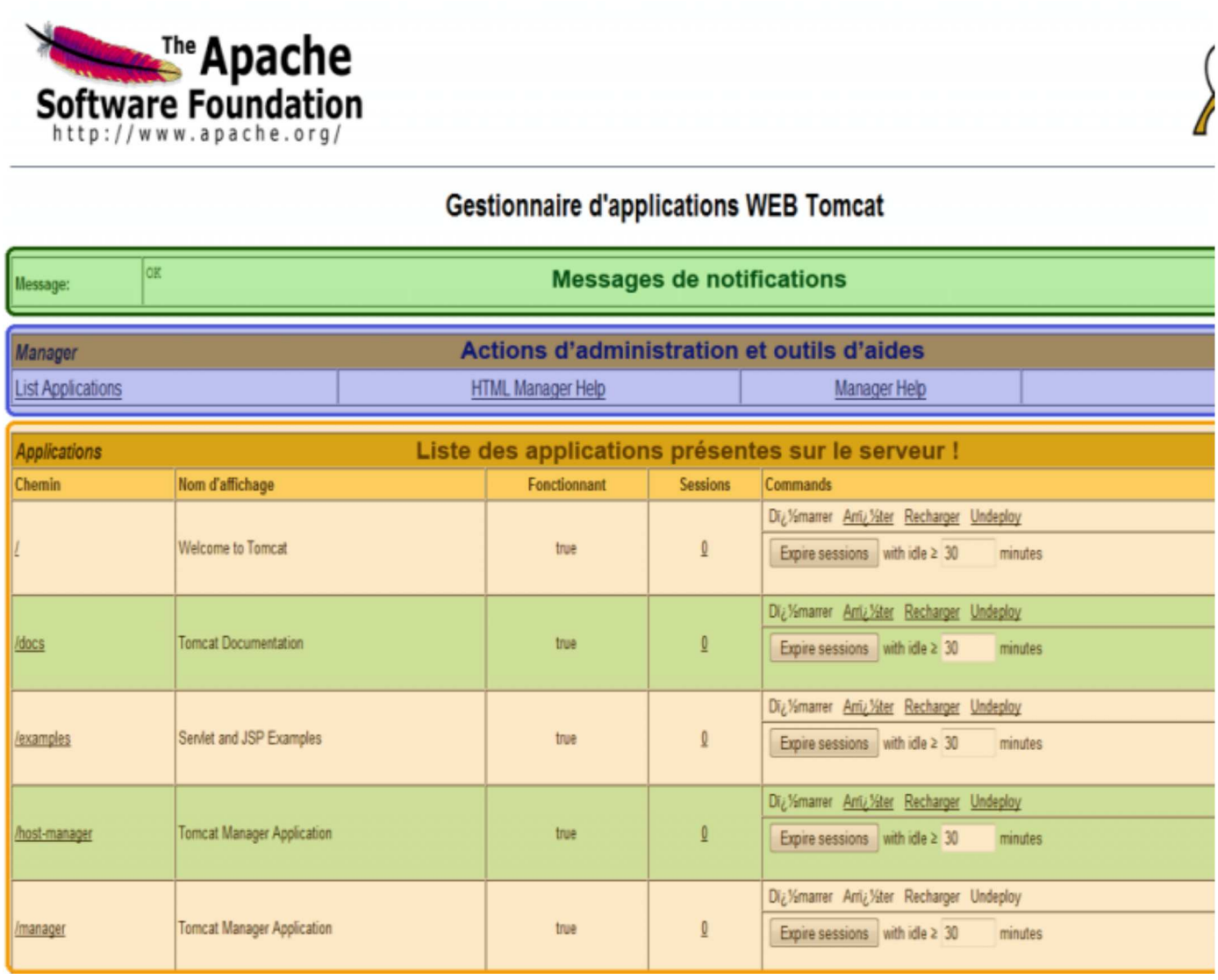

Il existe d'autres façons de faire, mais nous n'en parlerons que plus loin dans le tuto...

Bon, si vous ne l'avez pas encore fait, démarrez votre serveur Tomcat, ce qui devrait vous donner ceci dans la console d'Eclipse :



Et rendez-vous dans la partie d'administration, pour mémoire c'est ici :

Vous voici devant le listing des applications présentes dans Tomcat !

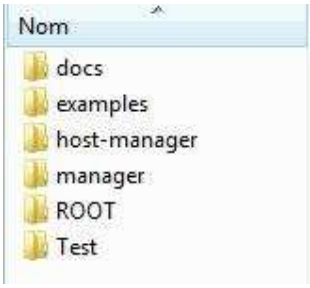


Nous aborderons plus loin les éléments présents en fin de page, de même pour les options qu'offre le listing des applications Tomcat !

Maintenant, le but du jeu est d'importer notre travail dans Tomcat afin de pouvoir visualiser les pages créées et voir apparaître notre (nos) application(s) dans le listing ci-dessus.

Rendez-vous dans votre espace de travail Eclipse, copiez le dossier "**Test**" correspondant à votre application JEE vide, allez dans le dossier d'installation de Tomcat et collez le dossier dans le répertoire "**webapps**".

Vous devriez avoir ceci dans le dossier "**webapps**" :

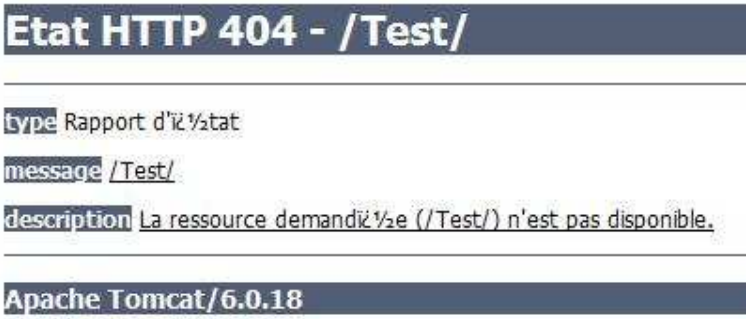


Maintenant, rendez-vous dans l'administration de Tomcat que vous avez déjà ouverte tout à l'heure.
Si vous aviez éteint votre navigateur ou redemandé la page d'administration, vous devriez avoir ceci sous les yeux :

Applications	
Chemin	**Nom d'affichage**
/	Welcome to Tomcat
/Test	
/docs	Tomcat Documentation

Si vous n'avez pas ceci, rafraîchissez la page avec la touche F5.

Il ne vous reste plus qu'à cliquer sur le lien correspondant à notre projet dans l'administration pour y accéder, et voilà :




Ouh là ! Qu'est-ce que c'est que ça ?

Eh oui ! Nous avons déployé un projet vide... Il n'existe aucune page web à afficher...
Du coup, la requête http de notre navigateur vers notre serveur a retourné une erreur : erreur 404.
Les gens qui ont tendance à fouiner sur le web ont dû souvent voir ce genre d'erreur. Il en existe plusieurs en fait, voici un petit listing des codes renvoyés par un serveur :

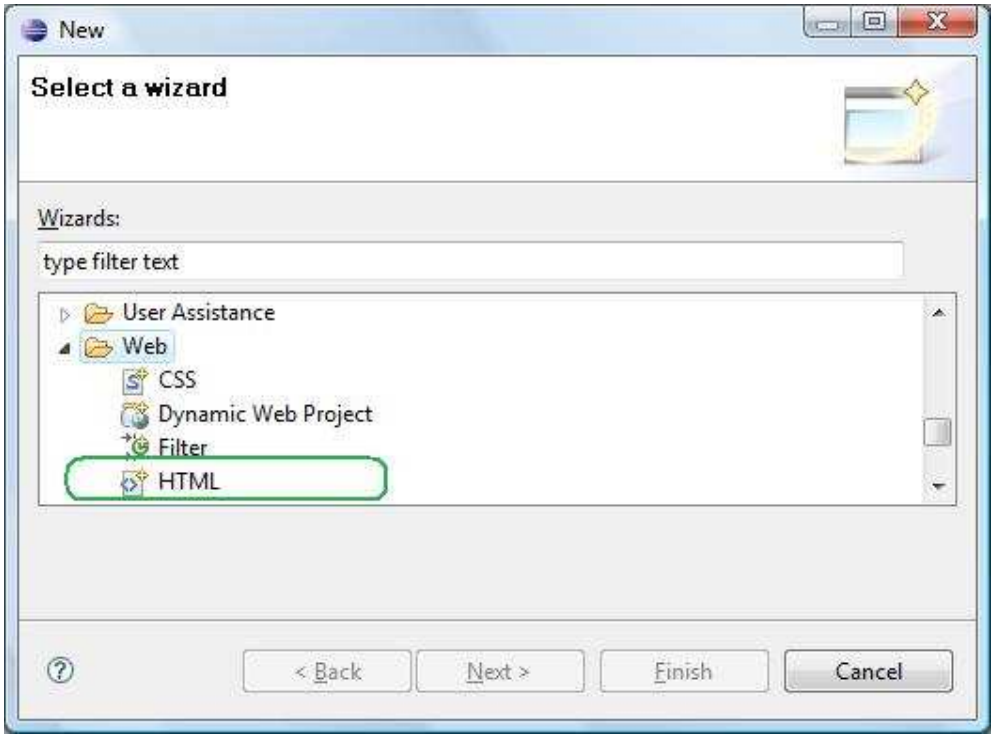
- Code 1XX, réponse provisoire :
  - **100** : OK pour continuer,
  - **101** : le serveur a changé de protocole ;
- Code 2XX, réussite :
  - **200** : (ok) la requête a été traitée avec succès,
  - **201** : (object created, reason = new URI) document créé,
  - **202** : (async completion (TBS)) requête achevée de manière asynchrone,
  - **203** : (partial completion) requête achevée de manière incomplète,
  - **204** : (no info to return) aucune information à retourner,
  - **205** : (request completed, but clear form) requête terminée mais formulaire vide,
  - **206** : (partial GET furfilled) requête GET incomplète ;
- Code 3XX, redirection :
  - **300** : (server couldn't decide what to return) code de retour impossible à déterminer par le serveur,
  - **301** : (object permanently moved) document déplacé définitivement,
  - **302** : (object temporarily moved) document déplacé temporairement,
  - **303** : (redirection with new access method) redirection avec nouvelle méthode d'accès,
  - **304** : (if-modified-since was not modified) le champ 'if-modified-since' n'était pas modifié,
  - **305** : (redirection to proxy, location header specifies proxy to use) redirection vers un proxy spécifié par l'entête,
  - **307** : (HTTP/1.1: keep same verb) HTTP/1.1 ;
- Code 4XX, erreur de requête client :
  - **400** : (invalid syntax) erreur de syntaxe,
  - **401** : (access denied) pas d'autorisation d'accès au document,
  - **402** : (payment required) accès au document payant,
  - **403** : (forbidden) la ressource demandée existe mais vous n'avez pas le droit de l'avoir,
  - **404** : (page not found) la ressource demandée n'existe pas sur le serveur,
  - **405** : (method is not allowed) méthode de requête du formulaire non autorisée,
  - **406** : (no response acceptable to client found) requête non acceptée par le serveur,
  - **407** : (proxy authentication required) autorisation du proxy nécessaire,
  - **408** : (server timed out waiting for request) temps d'accès à la page demandée expiré,
  - **409** : (user should resubmit with more info) l'utilisateur doit soumettre à nouveau avec plus d'infos,
  - **410** : (the resource is no longer available) cette ressource n'est plus disponible,
  - **411** : (the server refused to accept request w/o a length) le server a refusé la requête car elle n'a pas de longueur,
  - **412** : (precondition given in request failed) la précondition donnée dans la requête a échoué,
  - **413** : (request entity was too large) l'entité de la requête était trop grande,
  - **414** : (request URI too long) l'URI de la requête était trop longue,
  - **415** : (unsupported media type) type de média non géré ;
- Code 5XX, erreur du serveur :
  - **500** : (internal server error) erreur interne du serveur,
  - **501** : (required not supported) requête faite au serveur non supprimée,
  - **502** : (error response received from gateway) mauvaise passerelle d'accès,
  - **503** : (temporarily overloaded) service non disponible,
  - **504** : (timed out waiting for gateway) temps d'accès à la passerelle expiré,
  - **505** : (HTTP version not supported) version HTTP non gérée.

Je ne pense pas en avoir oublié...
Vous pouvez voir qu'il existe beaucoup de codes de retour pour un traitement de requête HTTP. Je ne vous cache pas qu'il n'est pas utile de les connaître tous par coeur, sachez seulement retrouver la correspondance entre un code d'erreur et sa signification et ça ira...

Donc, pour en revenir à notre problème, nous avons envoyé une requête HTTP à notre serveur pour qu'il nous retourne une page web. Or, si vous n'avez pas oublié les cours de tonton M@teo, si vous demandez une racine de répertoire à une serveur web, ce qui est notre cas, celui-ci cherche un fichier se nommant "**index.html**" (ou avec une autre extension, mais son nom est index).
Vous êtes d'accord avec moi pour dire que ce fichier n'existe nulle part dans notre répertoire "**Test**" !
Par conséquent, notre serveur nous retourne une **erreur 404 : page not found !**
Pour pallier ce problème, nous allons ajouter une page "**index.html**" dans notre projet grâce à Eclipse.

Attention : la page que vous allez créer sera dans votre projet Eclipse et non dans l'application déployée dans Tomcat !

Pour créer une page HTML à la racine du projet, faites - sous Eclispe - un clic-droit sur le dossier global de votre projet, choisissez ensuite "**new > other**" et dans le menu "**web**", sélectionnez **HTML** :



Eclipse vous génère une bonne dose de code HTML lui-même. J'ai juste rajouté cette ligne :

Code : HTML

```
<h1>Coucou les ZérOs</h1>
```

Vous n'avez plus qu'à copier-coller ce fichier dans le dossier "**webapps/Test**" dans Tomcat et de rafraîchir la page qui retournait une erreur.
Vous passez de ça :

à ça :

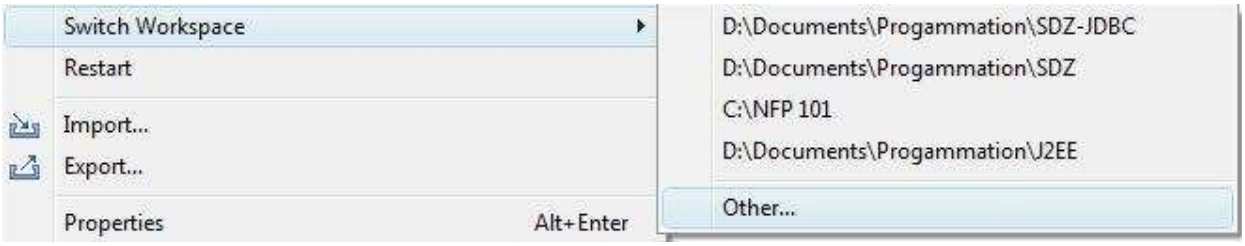
Victoire ! Vous venez de créer et de déployer votre première application JEE !
Je sais, j'en fais beaucoup... Et une page web statique n'a rien d'une application JEE...

Une chose... On va devoir faire des copier-coller sans arrêt ?

Non, bien sûr.
Il y a une alternative et les plus malins d'entre vous l'ont sans doute déjà trouvée : utiliser le dossier "**webapps**" de votre Tomcat comme espace de travail !

Une reconfiguration de Tomcat est peut-être nécessaire...

Pour faire ceci, il vous suffit d'aller dans le menu "**File**", de choisir l'option "**Switch Workspace**" et de sélectionner "**Other...**" comme ceci :



Vous n'avez plus qu'à choisir le dossier "**webapps**" de Tomcat et le tour est joué !

Il va de soi que je pars du principe que le Tomcat en question n'est pas le serveur que les clients utilisent dans le cas où vous travaillez en conditions réelles...
Il vaut mieux ne pas modifier des fichiers consultables directement sans être sûr que la modification fonctionne !

Arrêt, démarrage et suppression

C'est le genre de sous-chapitre que j'affectionne car il va être très court...

Vous avez vu que la page HTML que nous avons créée fonctionne ; cependant, vous devez savoir que l'application fonctionne tant que Tomcat la considère comme démarrée :

Applications

Chemin	Nom d'affichage	Fonctionnant	Se
/	Welcome to Tomcat	true	
/.metadata		true	
/Test		true	

Dans la colonne "**commands**", vous avez le choix entre plusieurs actions :

- **arrêter** : stoppe l'application, elle ne sera plus disponible ;
- **démarrer** : lance l'application ;
- **recharger** : arrête puis démarre l'application ;

- **undeploy** : supprime l'application.

Prenez votre application "**Test**" et cliquez sur "**arrêter**", vous devriez avoir ceci maintenant :

/Test		false	

Essayez d'accéder à la page de test... Impossible ! Si vous voulez y accéder de nouveau, vous devrez démarrer l'application.

Pour supprimer une application sur le serveur Tomcat, il faut tout simplement cliquer sur "**undeploy**".

ATTENTION : cette action supprime l'application dans le listing de Tomcat mais aussi dans le dossier webapps ! !

Bon, vous savez maintenant comment créer, déployer, démarrer, arrêter et supprimer une application JEE. Si on en profitait pour entrer dans le vif du sujet ?
Après le QCM, bien entendu...

Pas mal pour un début !
Il y a aussi quelques points obscurs mais ne vous inquiétez pas, nous allons les éclaircir d'ici peu...

Le mieux est encore de continuer dans notre lancée...

Les servlets : premier opus

Enfin, nous allons commencer à faire du Java !
Vous attendiez ce moment avec impatience mais vous allez peut-être le regretter...
Je plaisante bien sûr.
Nous allons tout prendre à partir de Zéro, c'est rassurant.

Tout d'abord, sachez qu'une servlet est une classe Java comme vous en avez fait des centaines de fois (si ce n'est pas des milliers...). La différence majeure réside dans le fait qu'elle a un rôle à jouer dans le jeu JEE !

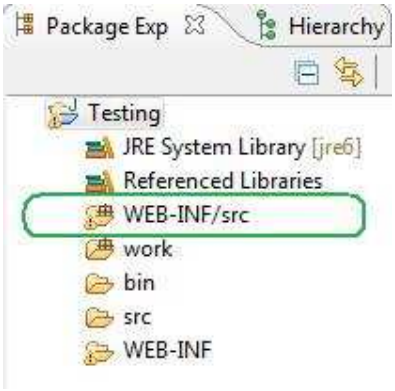
Hello world

À partir de maintenant, je pars du principe que vous savez comment déployer, démarrer, arrêter et consulter une application !

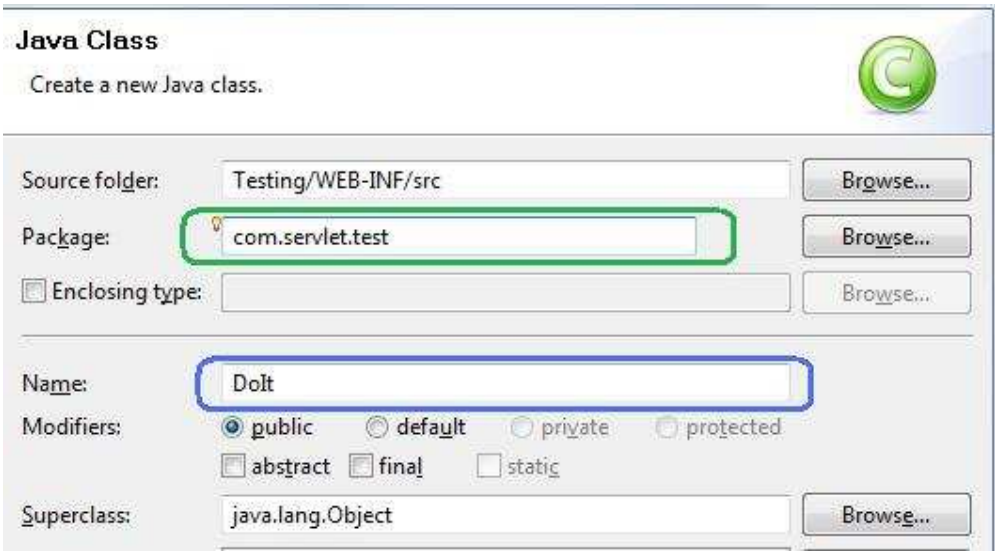
Avant toute chose, vous allez avoir besoin d'un autre plug-in pour Eclipse : le plugin **XML Buddy**.
Celui-ci vous permettra de vous simplifier la vie lorsque vous allez créer des fichier XML (oui, oui, vous allez en faire). Je vous laisse le soin de le télécharger et de décompresser l'archive dans le dossier **plugin** d'Eclipse.
Si vous êtes perdus, une simple recherche de ce plug-in sur Google vous donnera satisfaction !

Nous allons commencer par un *simple* "**hello world**" et je vous assure que rien qu'avec ceci, il y a du travail et plein de choses à voir !

Comme je vous l'avais dit, une servlet est une classe Java que Tomcat va utiliser.
Je vous invite donc à créer une nouvelle classe Java dans le dossier **WEB-INF/src** de votre projet Tomcat :



Faîtes un clic droit sur ce dossier dans Eclipse et choisissez "**new > Class**", nommez-la `DoIt` dans le package `com.servlet.test` , voyez ci-dessous :



Vous vous retrouvez avec une classe ayant un code minimal.
Je vous informe maintenant qu'une servlet digne de ce nom DOIT hériter d'une classe qui s'appelle `HttpServlet`, classe qui se trouve dans le package `javax.servlet.http.HttpServlet` . Vous vous retrouvez donc avec ce code :

Code : Java

```
package com.servlet.test;
import javax.servlet.http.HttpServlet;

public class DoIt extends HttpServlet{

}
```

Nous allons faire en sorte que notre servlet puisse être appelée à la place de notre fichier **index.html**
À partir de maintenant, je vous demande de bien vouloir me faire confiance et de partir du principe que la lumière sera faite sur ce qui va suivre. 😉

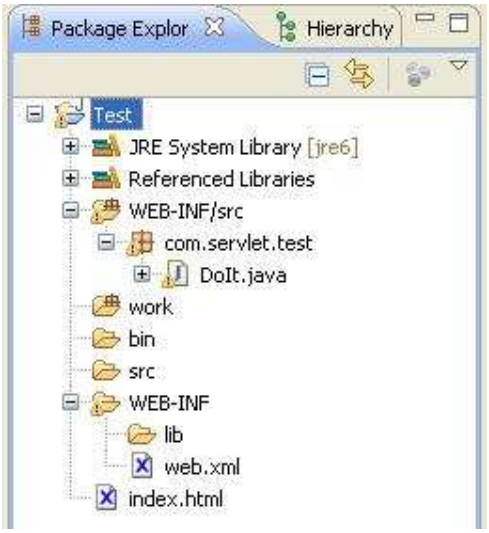
Vous allez compléter le code de votre servlet comme ceci :

Code : Java

```java
package com.servlet.test;
import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;


public class DoIt extends HttpServlet {

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

                response.setContentType("text/html");
                PrintWriter out = response.getWriter();
                out.println("<h1>Coucou toi !</h1>");
        }
}
```

Ne prenez pas peur à cause de tous ces imports... 

Maintenant, nous allons créer un fichier xml qui **DOIT** s'appeler **web.xml** et que vous créerez dans le dossier **WEB-INF** de votre projet, comme ceci :



Pour faire ceci, faites un clic droit sur le dossier **WEB-INF**, choisissez **new > XML Document** et appelez-le **web.xml**. Vous allez poursuivre en y ajoutant ce contenu :

Code : XML

```
<web-app>

	<servlet>
		<servlet-class>com.servlet.test.DoIt</servlet-class>
		<servlet-name>firstServlet</servlet-name>
	</servlet>

	<servlet-mapping>
		<servlet-name>firstServlet</servlet-name>
		<url-pattern>/</url-pattern>
	</servlet-mapping>

</web-app>
```

Faites-moi confiance... 
Maintenant, retournez sur la page d'administration de Tomcat, et sur votre page, vous devriez dorénavant avoir ceci :

Vous venez de faire votre première servlet fonctionnelle !
Maintenant, je peux vous expliquer ce que nous avons fait.

Expliquons tout ça

Vous devez avoir 10 000 questions à me poser et je compte bien y répondre !
Tout d'abord, afin que vous compreniez mieux la façon dont tout ceci fonctionne, vous devez savoir pourquoi on appelle Tomcat un conteneur de servlet.

Vous vous souvenez que le web fonctionne avec un système de **questions - réponses (requêtes HTTP, réponses HTTP)** et que, lorsque vous demandez une page web, vous envoyez **une requête de type GET**.
Les Zéros se souvenant de ces points ont très certainement remarqué que la méthode déclarée dans notre servlet contient le nom du type de requête envoyée : requête de type GET - méthode `doGet` .
Un autre point troublant : **la méthode `doGet` de notre servlet prend deux objets en paramètres et ceux-ci ont des noms qui ressemblent beaucoup à *requête HTTP* et *réponse HTTP* !**


Est-ce que ça veut dire que notre servlet reçoit la requête HTTP et retourne la réponse HTTP ?

Vous y êtes presque... 
En fait, ce n'est pas la servlet qui reçoit tout ceci directement : **c'est Tomcat !**
C'est le conteneur de servlets qui reçoit et envoie les requêtes HTTP... Dès que ce dernier reçoit une requête, il instancie deux objets :

- un objet `HttpServletRequest` contenant les informations de la requête HTTP ;
- un objet `HttpServletResponse`, servant à fournir la réponse attendue.

Une fois ceci fait, il va utiliser la servlet correspondant à la requête demandée et va invoquer la méthode adéquate, ici `doGet(HttpServletRequest request, HttpServletResponse response)` !


La méthode adéquate ? Tu veux dire qu'il peux y avoir plusieurs méthodes dans une servlet ?

Vous avez vu juste !
Pour faire court, il y existe une méthode pour chaque type de requête HTTP :

- **GET** : méthode `doGet()` , c' est la méthode la plus courante pour demander une ressource. Une requête GET est sans effet sur la ressource, il doit être possible de répéter la requête sans effet ;
- **POST** : méthode `doPost()` , cette méthode doit être utilisée lorsqu'une requête modifie la ressource ;

- **HEAD** : méthode `doHead()` , cette méthode ne demande que des informations sur la ressource, sans demander la ressource elle-même ;
- **OPTIONS** : méthode `doOptions()` , cette méthode permet d'obtenir les options de communication d'une ressource ou du serveur en général ;
- **CONNECT** : méthode `doConnect()` , cette méthode permet d'utiliser un proxy comme un tunnel de communication ;
- **TRACE** : méthode `doTrace()` , cette méthode demande au serveur de retourner ce qu'il a reçu, dans le but de tester et d'effectuer un diagnostic sur la connexion ;
- **PUT** : méthode `doPut()` , cette méthode permet d'ajouter une ressource sur le serveur ;
- **DELETE** : méthode `doDelete()` , cette méthode permet de supprimer une ressource du serveur.

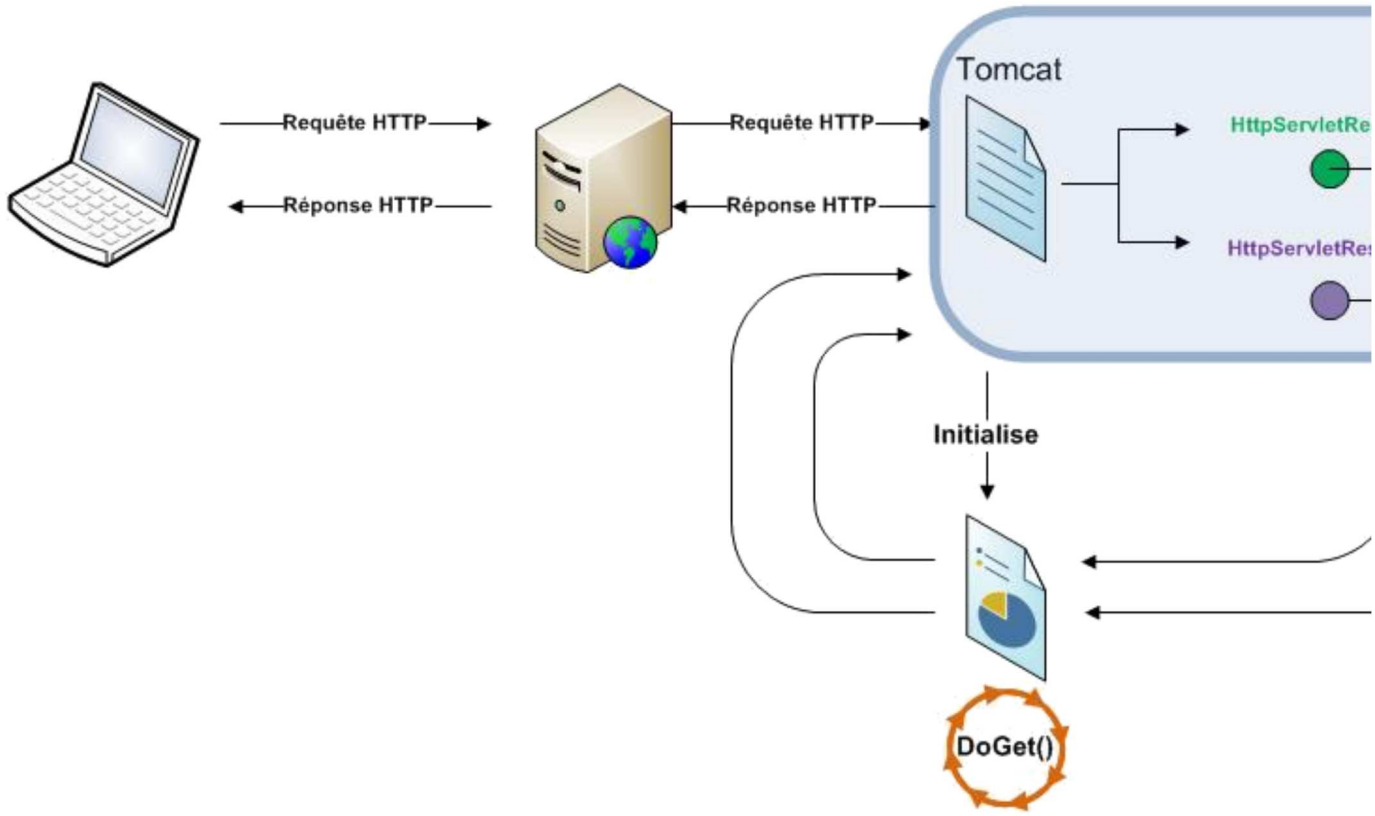
Je n'ai pas inventé ces définitions, celles-ci sont tirées de Wikipédia...

Pas de panique, dans 99.99999999 % des cas, vous utiliserez des requêtes de type **GET** ou **POST** !

Attends deux minutes ! Quand doit-on utiliser tel ou tel type de requête ?

Nous reviendrons sur ce point très bientôt. Pour le moment, essayez de comprendre comment fonctionne Tomcat lorsqu'il reçoit des requêtes HTTP.

Reprenons. Dans les grandes lignes, Tomcat reçoit une requête HTTP d'un certain type, il utilise un objet **requête** et un objet **réponse**, il instancie la servlet correspondante à la requête et invoque la méthode adéquate.
Le contenu de ladite méthode est exécuté, Tomcat récupère les objets et nous retourne la réponse HTTP ! Voici un schéma récapitulatif :



Ensuite, le contenu de la servlet est très simple :

- `response.setContentType("text/html")` : définit le type de réponse, ici, on retourne une page HTML ;
- `PrintWriter out = response.getWriter()` : on récupère un objet permettant d'écrire dans la future page HTML ;
- `out.println("....")` : écrit dans la page.

Les plus observateurs d'entre vous ont dû remarquer qu'une servlet n'a pas de constructeur !
Ne vous alarmez pas sur ce point pour le moment, nous aurons l'occasion d'y revenir.


D'accord, on a compris. Par contre, comment Tomcat sait quelle servlet instancier ?

Avec le dernier point à éclaircir pour le moment : le fichier **web.xml**.
Vous avez compris ce qu'il se passe lorsque une requête HTTP est reçue par Tomcat.
Avant tout ceci, il se passe quelque chose de vital : ***le mapping des servlets !***

Au lancement du serveur Tomcat, vous pouvez voir tout plein de choses incompréhensibles dans la console d'Eclipse, en voici une partie :

```
7 janv. 2009 14:41:54 org.apache.jk.common.ChannelSocket init
INFO: JK: ajp13 listening on /0.0.0.0:8009
7 janv. 2009 14:41:54 org.apache.jk.server.JkMain start
INFO: Jk running ID=0 time=0/31  config=null
7 janv. 2009 14:41:54 org.apache.catalina.startup.Catalina start
INFO: Server startup in 549 ms
7 janv. 2009 14:58:25 org.apache.catalina.startup.HostConfig checkResources
INFO: Reloading context [/Test]
```

Et quelque part dans tout ce charabia, il y a ceci :

```
7 janv. 2009 14:41:54 org.apache.catalina.core.ApplicationContext log
INFO: ContextListener: contextInitialized()
```

Ici on voit qu'une chose obscure nommée "***contexte***" a été initialisée !
Ceci est tout simplement une sorte d'annuaire pour Tomcat : la définition des applications sur le serveur ainsi que la structure de celles-ci.

Une question de contexte

Chaque application sur le serveur a une structure et cette structure est définie dans le fichier **web.xml**.
Ce fichier est utilisé par Tomcat pour faire une relation entre une requête HTTP et une servlet.
On dit aussi que le fichier sert à définir le contexte de votre application.


Un seul fichier web.xml par application !

Vous avez dû avoir peur de ce fichier, mais il n'y a pas de quoi, je vous assure...
Nous allons voir comment ce fichier est construit et à quoi correspond tout ceci... Tout d'abord, vous avez sûrement vu que le contenu est englobé par une balise :

Code : XML

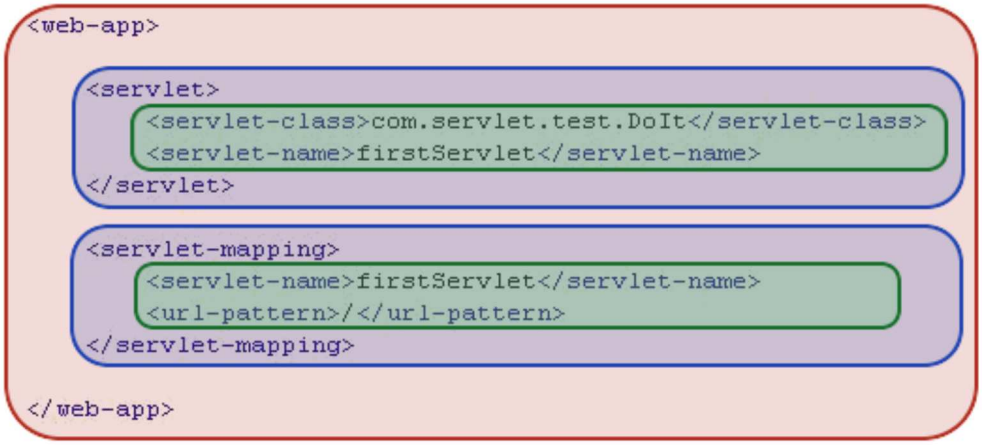
```
<web-app>
...
...
...
</web-app>
```

Cet encadrement, bien qu'incomplet, **est OBLIGATOIRE**.


Incomplet ?

Oui, il manque des informations à cette balise, mais nous les ajouterons plus tard. Le but étant toujours de comprendre le fonctionnement de Tomcat.

Donc, tout fichier web.xml doit être défini de la sorte ! Ensuite, nous trouvons deux éléments de même rang : par là, entendez balises xml se suivant et étant non imbriquées. Voici un schéma :



Chaque couleur correspond à un niveau dans le fichier xml.
Vous m'avez compis, les éléments de même rang sont `<servlet></servlet>` et `<servlet-mapping></servlet-mapping>` .
Dans ces éléments, nous allons définir les noms de notre servlet !


Tu veux dire que notre servlet a plusieurs noms ?

Tout à fait. En fait, notre servlet a :

- son nom, le nom de la classe complet avec le package ;
- un nom de mappage interne à l'application, nom que la servlet porte dans l'application ;
- nom de la servlet pour le client.

La servlet est tout d'abord définie dans l'application via l'élément `<servlet></servlet>` . Nous trouvons deux éléments dans ce dernier :

- `<servlet-class></servlet-class>` : code réel de la servlet ;
- `<servlet-name></servlet-name>` : nom interne à l'application.

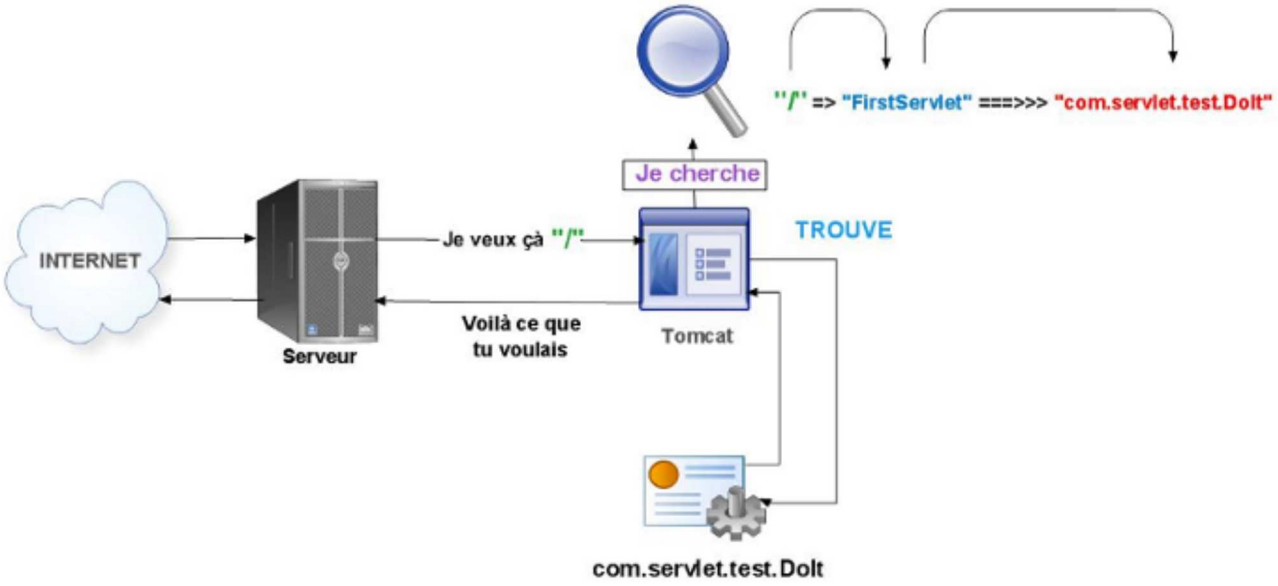
Ensuite, nous trouvons `<servlet-mapping></servlet-mapping>` contenant les informations de paramétrage client :

- `<servlet-name></servlet-name>` : nom interne à l'application ;
- `<url-pattern></url-pattern>` : nom qui apparaît côté client.

Voici un petit récapitulatif de ce qu'il se passe.

- 1/ Tomcat, à son lancement, prend connaissance des données mentionnées dans le fichier de configuration.
- 2/ Il reçoit une requête HTTP demandant "/", dans notre cas bien sûr.
- 3/ Il sait que ce nom est associé au nom de servlet "FirstServlet" qui, lui, correspond à la servlet `com.servlet.test.DoIt` .
- 4/ Celle-ci est instanciée et la méthode adéquate est invoquée.
- 5/ La réponse est récupérée par Tomcat qui renvoie cette dernière au client.

Un petit schéma (les instanciations des objets requêtes et réponses sont tacites) :



Pourquoi avoir autant de noms pour une servlet ?

Tout simplement pour faciliter les changements éventuels. Une chose importante à savoir : une servlet peut être utilisée pour faire plusieurs choses... Nous allons y venir.

Imaginez que vous appeliez votre servlet depuis une vingtaine de pages et que vous utilisiez le nom réel de votre servlet... Déjà, les utilisateurs connaîtront la hiérarchie de vos classes, et tous ceux qui ont déjà fait des sites web savent qu'il vaut mieux que le client en sache le moins possible.
Ensuite, si vous avez changé le nom de votre servlet, vous allez avoir 20 pages à reprendre une par une pour faire les changement alors que là, tout est automatique !
Vous pouvez changer le nom de votre servlet, ce n'est pas grave puisque nous utilisons un autre nom.

Vous avez appris beaucoup de choses dans ce chapitre. Le temps est venu de faire une pause et d'aller faire un tour sur le QCM...

Votre première servlet est au point maintenant et vous êtes toujours en vie !
Mais ne vous leurrez pas, on ne s'arrête pas là. La plateforme JEE offre bien d'autres outils à utiliser.
De plus, je vous avais dit, au début de ce tutoriel, que les servlets fonctionnent avec ce qu'on appelle des JSP.
Si vous voulez savoir ce qu'il en est, rendez-vous au chapitre suivant !

Gérer l'affichage

Dans le précédent chapitre nous avons réussi à faire notre première servlet !
Les Zéros qui ont l'habitude de faire des pages web se sont vite rendu compte que, si nous poursuivons dans cette voie, les choses seront loin d'être simples...

En effet, dans le langage HTML, le caractère " sert **TRÈS SOUVENT !**
Or en Java aussi, donc si vos balises HTML ne sont pas "nature" sans CSS, voici ce que ça pourrait donner :

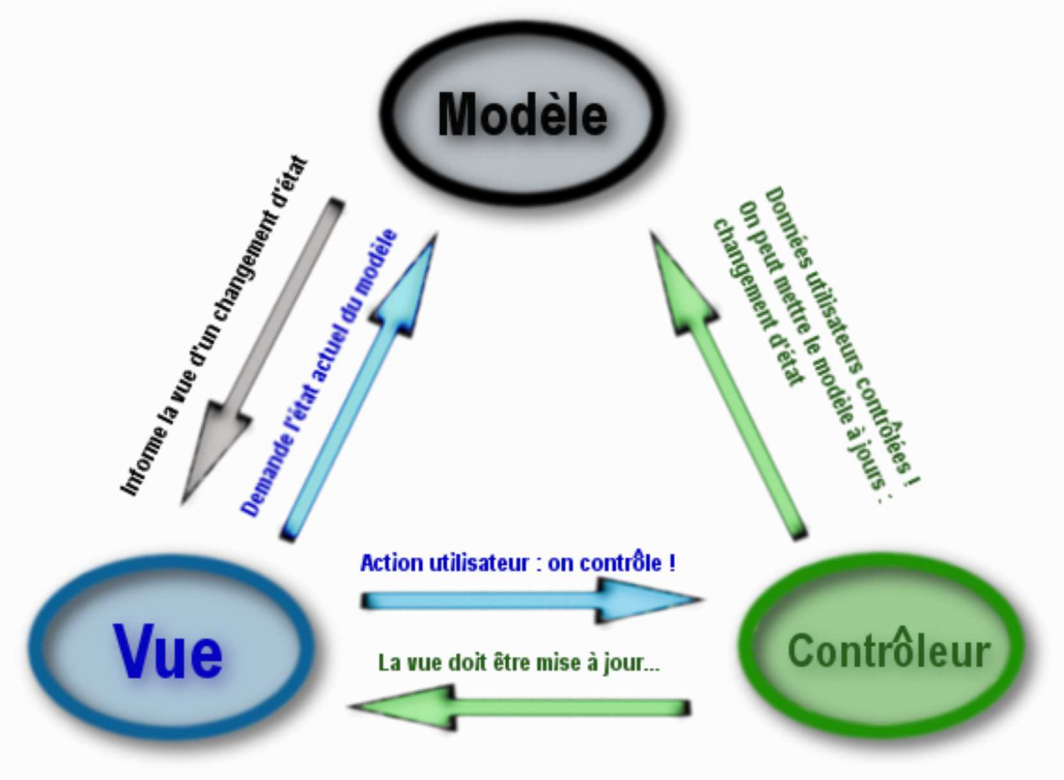
Code : Java

```
out.println("<html>");
out.println("<body onLoad=\"alert('bonjour');\">");
out.println("<p style=\"color:red\" id=\"monId\" onClick=\"alert('toto');
\">");
out.println("Pas facile à lire tout ça...");
out.println("</p>");
out.println("</body>");
out.println("</html>");
```

D'ailleurs, pour être honnête, les servlets ne servent pas à écrire du code Java dans des pages web !
En fait, la plateforme JEE implémente le pattern MVC. Je vous invite très vivement à aller rejeter un coup d'oeil au tuto concerné...

MVC et JEE

Voici le schéma que je vous avais proposé dans le chapitre sur MVC :



Avec la plate-forme JEE, vous allez devoir utiliser ce pattern. Disons qu'il est plus ou moins encapsulé dedans !

Bon, si nous avons à utiliser ce pattern, à quoi correspond la servlet que nous venons de réaliser ?

Réfléchissez un peu... Dans le pattern MVC, il y a un objet qui a pour rôle de récupérer des demandes, de les traiter et de retourner ce qu'on lui a demandé : **le contrôleur !**
En effet, nous avons fait en sorte que notre servlet écrive elle-même du code HTML, mais dans la plupart des cas, les servlets ont pour rôle de récupérer les requêtes client et d'agir en conséquence.

Attends : dans ce cas, que sont le modèle et la vue du pattern MVC ?

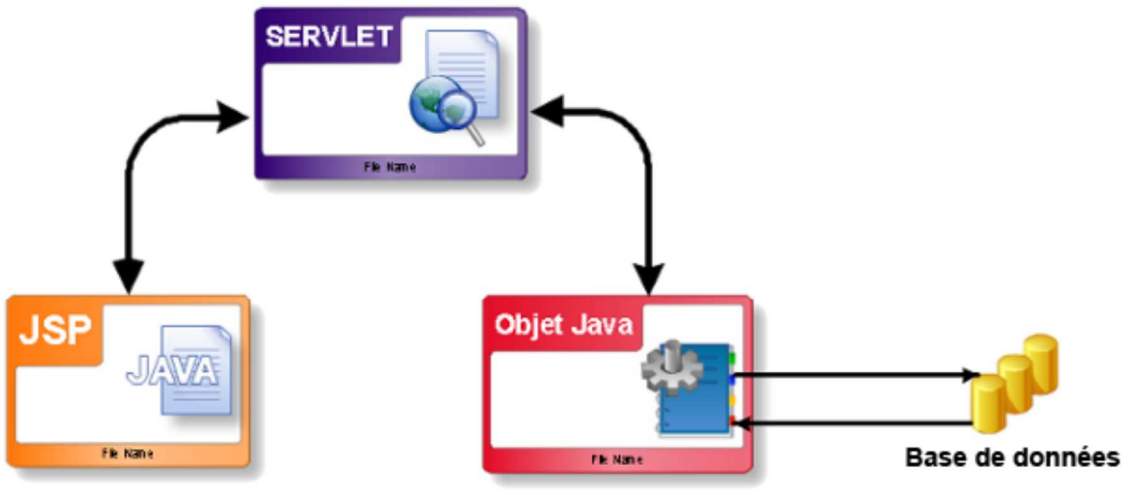
Le modèle peut être divers et varié, comme tout modèle digne de ce nom :

- un objet Java basique ;
- un objet Java devant faire appel à une base de données ;

- ...

La vue, elle, reste une page web contenant du code HTML, à la différence près que celle-ci contiendra aussi du code Java : on appelle ces pages web des **J**ava **S**erver **P**ages, ou JSP pour les intimes !

Donc, si nous appliquons le pattern MVC à l'architecture JEE, ça nous donne :



Vous remarquerez sûrement que des flèches de dépendance ont disparu ! En effet, vous allez voir que la vue est appelée par le contrôleur et reçoit les informations du modèle via ce dernier.
Je vous propose maintenant de voir à quoi ressemble une page JSP et comment la lier à notre servlet...

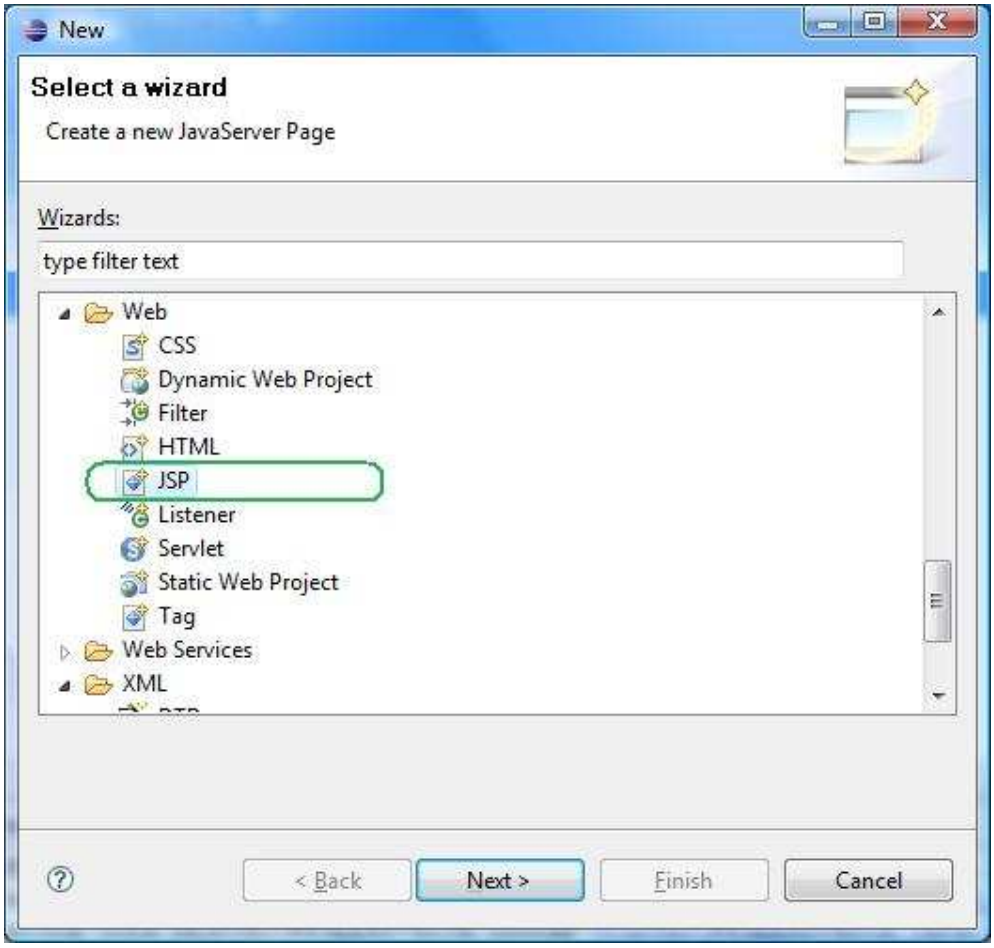
V comme JSP

Comme vous le savez déjà, une JSP est une page web contenant du code Java.
Le dit code est introduit dans des pages web via des balises spécifiques `<% %>` ; ensuite, tout ce que vous mettrez à l'intérieur est du pur code Java, voici un exemple :

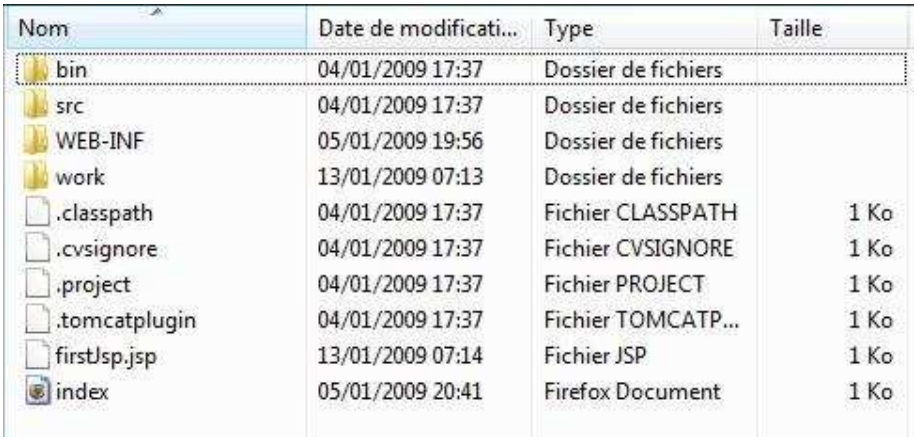
Code : JSP

```
<html>
<body>
 <% out.println("<h1>Une JSP !</h1>"); %>
</body>
</html>
```

Nous allons d'ailleurs l'ajouter dans notre projet !
Pour ce faire, il vous suffit de faire un clic droit sur votre projet et de choisir "**new/other/web/JSP**".



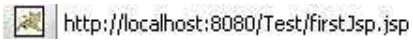
Appelez-la "**firstJsp**". Votre JSP se trouve donc à la racine du projet : à côté du fichier **index.html** pour ceux qui auraient continué de travailler sur le même projet Tomcat.



Nom	Date de modificati...	Type	Taille
bin	04/01/2009 17:37	Dossier de fichiers	
src	04/01/2009 17:37	Dossier de fichiers	
WEB-INF	05/01/2009 19:56	Dossier de fichiers	
work	13/01/2009 07:13	Dossier de fichiers	
.classpath	04/01/2009 17:37	Fichier CLASSPATH	1 Ko
.cvsignore	04/01/2009 17:37	Fichier CVSIGNORE	1 Ko
.project	04/01/2009 17:37	Fichier PROJECT	1 Ko
.tomcatplugin	04/01/2009 17:37	Fichier TOMCATP...	1 Ko
firstJsp.jsp	13/01/2009 07:14	Fichier JSP	1 Ko
index	05/01/2009 20:41	Firefox Document	1 Ko

Eclipse vous génère du code, beaucoup de code, toutes les entêtes HTML en fait... pour notre test nous n'en avons pas besoin : effacez le contenu du fichier et remplacez-le par mon code (non mais !).

Ensuite, vous connaissez la musique, rendez-vous dans l'administration de Tomcat, rentrez dans votre projet !
Pour accéder à votre JSP, il suffit de rajouter son nom dans la barre d'adresse de votre navigateur :



Ce qui vous donne ceci :

Une JSP !

Félicitations, vous avez votre première JSP !

Ici, nous avons affiché un message, mais on peut faire plein d'autres choses dans une JSP... Utiliser des objets, faire des boucles...

Voici notre JSP avec un peu plus de code :

Code : JSP

```
<html>
<body>
 <% out.println("<h1>Une JSP !</h1>"); %>
 <p>Alors...</p>
 <%
   String[] list = new String[]{"Et de un", "Et de deux", "Et de trois"};
   out.println("<ul>");

   //On peut même y mettre des commentaires...

   /*
Même des commentaires multilignes
*/

   //Allons-y pour une boucle...
   for(String str : list)
     out.println("<li>" + str + "</li>");

   out.println("</ul>");
 %>
</body>
</html>
```

Ce qui nous donne :

Les commentaires sont ignorés et n'apparaissent pas... Et encore heureux...

Maintenant, je vous propose de voir comment faire en sorte qu'une servlet puisse utiliser une JSP afin de la fournir au client. Vous allez voir, c'est simple comme tout...
Pour ce faire, nous allons créer une nouvelle servlet que nous appellerons `InvokeJsp`, toujours dans le package `com.servlet.test` , dont voici le contenu :

Code : Java

```java
package com.servlet.test;

import java.io.IOException;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class InvokeJsp extends HttpServlet {

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

		RequestDispatcher dispatch = request.getRequestDispatcher("firstJsp.jsp");
		dispatch.forward(request, response);
	}
}
```

Il vous faut ensuite modifier le fichier **web.xml** comme ceci :

Code : XML

```xml
<web-app>

	<servlet>
		<servlet-class>com.servlet.test.DoIt</servlet-class>
		<servlet-name>firstServlet</servlet-name>
	</servlet>

	<servlet>
		<servlet-class>com.servlet.test.InvokeJsp</servlet-class>
		<servlet-name>invoke</servlet-name>
	</servlet>

	<servlet-mapping>
		<servlet-name>firstServlet</servlet-name>
		<url-pattern>/</url-pattern>
	</servlet-mapping>

	<servlet-mapping>
		<servlet-name>invoke</servlet-name>
		<url-pattern>/invoke</url-pattern>
	</servlet-mapping>

</web-app>
```

Nous avons conservé la configuration de notre première servlet et nous avons ajouté la configuration pour la seconde.
Vous devez remarquer l'ordre de la déclaration : le fichier commence par définir tous les noms internes de nos servlets (1) et ensuite tous les chemins (2) pour y accéder de l'extérieur :

```xml
<web-app>
    <servlet>
        <servlet-class>com.servlet.test.DoIt</servlet-class>
        <servlet-name>firstServlet</servlet-name>
    </servlet>

    <servlet>
        <servlet-class>com.servlet.test.InvokeJsp</servlet-class>
        <servlet-name>invoke</servlet-name>
    </servlet>
```

1

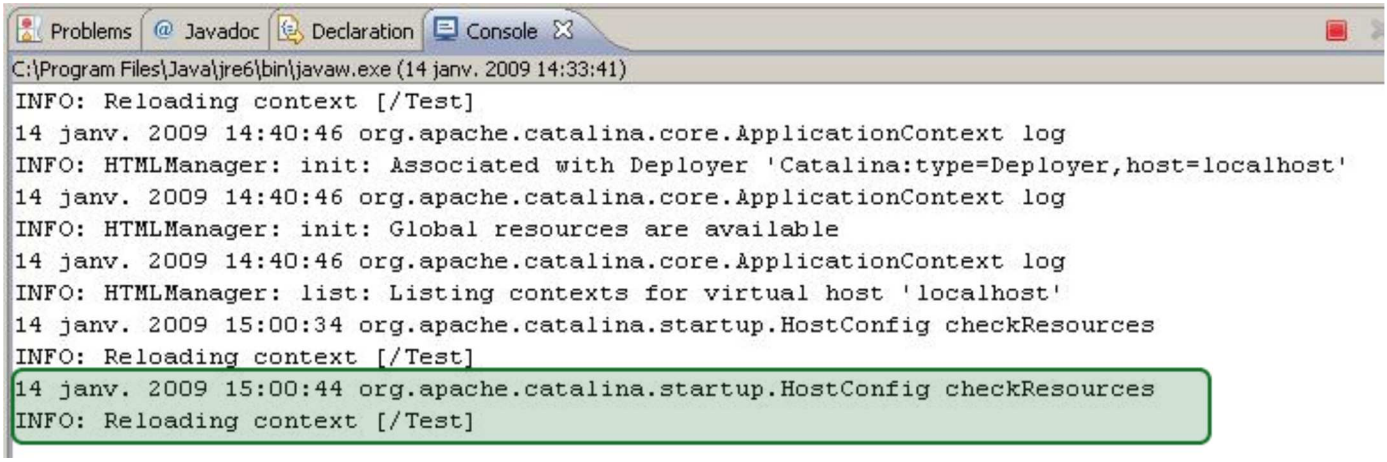
```xml
    <servlet-mapping>
        <servlet-name>firstServlet</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>

    <servlet-mapping>
        <servlet-name>invoke</servlet-name>
        <url-pattern>/invoke</url-pattern>
    </servlet-mapping>
</web-app>
```

2

Vous devez savoir une chose importante concernant Tomcat et le fichier **web.xml**.

Si vous n'avez pas opté pour la solution d'utiliser le dossier **webapps/** comme workspace, vous serez obligés de redémarrer Tomcat afin qu'il puisse prendre en compte les modifications apportées au fichier **web.xml** ! Il va donc le relire et le tour sera joué...

Pour ceux qui ont opté pour la solution fournie, s'ils attendent quelques secondes après avoir modifié le fichier **web.xml**, un message identique à celui-ci s'affichera :



Ce qui signifie que Tomcat a rechargé le contexte de l'application **Test** !
Vous pouvez aller à l'adresse suivante : **localhost:8080/Test/invoke** et vous aurez le contenu de votre JSP mais cette fois, celui-ci nous a été fourni par notre servlet `InvokeJsp`.

Vous avez compris que ceci se faisait par le biais de ces deux lignes de code :

Code : Java

```
RequestDispatcher dispatch = request.getRequestDispatcher("firstJsp.jsp");
dispatch.forward(request, response);
//en contracté :
request.getRequestDispatcher("firstJsp.jsp").forward(request, response);
```

Ces lignes de codes signifient en gros que notre servlet, après avoir fait son travail (ici elle n'en fait aucun, mais bon...) confie le tout à la page **firstJsp.jsp**. Cette page récupère les objets passés en paramètre de la servlet, le code Java à l'intérieur est interprété, ce qui génère du code HTML.

Au final, notre page JSP ne contiendra plus que du code HTML, elle est retournée à Tomcat, qui nous la retourne ! 

 Tu veux dire que nous pouvons utiliser les objets `HttpServletRequest` et `HttpServletResponse` dans nos pages JSP ?

Oui, ainsi que tout autre objet Java ! 
Par contre, les deux objets cités ci-dessus mis à part, ainsi que l'objet `PrintWriter` (`out` est un objet `PrintWriter`), un import sera nécessaire !
Je vous propose de modifier quelque peu votre JSP...

Code : JSP

```
<%@ page import="java.util.Enumeration, java.text.SimpleDateFormat,
java.util.Date" %>
<html>
<body>
<% out.println("<h1>Nous sommes le : " + new SimpleDateFormat("dd/MM
/yyyy").format(new Date()) + "</h1>"); %>
<h1>il est : <%=new SimpleDateFormat("HH:mm:ss").format(new Date()) %></h1>

<%
//Nous allons récupérer les en-têtes
Enumeration e = request.getHeaderNames();
while(e.hasMoreElements()){
        String element = e.nextElement().toString();
        out.println("<pre>" + element + " :
" + request.getHeader(element) + "</pre>");
}
%>
</body>
</html>
```

Si cette ligne de code, `<%=new SimpleDateFormat("HH:mm:ss").format(new Date()) %>` , vous turlupine, ne vous tracassez pas trop...
L'objet `SimpleDateFormat` prend une chaîne de caractères ("HH:mm:ss") correspondant à "Heures : Minutes : Secondes". On invoque ensuite la méthode format qui met en forme un objet `Date` : ***cette ligne de code nous retourne donc l'heure courante***. 
Tandis que ce format de date : "dd/MM/yyyy" ***nous retourne la date courante***.

 Rien ne vous empêche de faire vos imports où vous le souhaitez dans la JSP. Personnellement, je préfère les avoir en haut de page, mais ce n'est qu'un avis personnel !

Vous pouvez tout aussi bien ne pas faire d'import, mais vous devrez spécifier le nom complet de l'objet utilisé (avec le package !).
Donc, partant de ce postulat, ce code :

Code : JSP

```
<%@ page import="java.util.Enumeration, java.text.SimpleDateFormat,
java.util.Date" %>
<html>
<body>
<% out.println("<h1>Nous sommes le : " + new SimpleDateFormat("dd/MM
/yyyy").format(new Date()) + "</h1>"); %>
<h1>il est : <%=new SimpleDateFormat("HH:mm:ss").format(new Date()) %></h1>

<%
//Nous allons récupérer les en-têtes
Enumeration e = request.getHeaderNames();
while(e.hasMoreElements()){
        String element = e.nextElement().toString();
        out.println("<pre>" + element + " :
" + request.getHeader(element) + "</pre>");
}
%>
</body>
</html>
```

Nous donne le même résultat que ce code :

Code : JSP

```
<html>
<body>
<% out.println("<h1>Nous sommes le : " +
                    new java.text.SimpleDateFormat("dd/MM/yyyy")
                    .format(new java.util.Date()) +
                "</h1>"); %>
<h1>il est : <%= new java.text.SimpleDateFormat("HH:mm:ss")
                    .format(new java.util.Date()) %></h1>

<%
        //Nous allons récupérer les en-têtes
        java.util.Enumeration e = request.getHeaderNames();
        while(e.hasMoreElements()){
                String element = e.nextElement().toString();
                out.println("<pre>" + element + " :
" + request.getHeader(element) + "</pre>");
        }
%>
</body>
</html>
```

Qui, au final, vous donnera à l'affichage :

Vous noterez la façon dont les imports sont faits avec les JSP : `<%@ page import="package.Class, autrepackage.AutreClass, ..." %>` : **retenez bien ceci !**

Vous aurez remarqué que j'ai utilisé un raccourci pour écrire des données dans la page : `<%= "Une chaîne"%>` . Ceci est équivalent à `<% out.println("Une chaîne");%>`

Par contre, avec ce raccourcis, vous ne DEVEZ PAS terminer votre instruction avec un ";" !

Ceci est correct : `<%= "coucou" %>` .
Alors que ceci ne l'est pas : `<%= "coucou" ; %>` .
Nous verrons pourquoi dans la partie II. 

Voilà : vous venez de voir comment faire en sorte qu'une servlet délègue l'affichage à une page JSP ! Il reste encore un point à voir pour pouvoir utiliser MVC : le modèle.
Comme je vous l'avais dit plus haut, le modèle peut être divers et varié. Pour nous simplifier cet apprentissage, dans un premier temps, nous allons utiliser un simple objet Java, aussi appelé POJO.
Qu'attendons-nous ? Je vous le demande !

Le modèle

Voici un simple objet nous retournant une couleur de façon aléatoire. Ce dernier est placé dans le package `com.servlet.test.model` .

Code : Java

```java
package com.servlet.test.model;

public class ColorModel {

	private String[] colors = new String[]
{"red", "green", "yellow", "purple", "pink", "silver", "orange"};
	private String color;

	public ColorModel(){
		this.color = colors[Double.valueOf(Math.random()*7).intValue()];
	}

	public String getColor(){
		return this.color;
	}
}
```

Nous allons utiliser ce dernier lorsque vous appellerez une page. Tomcat va utiliser la servlet que nous aurons paramétrée, elle va utiliser cet objet, le passer à une JSP qui sera interprétée et renvoyée ! 

Il ne nous reste plus qu'à créer une servlet, une JSP et mettre à jour le fichier **web.xml** ! Fastoche !

Voici notre servlet :

Code : Java

```java
package com.servlet.test;

import java.io.IOException;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import com.servlet.test.model.ColorModel;

public class ColorServlet extends HttpServlet {

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

		request.setAttribute("colorString", new ColorModel().getColor());
		request.getRequestDispatcher("colorJsp.jsp").forward(request, response);
	}

}
```

 Vous passez des attributs à votre JSP grâce à la méthode `setAttribute(String name, Object value);` de l'objet `HttpServletRequest` ! Lorsque vous devrez utiliser ces attributs, c'est via le nom passé en premier paramètre de la méthode que vous les récupérerez !

Notre JSP :

Code : JSP

```jsp
<%@ page import="com.servlet.test.model.ColorModel" %>
<html>
<body>
<h1 style="color:<%=request.getAttribute("colorString") %>">
        Récupération de l'attribut "<em>colorString</em>"
</h1>
</body>
</html>
```

Vous voyez bien comment on récupère les attributs passés à notre JSP depuis la servlet : `request.getAttribute(String attributeName); .`

Et notre fichier **web.xml** :

Code : XML

```xml
<web-app>

        <servlet>
                <servlet-class>com.servlet.test.DoIt</servlet-class>
                <servlet-name>firstServlet</servlet-name>
        </servlet>

        <servlet>
                <servlet-class>com.servlet.test.InvokeJsp</servlet-class>
                <servlet-name>invoke</servlet-name>
        </servlet>

        <servlet>
                <servlet-class>com.servlet.test.ColorServlet</servlet-class>
                <servlet-name>color</servlet-name>
        </servlet>


        <servlet-mapping>
                <servlet-name>firstServlet</servlet-name>
                <url-pattern>/</url-pattern>
        </servlet-mapping>

        <servlet-mapping>
                <servlet-name>invoke</servlet-name>
                <url-pattern>/invoke</url-pattern>
        </servlet-mapping>

        <servlet-mapping>
                <servlet-name>color</servlet-name>
                <url-pattern>/color</url-pattern>
        </servlet-mapping>

</web-app>
```

Vous savez ce qu'il vous reste à faire si vous voulez allez voir ce que ça donne...
Voici deux-trois screens de ce que j'ai pu obtenir :

Screen 1 :

Screen 2 :

Récupération de l'attribut "*colorString*"

Vous avez réussit à passer une chaîne de caractères en paramètre à votre JSP depuis une servlet. Mais vous devez savoir que vous pouvez aussi passer des objets ! 

Ce que nous allons faire, c'est tout simplement garder notre exemple mais, en plus, passer un objet color et invoquer la méthode `getColor()` dans notre JSP.

Voici le code de notre servlet :

Code : Java

```
package com.servlet.test;

import java.io.IOException;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import com.servlet.test.model.ColorModel;

public class ColorServlet extends HttpServlet {

        public void doGet (HttpServletRequest request, HttpServletResponse response)
                                                throws ServletException, IOException{
                request.setAttribute("colorObject", new ColorModel());
                request.setAttribute("colorString", new ColorModel().getColor());
                request.getRequestDispatcher("colorJsp.jsp").forward(request, response);
        }

}
```

Et le nouveau code de notre JSP :

Code : JSP

```
<%@ page import="com.servlet.test.model.ColorModel" %>
<html>
<body>

<h1 style="color:<%=((ColorModel)request.getAttribute("colorObject")).getColor() %>">
        Récupération de l'objet "<em>ColorModel</em>"
</h1>
<h1 style="color:<%=request.getAttribute("colorString") %>">
        Récupération de l'attribut "<em>colorString</em>"
</h1>
</body>
</html>
```

 **Lorsque vous passez des objets en paramètre, la méthode `getAttribute(String name)` retourne un type `Object` : pensez à caster vos attributs !**

Voici ce que j'ai obtenu :

Screen 1 :

Screen 2 :

Vous l'aurez compris, vous pouvez passer les objets que vous souhaitez à vos JSP. Celles-ci s'occuperont de l'affichage des données reçues du contrôleur via le modèle !

Vous devez trouver ça beaucoup plus plus clair...
Quelle bouffée d'oxygène, ces JSP...

Je ne vous le dirai jamais assez, mais entre chaque chapitre, pratiquez, pratiquez et pratiquez !
C'est comme ceci que les choses rentrent le mieux.

En attendant, je vous attends au prochain chapitre.

Attends une seconde ! Tu ne nous a toujours pas expliqué comment les servlets sont gérées par Tomcat !

Je vois que vous êtes avides de savoir.
J'allais justement y venir, mais avant, vous avez encore une petite chose à voir.

Utiliser des formulaires

Nous avons réussi à utiliser les trois composantes de la plateforme JEE :

- Modèle : objets Java ;
- Vue : pages JSP ;
- Contôleur : nos servlets.

Il reste encore un point à aborder avant de clore cette partie sur les notions de base : **les formulaires !**
Nous allons donc voir comment ceux-ci sont utilisés et comment récupérer des informations saisies dans l'un d'eux.

Rappel

Je pense qu'un petit rappel s'impose, tout du moins sur le fonctionnement de ces chers formulaires !
Tout d'abord, dans des formulaires web, vous pouvez trouver plusieurs sorte de champs.

- Des champs de texte :
  ```
  <input type="text" name="nom" />
  ```
- Des boutons (de type button ou submit):
  ```
  <input type="button" name="nom" value="Valider" />
  ```
- Des champs cachés :
  ```
  <input type="hidden" name="nom" value="je suis caché..." />
  ```
- Des champs de mot de passe :
  ```
  <input type="password" name="nom" value="je suis caché..." />
  ```
- Des cases à cocher :

```
<input type="checkbox" name="nom" value="OUI" /> <input type="checkbox" name="nom2" value="NON" />
```

- Des boutons radio :
```
<input type="radio" name="nom" value="OUI" /> <input type="radio" name="nom" value="NON" />
```
- Des liste simples ou multiples :
```
<select name="nom"> <option value="1">Choisissez</option> </select>
```
- Des zones de texte :
```
<textarea name="nom"> Du texte </textarea>
```

Tous ces champs sont porteurs d'informations qui sont transmises vers une page web via un formulaire.
Un formulaire, en HTML, c'est tout simplement ceci :

Code : HTML

```
<form name="formulaire" action="/traitement" method="post">
<!-- Différents champs de formulaires -->
</form>
```

Ici, nous avons un formulaire :

- qui s'appelle "***formulaire***" ;
- qui envoie les données des champs vers la page "***/traitement***" ;
- et qui envoie ces informations via une requête de type "***post***".

À quoi sert tout ceci ?

Le nom du formulaire sert surtout en Javascript lorsque vous voulez vérifier les données de celui-ci...
Les autres servent pour la communication avec le serveur :

- l'attribut ***action*** **permet au navigateur de savoir vers quelle page envoyer les données** ;
- l'attribut ***method*****, lui, permet de savoir comment on envoie les données.**

Nous aborderons l'attribut ***action*** dans la sous-partie suivante ; pour le moment, nous allons nous atteler à l'attribut ***method***.
Vous savez déjà que, par défaut, les informations entre pages web sont communiquées en **GET** ; eh bien, pour les formulaires, c'est pareil : **par défaut, les informations envoyées via un formulaire sont envoyés en GET !**

J'ai fait un formulaire :

Son code source est le suivant :

Code : JSP

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>

        <form name="firstForm" action="firstFormAction.do"
style="width:50%;margin:auto;background-color:#c1d9fc;padding-bottom:15px;">

                <h2 style="text-align:center;color:white;background-
color:#6683b1;">Formulaire HTML</h2>
                <p style="text-
align:center;">Nom : <input type="text" name="nom" /></p>
                <p style="text-
align:center;">Prénom : <input type="text" name="prenom" /></p>

                <p style="text-align:center;width:50%;margin:auto;">
<input type="submit" name="Valider" value="Valider"/></p>

        </form>

</body>
</html>
```

La page qui reçoit les informations ressemble à ça :

Vous avez saisie :

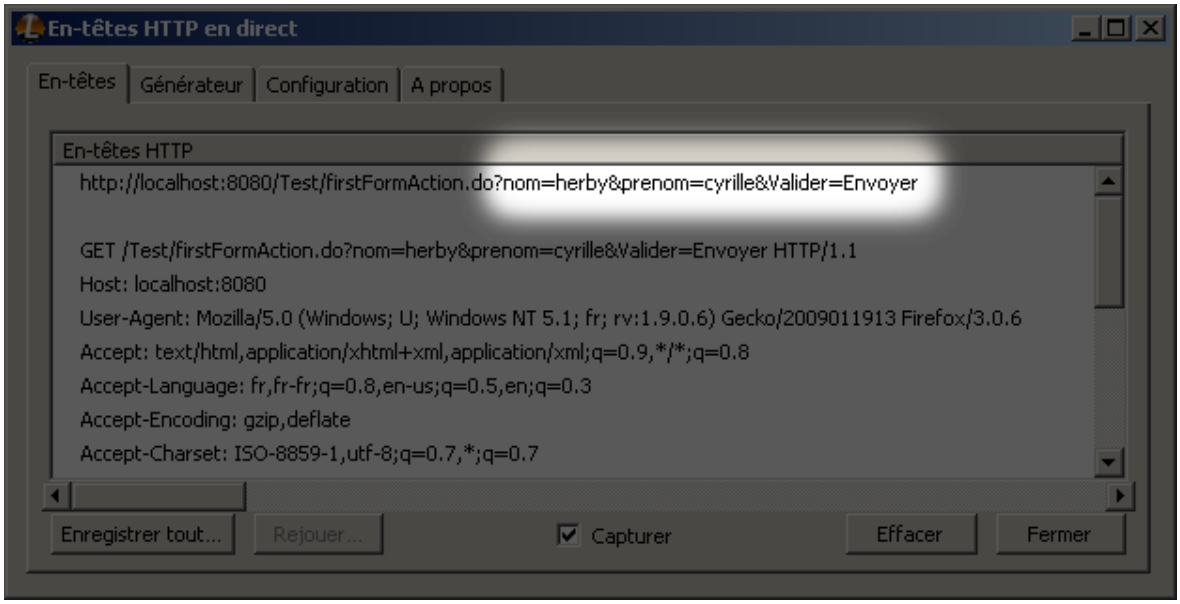
- Nom : HERBY
- Prénom : CYRILLE

Pour retourner au formulaire, c'est par ici

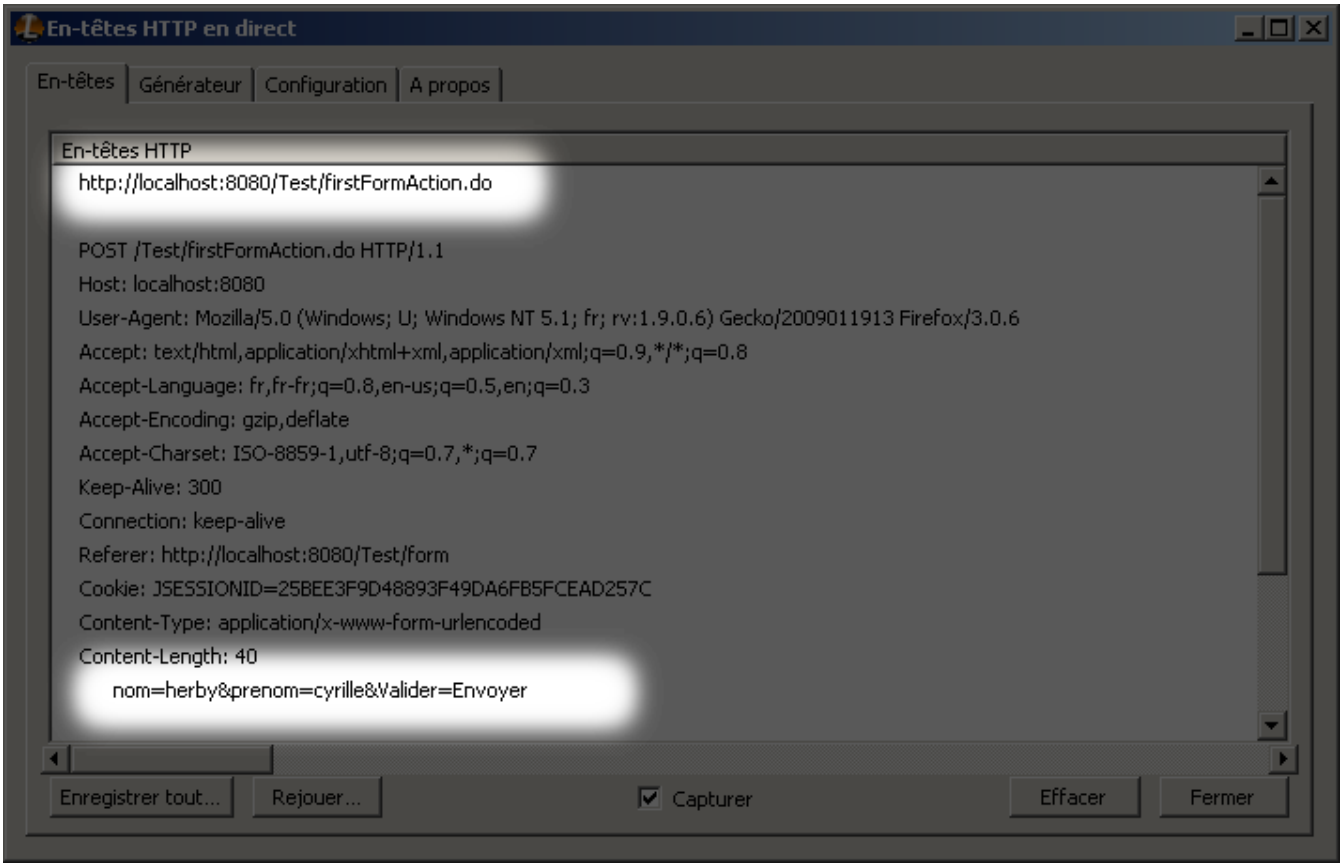
Nous verrons le code source de la page recevant les informations très bientôt... En fait, dans le sous-chapitre qui suit... Mais, pour le moment, le sujet n'est pas là ! 

Ce formulaire n'a pas de méthode de transit d'informations déclarée, donc, les données passent en GET et en voici la preuve :

Et voici les en-têtes HTTP correspondantes à la requête envoyée :



Vous pouvez clairement voir que les données sont directement visibles dans l'URL du navigateur ainsi que dans l'URL de la requête HTTP...
Par contre, voici ce que j'ai obtenu avec le même formulaire ayant l'attribut ***method*** clairement renseigné avec post :



Les données ont bien transité mais elles n'apparaissent plus dans l'URL du navigateur ni dans l'URL de la requête HTTP !

Ceci est la principale différence entre les requêtes HTTP de type GET et celles de type POST, mais il en existe deux autres.
La taille maximale d'une URL est d'environ 2 000 caractères ! Alors qu'elle est quasiment illimitée avec une requête de type POST !
Par contre, avec une requête de type POST, lorsque vous êtes sur la page ayant reçue les informations, si vous rechargez la dite page avec F5 ou CTRL + F5, le navigateur vous demandera si vous souhaitez renvoyer les données ! Ceci peut engendrer des erreurs de taille : dupliquer des données par exemple...

Donc, pour simplifier, si votre formulaire est destiné à faire de l'affichage, vous pouvez utiliser des requêtes de type GET. Par contre, si votre formulaire est utilisé afin de renseigner des champs vitaux pour l'application, mieux vaut utiliser les requêtes de type POST : les utilisateurs ne voient pas le nom de vos champs, ni la valeur de ceux-ci et, en plus, la taille est quasiment illimitée. Mais faites attention à la réexpédition des données !


Et pour l'attribut "***action***" ? Il a aussi une valeur par défaut ?

Oui ! Par défaut, si vous ne renseignez pas cet attribut, les données de cette page seront envoyées à elle-même ! 

Les sources de notre formulaire

Les Zéros les plus hardis auront déjà deviné que j'ai utilisé le couple servlet - JSP pour afficher le formulaire et pour afficher les données de celui-ci.

Voici les codes source de mon couple d'affichage de formulaires :

Secret (cliquez pour afficher)

Et le code source des pages d'affichage des données de formulaire :

Secret (cliquez pour afficher)

Bon, vous avez vu que je n'ai fait que de l'affichage, aucun contrôle dans nos servlets, juste une délégation de l'affichage : donc, vous savez faire.


Euh, attends une seconde, c'est quoi ça : `<form name="firstForm" action="firstFormAction.do"...>` ? Qu'est-ce que c'est que cette extension de fichier ?

Je vois que vous avez un oeil de lynx... Moi qui voulait éluder la question ! 
Vous avez donc aussi remarqué ceci :

Code : XML

```
<servlet>
		<servlet-class>com.servlet.test.FormulaireAction</servlet-
class>
		<servlet-name>FormAction</servlet-name>
	</servlet>

	...

	<servlet-mapping>
		<servlet-name>FormAction</servlet-name>
		<url-pattern>*.do</url-pattern>
	</servlet-mapping>
</web-app>
```

En fait, ce genre de nom fait aussi partie de la convention de nommage JEE.
On appelle ce genre de noms **des noms logiques**.
Comme la plupart des liens que vous trouverez dans les codes source de vos JSP, ce sont des noms totalement fictifs. Ils sont mappés au même titre que les autres type de liens, **dans le fichier web.xml !**
La convention Java nous dit que pour les pages web faisant des traitements, il est préférable d'utiliser des noms portant l'extension "**.do**" : comme ça, un développeur passant par là saura de suite qu'il s'agit d'une page de traitement !
Ensuite, vous avez sans doute remarqué que le mappage dans le fichier web.xml était un peu particulier pour le nom logique.

C'est juste que là, le mappage dit que la servlet ayant pour nom "***FormAction***" est chargée de traiter **TOUTES** les requêtes ayant un nom se terminant par "**.do**"

Sinon, rien de sorcier, mis à part le fait que les données que le serveur reçoit sont dans l'objet `HttpServletRequest`.
Ce que j'ai fait aussi. Dans la servlet qui se charge de récupérer les données du formulaire, j'ai redéfini deux méthodes :

- doGet(HttpServletRequest request, HttpServletResponse response) ;
- doPost(HttpServletRequest request, HttpServletResponse response).

La méthode `doGet()` est définie pour déléguer l'affichage à une JSP et la méthode `doPost()` invoque la méthode `doGet()` . Comme ça, la boucle est bouclée... 

Ensuite, dans notre page JSP, nous affichons les données reçues grâce à l'objet `request` (objet faisant partie des objets n'ayant pas besoin d'import dans les JSP... Idem pour `response`)...
Nous utilisons la méthode `<p>Nom : <%= request.getParameter("nom")%></p>` afin de récupérer la donnée.

Rajoutons des champs

Ce que je vous propose de voir maintenant, c'est ce que nous retournent les différents types de champs de formulaire HTML.
Pour ce faire, nous allons étoffer un peu notre formulaire. Voici le nouveau code source du formulaire de tout à l'heure :

Code : JSP

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4
/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>

        <form name="firstForm" action="firstFormAction.do" method="post"
             style="width:50%;margin:auto;background-color:#c1d9fc;padding-bottom:15px;">

                <div style="text-align:center;">
                        <h2 style="color:white;background-color:#6683b1;">Formulaire HTML</
                        <p>Nom : <input type="text" name="nom" /></p>
                        <p>Prenom : <input type="text" name="prenom" /></p>
                        <p>Sexe : <input type="radio" name="sexe" value="Masculin" />Mascul
                                  <input type="radio" name="sexe" value="Féminin" />Féminir
                        <p>Couleur de vos yeux :
                                <select name="yeux">
                                        <option value="Bleu">Bleu</option>
                                        <option value="Marron">Marron</option>
                                        <option value="Vert">Vert</option>
                                </select>
                        </p>
                        <p>Vous programmez en :<br >
                                <input type="checkbox" name="C" />En C
                                <input type="checkbox" name="C++" />En C++
                                <input type="checkbox" name="Java" />En Java
                                <input type="checkbox" name=".NET" />En .NET
                                <input type="checkbox" name="PHP" />En PHP
                        </p>
                        <p style="width:50%;margin:auto;">
<input type="submit" name="Valider" value="Valider"/></p>
                </div>
        </form>

</body>
</html>
```

Nous allons également modifier l'affichage des données. Pour ne pas trop alourdir le tout, nous nous contenterons d'afficher les données du formulaire via notre servlet :

Code : Java

```java
package com.servlet.test;

import java.io.IOException;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.Enumeration;
import java.util.List;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class FormulaireAction extends HttpServlet {

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

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

		//On récupère la liste des noms de paramètres
		Enumeration<String> e = request.getParameterNames();
		List<String> prog = new ArrayList<String>();

		//On parcours cette liste
		while(e.hasMoreElements()){
			String key = e.nextElement();

			//On vérifie les valeurs des checkbox : 'on' signifie
que la checkbox est cochée
			if(request.getParameter(key).equals("on")){
				prog.add(key);
			}
			else{
				out.println("<p><strong>" + key + " :
</strong>" + request.getParameter(key) + "</p>");
			}
		}

		//Si nous avons au moins un langage de programmation
		if(prog.size() > 0)
			out.println("<p><strong>Je programme en :
</strong>");

		out.println("<ul>");

		for(String str : prog)
			out.println("<li>" + str + "</li>");

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

		//On n'utilise plus notre JSP...
		//request.getRequestDispatcher("formResult.jsp").forward(request,
response);
	}

	public void doPost(	HttpServletRequest request,
				HttpServletResponse response)
				throws IOException, ServletException{
		//Nous invoquons la méthode doGet avec les paramètres reçu
par la méthode doPost
		doGet(request, response);
	}
}
```

Et le résultat, sous vos yeux ébahis :

Valider : Envoyer

nom : herby

sexe : Masculin

yeux : Marron

prenom : cyrille

Je programme en :

- Java
- PHP

Alors, vous avez vu que nous pouvions récupérer les champs de nos formulaires dans nos servlets ainsi que dans nos JSP via l'objet `HttpServletRequest`.
Vous aurez sans doute remarqué que tous les champs ne retournent pas leurs valeurs...
Par exemple, les champs de type texte, les radio, les listes et les bouton retournent leurs valeurs ; par contre, les cases à cocher ne retournent rien si elles ne sont pas cochées et "**on**" dans le cas contraire !


On a cru remarquer que tu as utilisé des liens HTML dans tes JSP.
Ils ne sont pas un peu bizarres ?

Vous avez réussi à voir ça ? Vous êtes en forme, aujourd'hui ! 
En effet, je me doutais que ceci allait piquer votre curiosité...
J'ai bel et bien un lien sur ma page qui a l'air de pointer vers un endroit non mappé dans mon fichier web.xml !

Celui-ci : `<a href="form">par ici</a>` qui serait bien lié à cela : `<url-pattern>/form</url-pattern>` , mais un truc cloche !
Nous allons voir ça tout de suite...

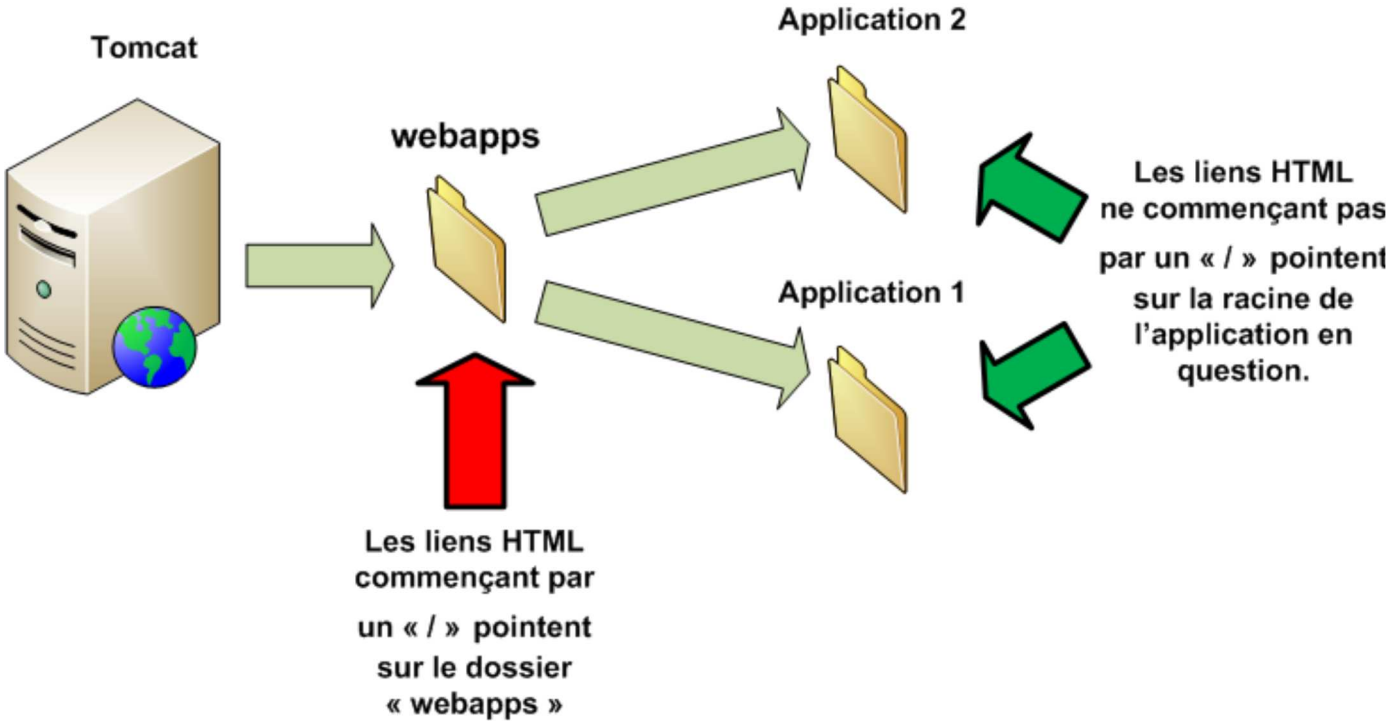
Tout est lié

En fait, dites-vous bien qu'il y a deux endroits distincts pour accéder à des pages de notre application :

- depuis le serveur ;
- depuis votre navigateur.

La différence entre ces deux acteurs est la suivante : ils n'interprètent pas les liens commençant par un "**/**" de la même manière !

Pour le serveur, lorsque vous êtes dans une application, le "**/**" signifie **la racine de l'application** tandis que pour le navigateur, cela signifie **la racine du serveur** !



Donc, pour faire un lien HTML pour faire en sorte que notre application puisse utiliser le mappage `<url-pattern>/form</url-pattern>` nous pouvons faire :

- soit ça : `<a href="form">par ici</a>` **aussi appelé chemin relatif (car relatif à l'application)** ;
- soit ça :`<a href="/Test/form">par ici</a>` **aussi appelé chemin absolu (car relatif au serveur)**.


Il va de soit que "Test" est le nom de mon dossier contenant mon application.


Lequel utiliser alors ?

Réfléchissez et regardez comment on utilise nos liens depuis le début pour accéder à nos pages.
Que sert à faire le fichier **web.xml** ?
Rappelez-vous que, moins les utilisateurs en savent sur votre application, plus elle sera sûre !


Nous devrions donc préférer les liens relatifs ?

Tout à fait !

Je crois que vous en avez assez vu pour le moment...

Bon, ne me dites pas que ce chapitre était compliqué, je ne vous croirais pas...
Donc, tout ceux qui n'ont pas 100 % au QCM, je les flagelle ! 

Chapitre très simple, en tous cas, pour ceux qui connaissaient déjà le fonctionnement des formulaires web.
Nous avons maintenant fait le tour des notions de base nécessaires au bon déroulement du tuto.

Cette partie est donc terminée, mais avant de réellement clore cette dernière, un TP vous attend ! 

TP : la loterieZ

Vous voilà rendus au premier TP de ce tuto.

Il est assez simple, je vous rassure. Par contre, il mettra en oeuvre tout ce que vous aurez vu jusqu'ici, alors accrochez-vous bien !

Cahier des charges

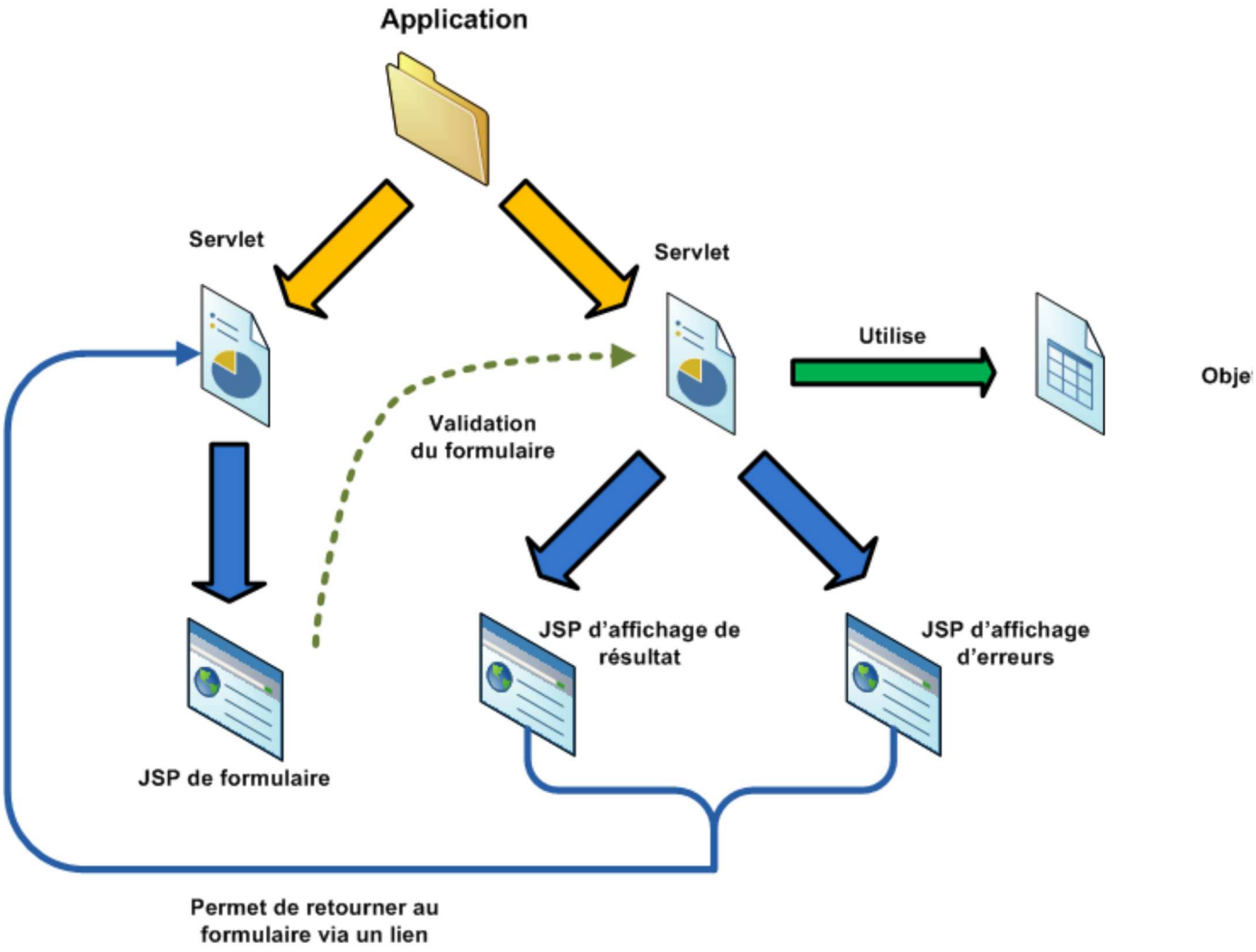
Alors... La loterieZ est une loterie sous forme de page web que vous devrez coder !

Celle-ci vous demande de choisir **deux chiffres DIFFÉRENTS** dans deux listes de 1 à 10. Une fois ceci fait, vous devrez envoyer les informations au serveur (ça sent bon les formulaires...).

Une fois les informations reçues, nous aurons un objet `LoterieZ` qui se charge de faire le tirage des numéros, de voir si vous avez gagné (ou perdu) et qui retourne aussi les numéros de ce tirage !

Par contre, vous devrez gérer un message d'erreur si vous avez choisi deux fois le même numéro.
Il serait de bon ton que vous leviez une exception de votre cru lors du contrôle du tirage. Dans ce cas, vous utiliserez une page JSP différente de celle à utiliser pour l'affichage normal.

Voici un petit schéma :

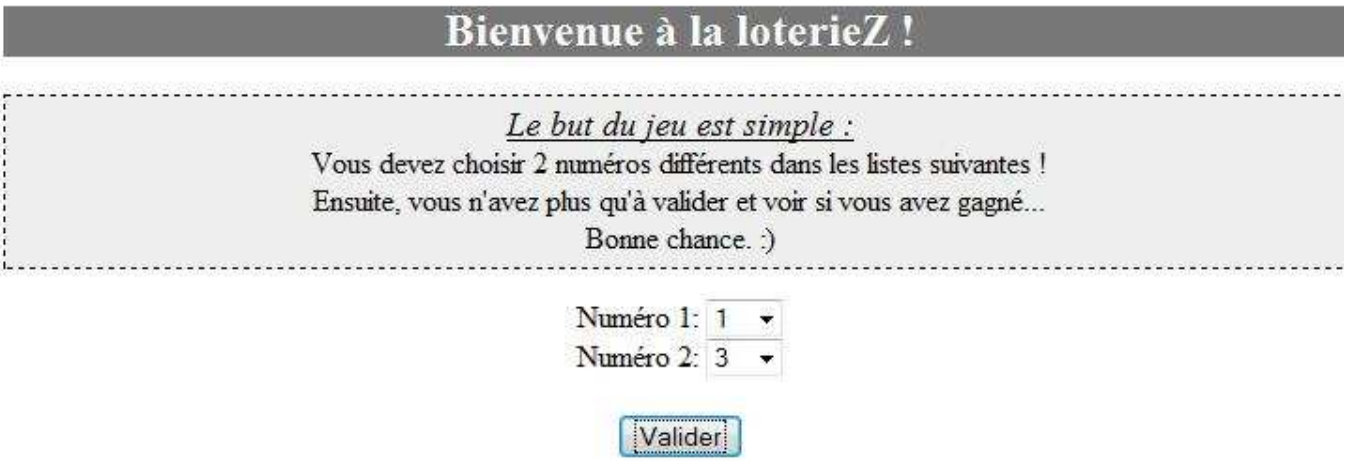


Vous avez carte blanche sur la façon de gérer tout ça... Mais je vais tout de même vous fournir des copies d'écran... Histoire de vous diriger un peu.

Copies d'écran

Voici quelques copies d'écran de ce que j'ai pu obtenir.

Le formulaire



L'affichage en cas de victoire

Vous aviez joué : 4 - 0
Félicitation, vous avez gagné ! ! !

Vous pouvez tenter à nouveau votre chance en suivant ce lien

L'affichage en cas de défaite

Vous aviez joué : 1 - 2
Quel dommage, vous avez perdu...

Vous pouvez tenter à nouveau votre chance en suivant ce lien

La page d'erreur

Une erreur est survenue :
Vous avez choisi plusieurs fois le même numéro ! !

Vous pouvez tenter à nouveau votre chance en suivant ce lien

Vu que je suis un véritable amour sur pattes, je vous fournis aussi un screenshot du contenu de ma classe `LoterieZ` :

Voilà, les dernières recommandations ont été données, je n'ai plus qu'à vous souhaiter bonne chance et bon courage !
Allez ! **GGGGOOOOOO !**

Correction

Je me suis longtemps demandé si je devais vous fournir une correction ou si je devais vous laisser chercher...
Nan... Je plaisante.

Vous avez bien cherché, j'espère. Vous ne vous ruez pas sur la correction comme ça...
Bon, voici les sources de mon projet :

Secret (cliquez pour afficher)

Voilà, vous avez toutes les informations pour comprendre ce code.
Si vous avez du mal, je suis là, mais essayez d'abord de comprendre par vous-mêmes, en relisant les chapitres par exemple.
Car c'est comme ça qu'on retient le mieux les choses !

Pour les Zéros qui le souhaitent, je mets le contenu de mon application dans une archive.
Vous pouvez la télécharger ici !
Il ne vous restera plus qu'à décompresser celle-ci dans le dossier "**webapps**" de Tomcat et à redémarrer celui-ci.

Vous avez aimé ce TP ?
Quoi ? On a perdu des Zéros dans la bataille ?

Ouf ! C'était une fausse alerte...

Bon, prenez bien le temps de digérer cette partie car la suite promet d'être un chouilla plus compliquée...

Bon, nous avons réussi tant bien que mal à poser les bases de la plateforme JEE.

Vous avez appris à créer un projet, faire des servlets, combiner ces dernières avec des JSP tout en utilisant des objets métiers !
Maintenant, vu que vous vous êtes sûrement posé beaucoup de questions sur le fonctionnement de tout ceci, le moment est venu d'apporter quelques éléments de réponses...

Partie 2 : Ce que sait faire le conteneur

Dans la partie précédente, nous avons fait un tour rapide de la plateforme JEE.

Ceci dans le sens où nous avons créé une servlet liée à une JSP tout en utilisant un objet métier.

Par contre, bon nombre de points doivent vous sembler obscurs.

Ce que je vous propose dans cette partie n'est rien d'autre que de faire la lumière sur ce qu'il se passe dans notre conteneur.

 Vous vous sentez prêts ?

Alors go ! 

Paramètres de servlets

Nous allons aborder un aspect bien pratique de nos chères servlets : les paramètres !
Nous allons voir que ceux-ci peuvent nous sauver la mise dans certains cas et nous évitent bien des tracas...

Paramètres d'initialisation

Imaginez que vous souhaitez attribuer un titre à votre application et que celui-ci soit lié à vos servlets.
Pour éclaircir la chose, vous savez qu'une page JSP est appelée via une servlet :

 Comment feriez-vous pour affecter un titre différent à votre JSP selon la servlet qui appellera cette JSP ?

Je suppose que la première chose à laquelle vous avez pensé était de mettre ce titre "***en dur***" dans le code de votre servlet...
C'est-à-dire que ce nom est défini et attribué à votre page JSP dans une servlet ! Un peu comme ceci :

Code : Java

```
request.setAttribute("titre", "Titre de ma page");
```

Ou encore de définir une variable de classe initialisée avec le titre...
Ces méthodes fonctionnent très bien, je vous rassure ! Par contre, il y a un autre moyen de faire ceci : en utilisant des **paramètres d'initialisation de servlets**.

 Les quoi ? 

Les paramètres d'initialisation.
C'est très simple. Lorsque votre servlet vient à la vie, vous pouvez lui dire qu'elle a, à sa disposition, différents paramètres que vous lui aurez attribués !
Afin de vous montrer que nous pouvons initialiser plusieurs paramètres, je ne vais pas en créer un, mais deux ! 

 Et comment tu fais ça ?

Ceci se fait grâce à notre bon vieux fichier **web.xml**. Par exemple, pour affecter l'attribut "**nomPage**" ayant pour valeur "**Nom de ma page**" et un autre ayant pour nom "**sousTitre**" et comme valeur "**Sous titre de la page**"à une servlet, vous n'avez qu'à faire ceci :

Code : XML

```
<web-app>

	<servlet>
		<servlet-class>com.sdz.control.Index</servlet-class>
		<servlet-name>StartPage</servlet-name>

		<init-param>
			<param-name>titrePage</param-name>
			<param-value>Nom de la page</param-value>
		</init-param>

		<init-param>
			<param-name>sousTitre</param-name>
			<param-value>Sous titre de la page</param-value>
		</init-param>

	</servlet>

	<servlet-mapping>
		<servlet-name>StartPage</servlet-name>
		<url-pattern>/</url-pattern>
	</servlet-mapping>

</web-app>
```

C'est simple et clair ! La balise XML **<servlet></servlet>** encapsule autant de balises **<init-param></init-param>** qu'elle le souhaite (enfin, que vous le souhaitez...).
Comprenez bien que ces paramètres ne seront utilisables uniquement par la servlet concernée : ici, `com.sdz.control.Index` !

Et comment on les utilise ?

Voici la servlet liée au fichier **web.xml** sus-mentionné et qui utilise les paramètres :

Code : Java

```java
package com.sdz.control;

import java.io.IOException;
import java.io.PrintWriter;
import java.util.Enumeration;

import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class Index extends HttpServlet {

	public void doGet(	HttpServletRequest request,
						HttpServletResponse response)
						throws IOException, ServletException{
		//Bon, là, c'est simple tout de même...
		response.setContentType("text/html");
		PrintWriter out = response.getWriter();

		//La servlet a une méthode de récupération des noms de ces
paramètres
		Enumeration e = getInitParameterNames();

		//On récupère un objet de configuration de servlets
		ServletConfig conf = getServletConfig();

		//Et on parcourt le tout
		while(e.hasMoreElements()){
			String name = e.nextElement().toString();
			//On appelle la méthode getInitParameter(String name)
			//afin de récupérer la valeur
			out.println("<p><strong>" + name + " =
" + conf.getInitParameter(name) + "</strong></p>");
		}
		//Vous pouviez aussi y accéder comme ceci
		//out.println(getServletConfig().getInitParameter("titrePage"));
		//out.println(getServletConfig().getInitParameter("sousTitre"));

	}
}
```

Ce qui nous donne :

sousTitre = Sous titre de la page

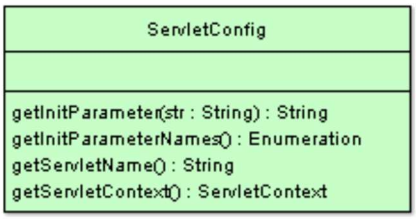
titrePage = Nom de la page

C'est très simple d'utilisation et très utile !
Bon, nous avons utilisé une façon de faire un peu plus générique puisque nous avons récupéré tous les noms de paramètres afin de tous les afficher...
Nous avons récupéré la liste des noms de paramètres grâce à l'instruction `Enumeration e = getInitParameterNames();` puis nous avons affiché le contenu des différents paramètres en les invoquant via cette instruction : `getInitParameter(name);` de l'objet `ServletConfig`.

C'est sûr, mais qu'est-ce que c'est que cet objet : `ServletConfig` ?

Dites-vous que votre conteneur, Tomcat, est très sympa et qu'au moment de créer votre servlet, il crée un objet contenant tous les paramètres d'initialisation !

Et cet objet est le suivant :



Cet objet n'a pas beaucoup de méthodes, mais vous pouvez deviner ce que celles-ci font...

Dis donc, la méthode `getInitParameterNames()` , tu n'as pas utilisé cet objet pour l'invoquer tout à l'heure ?

Votre sens de l'observation s'est grandement amélioré depuis le début de ce tuto, ou c'est moi...
En fait, dans nos servlets, il y a une méthode qui s'appelle ainsi et qui a le même rôle que la méthode de l'objet en question, et pour cause : la méthode présente dans notre servlet invoque la méthode de l'objet `ServletConfig`...

Par contre, que fait la méthode `getServletContext()` ?

Nous allons aborder ceci tout de suite...
Par contre, même si vous êtes d'accord sur l'utilité de ces paramètres d'initialisation, imaginez un instant le cas contraire à ce que je vous demandais plus haut :
imaginez que vous ayez besoin d'un titre d'application commun à chaque page !

Comment procéderiez-vous ? Où mettre ce titre ? Dans toutes nos servlets ? Dans toutes nos JSP ?
Je vous laisse réfléchir sur la marche à suivre quelques instants avant de vous donner une réponse, mais ceci a un rapport avec le fameux objet `ServletContext`.

Tout dépend du contexte...

Voilà un mot qui vous avait chatouillés lors d'un précédent chapitre...
En fait, c'était lorsque je vous avais expliqué à quoi servait le fichier **web.xml**.

Comme je vous le disais alors, **ce fichier permet de définir le contexte de l'application !**
Voyez un peu ça comme l'environnement de l'application : ce dont l'application a besoin pour travailler.
Vous savez déjà que c'est grâce à ce contexte que le conteneur sait faire fonctionner l'application, mais ce que vous ignorez, c'est que vous pouvez aussi définir des paramètres à ce contexte.

Ah oui ?

En plus, ça se fait quasiment comme la déclaration de paramètres pour une servlet, à quelques différences près.
Vous l'aurez deviné, ça se passe encore dans le fichier **web.xml**...

Mais concrètement, en quoi cela va répondre à notre besoin concernant un titre commun à chaque page ?

Hi hi !
C'est là que la magie s'opère : **les paramètres de contexte sont globaux à toute l'application !**
Comprenez ceci dans le sens où tous ces paramètres seront accessibles à toutes nos servlets !

Tout comme les paramètres d'initialisation de servlets, le conteneur va créer un objet, mais cette fois, il s'agira d'un objet `ServletContext` qui sera partagé par toutes nos servlets !

La méthode `getServletContext()` de l'objet `ServletConfig` nous retourne donc cet objet ?

Tout à fait ! Mais comme les concepteurs de la technologie JEE sont des gens super gentils, ils ont fait en sorte que nos servlets puissent invoquer une méthode `getServletContext()` sans passer par l'objet `ServletConfig`.

Voyez ça comme un raccourci au même titre que la méthode `getServletConfig()` !

Bon, assez de blabla, voici comment on déclare des paramètres pour toute notre application :

Code : XML

```
<web-app>

	<servlet>
		<servlet-class>com.sdz.control.Index</servlet-class>
		<servlet-name>StartPage</servlet-name>

		<init-param>
			<param-name>titrePage</param-name>
			<param-value>Nom de la page</param-value>
		</init-param>

		<init-param>
			<param-name>sousTitre</param-name>
			<param-value>Sous titre de la page</param-value>
		</init-param>

	</servlet>

	<context-param>
		<param-name>contextParam</param-name>
		<param-value>Paramètre global à l'application !</param-value>
	</context-param>

	<context-param>
		<param-name>contextParam2</param-name>
		<param-
value>Paramètre global à l'application ! (numéro 2)</param-value>
	</context-param>

	<servlet-mapping>
		<servlet-name>StartPage</servlet-name>
		<url-pattern>/</url-pattern>
	</servlet-mapping>

</web-app>
```

Voici une servlet qui utilise les paramètres d'initialisation et des paramètres de l'application (vous pourrez comparer l'utilisation des deux types de paramètres) :

Code : Java

```java
package com.sdz.control;

import java.io.IOException;
import java.io.PrintWriter;
import java.util.Enumeration;

import javax.servlet.ServletConfig;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class Index extends HttpServlet {

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

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

		Enumeration e = getInitParameterNames();
		ServletConfig conf = getServletConfig();

		while(e.hasMoreElements()){
			String name = e.nextElement().toString();
			out.println("<p><strong>" + name + " =
" + conf.getInitParameter(name) + "</strong></p>");
		}

		out.println("<h1>-------------------------------</h1>");

		//Ici, on récupère le contexte de l'application
		ServletContext context = getServletContext();
		//De la même manière que ci-dessus, nous récupérons la liste
de paramètres
		e = context.getInitParameterNames();

		//Et on parcourt le tout
		while(e.hasMoreElements()){
			String name = e.nextElement().toString();
			//On appelle la méthode getInitParameter(String name)
			//afin de récupérer la valeur
			out.println("<p><strong>" + name + " =
" + context.getInitParameter(name) + "</strong></p>");
		}

	}
}
```

Et le résultat, sous vos yeux pétillants  :

sousTitre = Sous titre de la page

titrePage = Nom de la page

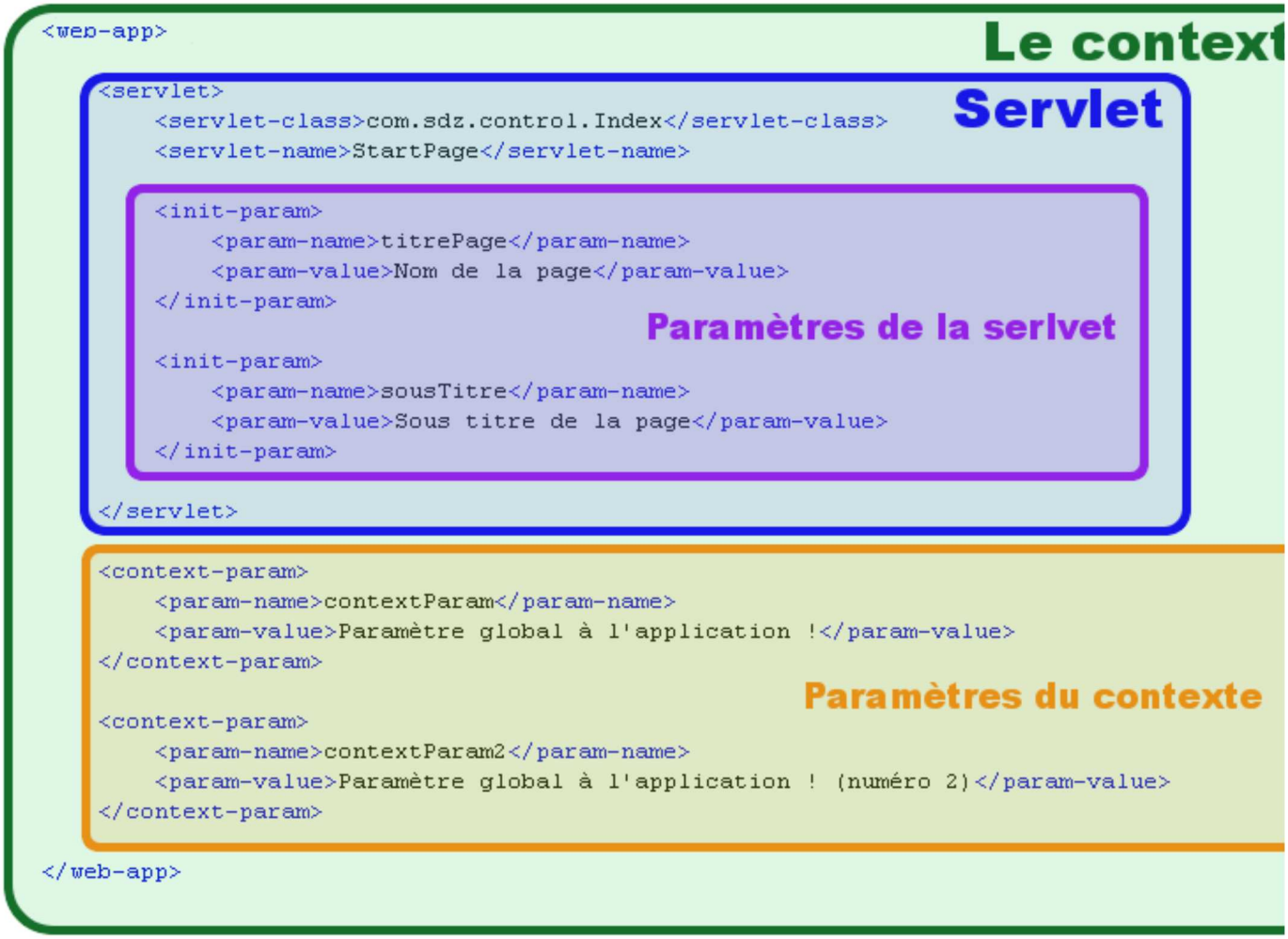
---

contextParam = Paramètre global à l'application !

contextParam2 = Paramètre global à l'application ! (numéro 2)

Et là, vous pourrez faire autant de servlets que vous le souhaitez, ces paramètres seront accessibles à toutes !

Vous avez vu que, mis à part le nom de la balise, l'endroit où se trouvait la balise a son importance :



Vous voyez bien que les paramètres de la servlet se trouvent dans la balise **<servlet></servlet>** alors que les paramètres de l'application se trouvent dans la balise **<web-app></web-app>** .

C'est vrai que l'utilisation est très ressemblante aux paramètres d'initialisation de servlets...

Oui, là où la plateforme est bien faite, c'est que, mis à part le type d'objet utilisé, la méthode invoquée est la même :

Code : Java

```
//Pour récupérer un paramètre de la servlet
getServletConfig().getInitParameter("nomParamter");
//Pour récupérer un paramètre de l'application
getServletContext().getInitParameter("nomParamter");
```

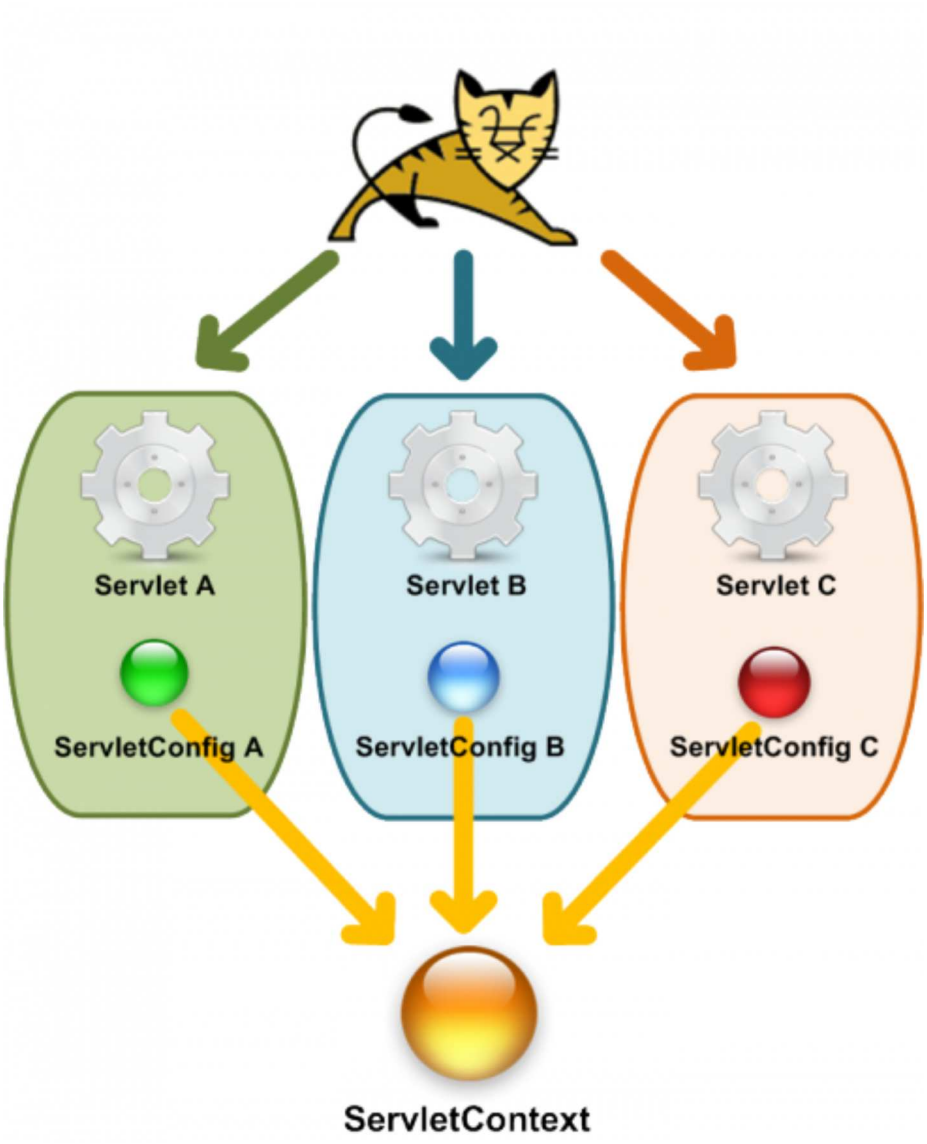
Par contre, ne confondez surtout pas les deux ! Il y a un objet `ServletConfig` par servlet et un objet `ServletContext` par application !

Pourtant l'objet `ServletConfig` nous retourne un objet `ServletContext` via la méthode `getServletContext()` ?

Oui, bien sûr ! Mais il s'agit d'un seul et même objet !

Dans le chapitre suivant nous approfondirons ce point, mais pour le moment, sachez qu'il existe **un et un seul objet** `ServletContext` dans toute l'application alors qu'il y a un objet `ServletConfig` **par servlet** !

Voici un petit schéma afin d'illustrer tout ça :



Un autre point important et que vous aurez sans doute remarqué : **qu'il s'agisse de paramètres de servlet ou de paramètres de contexte, on ne peut utiliser que des `String` comme valeurs de nos paramètres !**

Tu veux dire que nous ne pouvons pas utiliser d'objets ?

Si, mais par un moyen détourné. En fait, nos servlets peuvent créer des paramètres prenant des objets comme valeurs ! Voyons ça tout de suite... 

Des objets en paramètres

Alors, déjà, au cas où ne l'auriez pas remarqué, l'objet `ServletConfig` n'a pas de méthode permettant de définir des paramètres ; d'ailleurs, l'objet `ServletContext` non plus.

Qu'est-ce que tu nous chantes ?

Je vous dis simplement que nous n'allons pas définir des paramètres d'initialisation, mais des attributs !
Et, en fait, ceci est logique puisque, dans nos servlets, la phase d'initialisation est terminée depuis longtemps.
Nous verrons ceci en détail dans le prochain chapitre.

Du coup, comment fait-on pour définir des objets en attribut de l'objet `ServletContext` ?

De la même façon que vous avez procédé avec l'objet `HttpServletRequest` : avec la méthode `setAttribute(String name, Object value)` .

La différence de taille ici, c'est que, vu que nous travaillons avec l'objet `ServetContext`, ces attributs seront accessibles depuis une autre servlet que celle les ayant définis.

Afin d'illustrer mes dires, je vous ai concocté un petit exemple. Nous allons définir trois attributs à notre objet `ServletContext` dans une servlet :

- un objet `Date` ;
- deux objets `ZColor` (objet créé par mes soins).

Voici le fichier ZColor.java

Code : Java

```
package com.sdz.model;

public class ZColor {

	private String color = "";
	public ZColor(String color){
		this.color = color;
	}

	public String toString(){
		return "COULEUR -> " + this.color;
	}
}
```

Nous utilisons donc ces objets dans une servlet de base dont voici le code.

Le fichier Index.java

Code : Java

```java
package com.sdz.control;

import java.io.IOException;
import java.io.PrintWriter;
import java.util.Enumeration;

import javax.servlet.ServletConfig;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;


public class Index extends HttpServlet {

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


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

                //Ici, on récupère le contexte de l'application
                ServletContext context = getServletContext();

                //On ne définit pas de paramètres d'initialisation mais des
attributs !
                //Ceci en spécifiant un couple clé -> valeur
                context.setAttribute("obj", new java.util.Date());
                context.setAttribute("obj2", new com.sdz.model.ZColor("Rouge"));
                context.setAttribute("obj3", new com.sdz.model.ZColor("Jaune"));

                out.println("<h2>obj :
" + context.getAttribute("obj") + "</h2>");
                out.println("<h2>obj2 :
" + context.getAttribute("obj2") + "</h2>");
                out.println("<h2>obj3 :
" + context.getAttribute("obj3") + "</h2>");
                out.println("<a href=\"context\">Lien vers une autre
page</a>");

        }
}
```

La servlet ci-dessus va affecter et afficher les attributs que nous passons à l'objet `ServletContext`.
En plus de ça, la servlet affiche un lien qui pointe vers une autre servlet, que voici :

Fichier TestServlet.java

Code : Java

```java
package com.sdz.control;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class TestContext extends HttpServlet{

	public void doGet(	HttpServletRequest request,
				HttpServletResponse response)
				throws IOException, ServletException{
		ServletContext context = getServletContext();
		PrintWriter out = response.getWriter();

		response.setContentType("text/html");

		out.println("<h2>obj :
" + context.getAttribute("obj") + "</h2>");
		out.println("<h2>obj2 :
" + context.getAttribute("obj2") + "</h2>");
		out.println("<h2>obj3 :
" + context.getAttribute("obj3") + "</h2>");
	}

}
```

Le tout mappé dans le fichier web.xml

Code : XML

```xml
<web-app>

	<servlet>
		<servlet-class>com.sdz.control.Index</servlet-class>
		<servlet-name>StartPage</servlet-name>
	</servlet>

	<servlet>
		<servlet-class>com.sdz.control.TestContext</servlet-class>
		<servlet-name>TestContext</servlet-name>
	</servlet>

	<servlet-mapping>
		<servlet-name>StartPage</servlet-name>
		<url-pattern>/</url-pattern>
	</servlet-mapping>

	<servlet-mapping>
		<servlet-name>TestContext</servlet-name>
		<url-pattern>/context</url-pattern>
	</servlet-mapping>

</web-app>
```

Le résultat de tout ceci est le suivant :

Arrivés sur la page d'accueil, nous spécifions des attributs à notre contexte et nous les affichons :

obj : Wed Mar 11 16:38:46 CET 2009

obj2 : COULEUR -> Rouge

obj3 : COULEUR -> Jaune

Lien vers une autre page

Le lien de la page précédente nous envoie sur cette page :

obj : Wed Mar 11 16:38:46 CET 2009

obj2 : COULEUR -> Rouge

obj3 : COULEUR -> Jaune

Vous avez donc la preuve que les objets passés à notre contexte sont toujours accessibles via une autre servlet !

Bon, je crois que vous en avez assez vu pour aujourd'hui. Rendez-vous au QCM ! 

Pour commencer cette partie 2, ce n'était pas un chapitre très copieux et très difficile à assimiler.
Par contre, les choses vont se compliquer avec le chapitre suivant...

Nous avons vu comment utiliser nos servlets, comment spécifier des paramètres à celles-ci mais nous ne savons toujours rien sur ce qu'il se passe dans notre cher conteneur. 

Le chapitre suivant mettra quelques points au clair.

Cycle de vie d'une servlet

Vous avez appris comment les applications JEE sont construites : servlet + JSP + objets Java !

Dans ce chapitre vous apprendrez comment une servlet vient à la vie !
Comment Tomcat crée, utilise, initialise et invoque nos servlets.

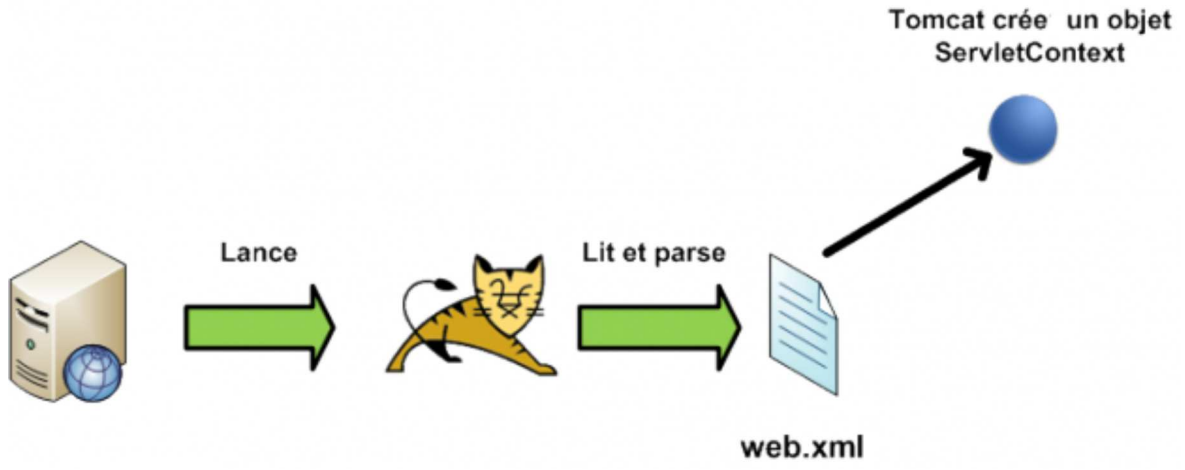
Vous allez voir que ce dernier est vraiment très fort et qu'il nous soulage de beaucoup de travail.

Initialisation de la servlet

Alors ! Accrochez-vous bien car il y aura beaucoup de choses à apprendre et à assimiler !
Déjà, vous aurez sans doute remarqué que **nos servlets ne possèdent pas de constructeur.**


Donc, comment sont-elles construites ?

Et je vous réponds : **oui, nos servlets ont toutes un constructeur !** Mais il est invoqué automatiquement par le conteneur !
En gros, le conteneur, au moment de son lancement, va initialiser toutes les servlets en invoquant leurs constructeurs. Mais avant toute chose, le conteneur va aller lire le fichier **web.xml**. Les choses se passent un peu comme ça :



Voilà ; dans un premier temps, le conteneur construit un objet `ServletContext` contenant tous les couples "**clé - valeur**" spécifiés dans le fichier **web.xml** correspondant aux paramètres d'initialisation de contexte.

Ensuite, le conteneur continue son investigation et va construire puis initialiser les servlets présentes. Lors de l'instanciation d'une servlet par le conteneur, il se passe beaucoup de choses que vous ne soupçonnez même pas...
Voyons un peu.
En fait, lors de l'appel au constructeur de l'objet, le conteneur va aussi créer un objet `ServletConfig` contenant les paramètres d'initialisation de la servlet (couple clé - valeur). Cet objet contiendra aussi l'objet `ServletContext`.

Vous savez déjà que ceci :

Code : Java

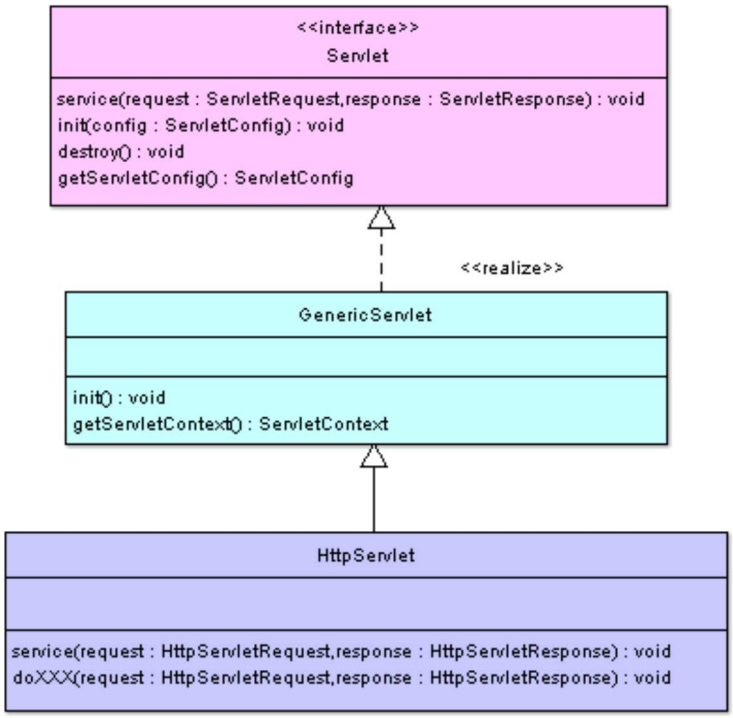
```
ServletContext context = getServletContext();
```

est équivalent à ceci :

Code : Java

```
ServletContext context = getServletConfig().getServletContext();
```

Mais ce n'est réellement utile que si votre servlet n'hérite pas de `HttpServlet` ou de `GenericServlet`...
En effet, la méthode que vous avez utilisée plus haut provient de la classe `GenericServlet` dont hérite la classe `HttpServlet` ! 
Voici un petit diagramme de classe comme vous les aimez :



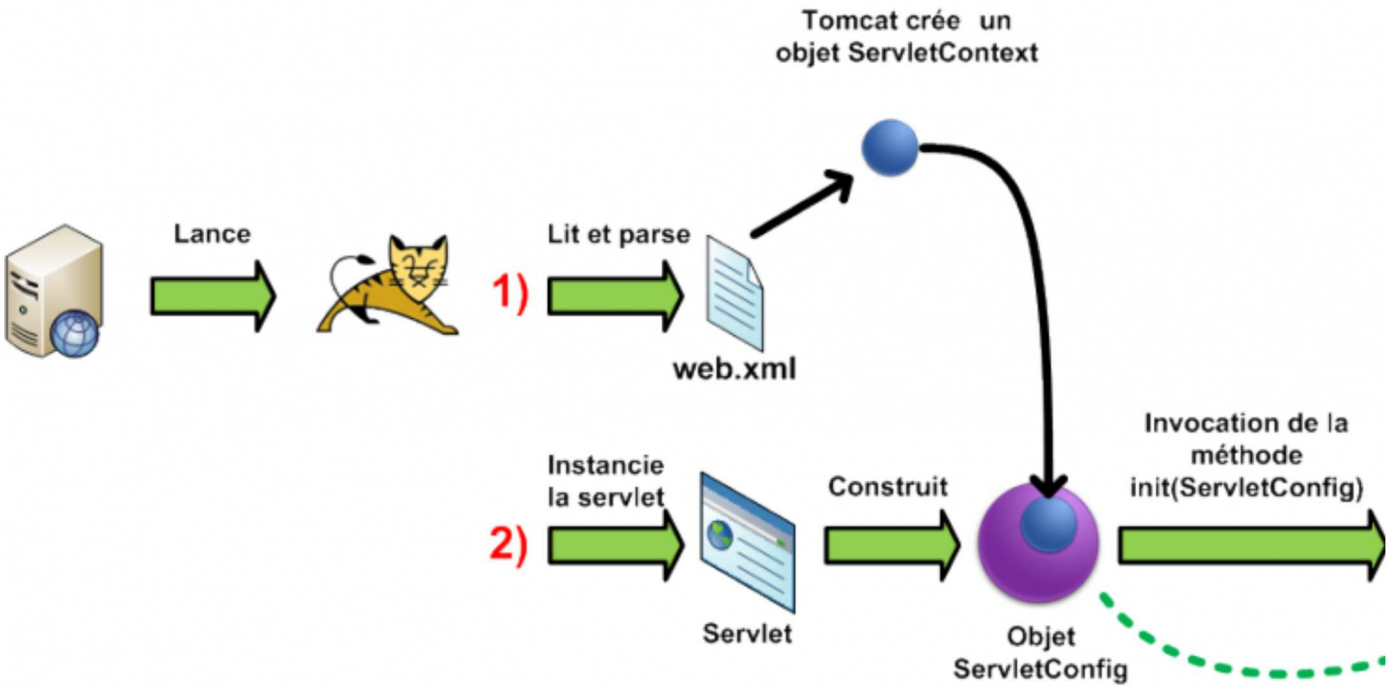
Vous pouvez voir que ladite méthode n'est présente qu'à partir du deuxième niveau de cette hiérarchie... Mais ne vous en faites pas trop, vous pouvez utiliser la première méthode que je vous ai montrée sans risque.

Oui, on voit bien, mais qu'est-ce que c'est que toutes ces méthodes `init(ServletConfig config)` ou `service(HttpServletRequest request, HttpServletResponse response)` ...

Ne vous en faites pas, nous y arrivons...

Vous voyez une méthode `init(ServletConfig config)` présente au plus haut niveau de la hiérarchie. Dans le constructeur, le conteneur crée l'objet `ServletConfig` et invoque ladite méthode.

Ceci pourrait être schématisé comme ceci :



La méthode `init(ServletConfig config)` qu'utilise le conteneur ne doit pas être redéfinie !

Si vous souhaitez faire quelques traitements avant d'utiliser votre servlet, comme l'initialisation d'objets, vous avez une autre méthode `init()` qui, elle, ne prend pas de paramètre. Cette méthode vous est offerte afin que vous puissiez y mettre ce que vous voulez.


Du coup, comment cette méthode est-elle appelée ?

C'est simple, c'est la méthode `init(ServletConfig config)` invoquée par le constructeur qui appelle la méthode `init()` que vous pouvez redéfinir. 

Voici comment vous pouvez utiliser ceci :

Code : Java

```
package com.servlet.test;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletContext;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class DoIt extends HttpServlet{

	String str = "";
	public void init(){
		str = "<p>Paramètre initialisé dans la méthode init !</p>";
	}

	public void doGet(HttpServletRequest request, HttpServletResponse response)
		throws IOException{
		response.setContentType("text/html");
		PrintWriter out = response.getWriter();
		out.println("<h1>Servlet d'exemple ! </h1>");
		out.println(str);
	}

}
```

Bon, tout ceci fait, votre servlet est initialisée et prête à être utilisée par le conteneur.


Donc voici ce qu'il se passe lorsque le conteneur créer la servlet. Et ça à chaque fois que le conteneur reçoit une requête HTTP ?

NON ! MON DIEU NON !
Je crois qu'il faut que je vous fasse aussi un petit topo sur la façon dont la servlet est utilisée... 

Utilisation de la servlet

Vous pensiez que les servlets étaient des objets instanciés à chaque requête HTTP, eh bien non !
Je sais que vous pourrez entendre ce genre de chose "***Blablabla... chaque instance de ma servlet... Blablabla***". Ceci est faux !
Il n'existe qu'une et une seule instance d'une servlet dans le conteneur !

Ce qu'il se passe c'est que pour chaque requête HTTP, un thread est créé pour satisfaire la requête et proposer une réponse !


Que se passe-t-il exactement ?

C'est simple et compliqué à la fois...
Le conteneur reçoit une requête HTTP. Vous savez déjà que ce dernier crée deux objets pour traiter la demande du client :

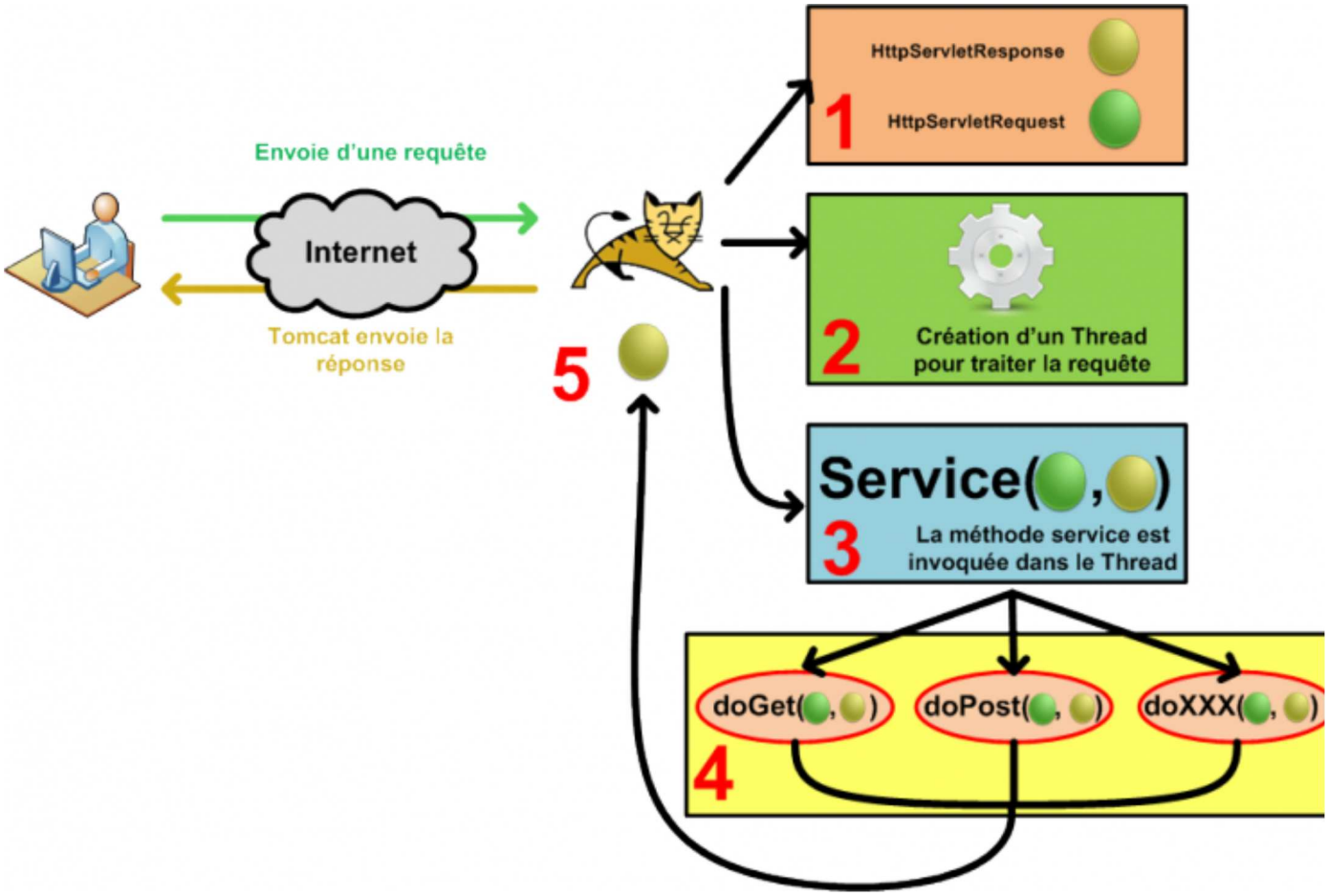
- un objet `HttpServletRequest` ;
- un objet `HttpServletResponse`.

Vous savez aussi que la servlet sait quelle méthode utiliser (`doGet`, `doPost`...) mais ne vous êtes-vous jamais demandé comment nos servlets savent faire ça ?
Si, bien sûr que si... Eh bien c'est grâce à la méthode `service(HttpServletRequest request, HttpServletResponse response)` que tout ceci est possible !
C'est dans cette dernière que le code permettant à la servlet de savoir quelle méthode utiliser se trouve ; par conséquent, il est très vivement déconseillé de redéfinir cette méthode !

Si je reprends la vie de notre servlet partant du principe que la servlet est initialisée, voici ce que ça donnerait :

- la servlet reçoit une requête HTTP ;
- elle crée les deux objets que vous connaissez ;
- elle alloue un nouveau thread à cette requête ;
- elle invoque la méthode `service(HttpServletRequest request, HttpServletResponse response)` en passant les deux objets en paramètres ;
- ladite méthode invoque la bonne méthode de traitement (doGet, doPost...) ;
- le traitement se fait ;
- la réponse est renvoyée au client.

Je sais que vous êtes attachés à mes schémas... Donc je vous en ai préparé un pour reprendre mes dires :

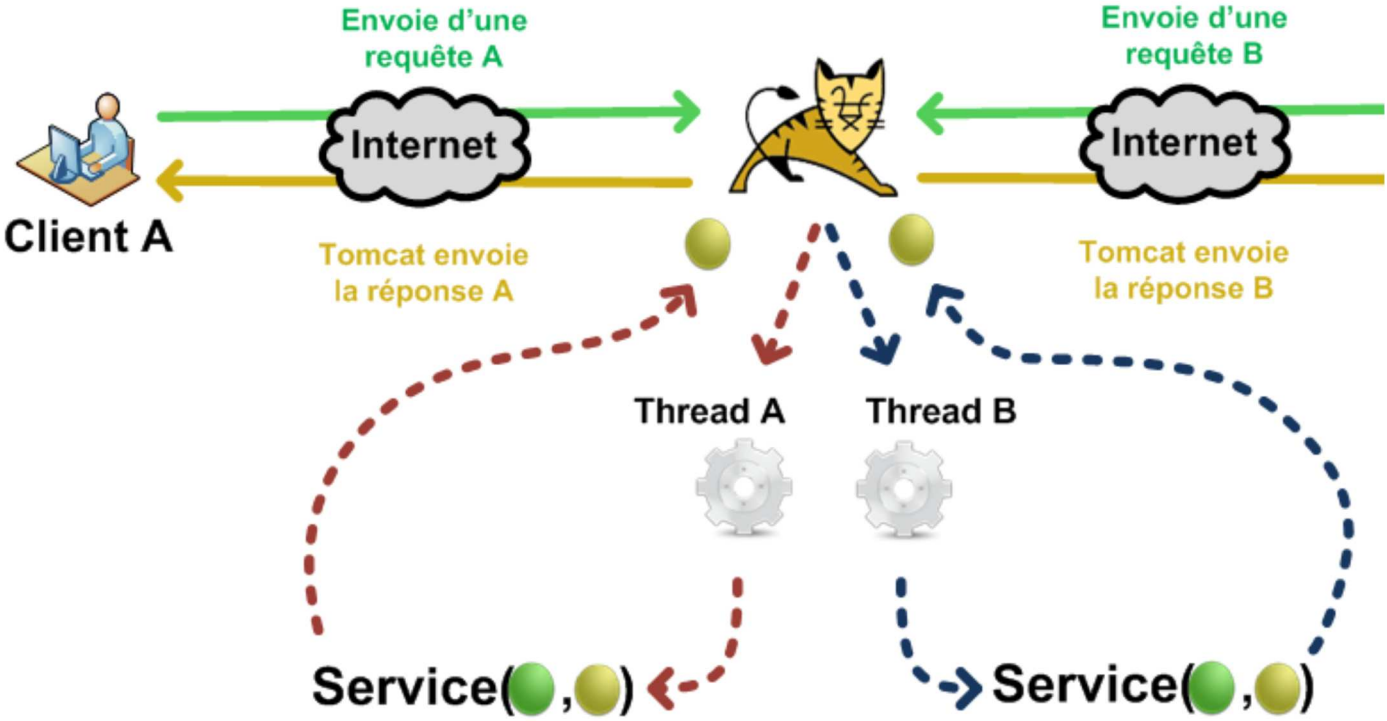


Vous devez y voir plus clair, non ?

Euh, tu nous disais que Tomcat crée un thread pour chaque requête...

Oui, à chaque fois que celui-ci reçoit une requête HTTP, il crée les objets requête et réponse puis, il crée un thread afin d'invoquer la méthode `service(HttpServletRequest req, HttpServletREsponse rep)` . La suite, vous la connaissez...

Je vous ai fait un petit schéma résumant ce qu'il se passe lorsque deux personnes envoient une requête vers la même ressource sur le serveur :



C'est assez explicite je pense...
Par contre, vous devez savoir encore une chose sur notre conteneur : il vous permet d'espionner vos applications...

Le retour des listeners

Nous venons de voir comment sont construites et initialisées nos servlets mais il reste une notion, que j'approfondirai au fil du tuto et que vous devez connaître.
En fait, votre conteneur favori vous permet d'écouter toute la phase de construction de nos servlets et plus encore.

Au long du chapitre précédent, nous avons vu comment définir des objets dans notre contexte mais, parce qu'il y a un mais, ceux-ci étaient définis par une servlet... Pas terrible !...

Comment vous y prendriez-vous pour initialiser un objet au lancement de l'application ? (Une connexion SQL par exemple, mais un bête objet fera l'affaire.)

En définissant cet attribut à notre contexte dans la méthode `init()` de la servlet correspondant à notre page d'accueil.

Ah oui ? Et comment ferez-vous si je passe outre la page d'accueil, en saisissant une URL connue par exemple ?
Vous n'allez tout de même pas mettre cette instruction dans la méthode `init()` de toutes vos servlets !

Dans ces moments-là, la méthode `public static void main(String[] args)` vous manque réellement...
Avoir un endroit où on peut faire ce que l'on veut avant le réel démarrage de l'application.

Cependant, comme je vous le disais dans le chapitre précédent, les concepteurs de la technologie sont de gentils concepteurs : **ils vous ont donné le moyen d'initialiser des objets en dehors de toute servlet !**

Et comment on fait ça ?

Avec de bons vieux événements, comme en programmation événementielle.
Durant toute la phase d'initialisation de notre application, le conteneur vous offre la possibilité d'écouter des événements afin

de pouvoir faire des actions spécifiques à ce moment-là.

Bon : à partir de maintenant, tous les ZérOs qui ne savent pas comment les événements graphiques sont gérés en Java sont priés d'aller faire un tour sur le tuto concernant le pattern observer !

Alors, le but du jeu est de pouvoir initialiser un objet avant qu'une seule servlet ne soit instanciée, ceci afin que ce qui est fait à ce moment (l'initialisation de l'application) puisse être utile à toute l'application.
Déjà, avec ma phrase, vous avez dû comprendre que l'événement que nous allons utiliser était attaché au contexte de notre application : **une seule chose est commune à l'application, le contexte.**

Alors qui dit événement, dit interface à implémenter et, concernant le contexte, nous allons implémenter l'interface : `ServletContextListener`.

Vous pouvez voir que cette interface nous donne la possibilité d'agir lors de la création du contexte ou lors de sa destruction.
Alors, vu que vous êtes friands d'exemples en tout genre, nous allons faire un test.
Nous allons faire une servlet qui appelle un attribut attaché à l'objet `ServletContext` mais défini lors de la création dudit objet.
Pour faire ceci, nous allons avoir besoin :

- d'une servlet ;
- d'un objet servant à écouter l'événement ;
- d'un objet codé par nos soins.

J'ai fait un nouveau projet Tomcat et voici ce que j'y ai mis.
Un objet métier que vous avez déjà vu tout à l'heure, l'objet `ZColor`.

Fichier ZColor.java

Code : Java

```
package com.sdz.test;

public class ZColor {

	private String color = "";
	public ZColor(String color){
		this.color = color;
	}

	public String toString(){
		return "COULEUR -> " + this.color;
	}
}
```

Bon, vous connaissez cet objet, rien à ajouter à son sujet...
Maintenant, nous allons ajouter un objet `ZColor` à notre contexte lors de sa création, ceci via la classe qui suit.
Il s'agit de notre classe qui va se charger d'écouter aux portes de notre conteneur... Une fois le contexte créé, la méthode `contextInitialized(ServletContextEvent event)` sera invoquée.

Fichier ContextListener.java

Code : Java

```java
package com.sdz.test;

import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;

public class ContextListener implements ServletContextListener{

        public void contextDestroyed(ServletContextEvent event) {}

        public void contextInitialized(ServletContextEvent event) {
                //Nous avons accès à l'objet ServletContext via l'objet
ServletContextEvent
                event.getServletContext().setAttribute("color", new ZColor("ROUGE"));
        }
}
```

Maintenant, construisons une servlet qui aura pour rôle d'utiliser l'objet initialisé dans le contexte.

Fichier Index.java

Code : Java

```java
package com.sdz.test;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class Index extends HttpServlet {

        public void doGet(HttpServletRequest request, HttpServletResponse response)
                                                throws IOException, ServletException{
                response.setContentType("text/html");
                PrintWriter out = response.getWriter();
                out.println("<h2>Alors le contexte ?</h2>");
                out.println("<p>
" + getServletContext().getAttribute("color") + " </p>");
        }
}
```

Voilà, nous avons presque terminé, nous n'avons plus qu'à configurer le contexte de notre application avec le fichier **web.xml**.

Fichier web.xml

Code : XML

```
<web-app>

	<servlet>
		<servlet-class>com.sdz.test.Index</servlet-class>
		<servlet-name>IndexServlet</servlet-name>
	</servlet>

	<servlet-mapping>
		<servlet-name>IndexServlet</servlet-name>
		<url-pattern>/</url-pattern>
	</servlet-mapping>

	<listener>
		<listener-class>com.sdz.test.ContextListener</listener-class>
	</listener>

</web-app>
```


Oh ! Une nouveauté !

Oui, difficile de ne pas la voir...
Vous l'aurez compris, ces balises servent à dire à Tomcat que des classes sont à utiliser comme des **listeners**.
Alors démarrez Tomcat et allez sur notre test, et voici ce que vous devriez avoir sous les yeux :

Alors le contexte ?

COULEUR -> ROUGE

Pratique, n'est-ce pas ?
Vous venez de créer des attributs pour toute votre application sans ajouter une ligne de code à vos servlets.


En effet, en plus ce n'est pas trop dur... 

Ah, je peux compliquer la tâche alors ! 

Des listeners, encore des listeners

Puisque vous êtes chauds, c'est le bon moment pour vous dire que vous avez plusieurs listeners à votre disposition.
Vous venez de voir comment utiliser celui qui est invoqué lorsque le contexte est initialisé, mais il y en a d'autres ! 

En voici une liste :

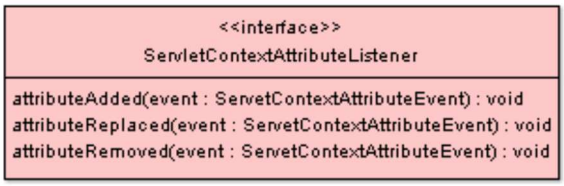
- `ServletContextListener` ;
- `ServletContextAttributeListener` ;
- `ServletRequestListener` ;
- `SerlvetRequestAttributeListener` ;
- `HttpSessionListener` ;
- `HttpSessionAttributeListener` ;
- `HttpSessionBindingListener` ;
- `HttpSessionActivateListener`.

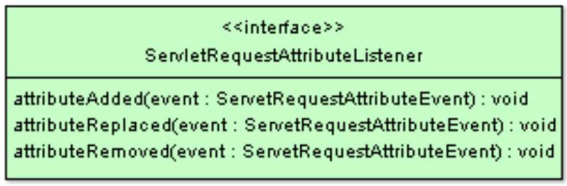
Je ne vous parlerai que des quatre premiers pour le moment puisque les autres concernent les sessions : nous n'avons pas

encore parlé des sessions.
Par contre, pour les premiers, voici à quoi ils correspondent :

- `ServletContextListener` : invoqué lorsque l'objet `ServletContext` est créé ;
- `ServletContextAttributeListener` : invoqué lorsqu'un attribut est ajouté à l'objet `ServletContext` ;
- `ServletRequestListener` : invoqué lorsque l'objet `ServletRequest` est créé ;
- `SerlvetRequestAttributeListener` : invoqué lorsqu'un attribut est ajouté à l'objet `ServletRequest` .

Voici le détail de ces interfaces avec leurs méthodes :





En rouge vous pouvez voir les événements relatifs au contexte et en vert les événements relatifs à la requête.
Nous allons réaliser un petit exemple. Nous allons donc utiliser ces événements, ceci avec des classes qui auront pour rôle de compter le nombre d'attributs affectés à nos différents objets.

Voici ci-dessous les objets qui auront cette fonction.

Fichier CountContext.java

Code : Java

```java
package com.sdz.test;

public class CountContext {

	private static int count = 0;

	public static void incremente(){++count;}
	public static int getCount(){return count;}
}
```

Fichier CountContextAttribute.java

Code : Java

```java
package com.sdz.test;

public class CountContextAttribute {
	private static int count = 0;

	public static void incremente(){++count;}
	public static int getCount(){return count;}
}
```

Fichier CountRequest.java

Code : Java

```
package com.sdz.test;

public class CountRequest {
        private static int count = 0;

        public static void incremente(){++count;}
        public static int getCount(){return count;}
}
```

J'ai fait trois objets pour être sûr que vous ne m'accuseriez pas de tricherie. 
Rien de compliqué ici, parce que, si vous ne comprenez pas la fonction de ces objets, retour au tuto Java illico presto !

Maintenant, voici les objets implémentant les interfaces mentionnées plus haut (je ne les ai pas toutes mises).

Fichier ContextListener.java

Code : Java

```
package com.sdz.test;

import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;

public class ContextListener implements ServletContextListener{

        public void contextDestroyed(ServletContextEvent event) {}

        public void contextInitialized(ServletContextEvent event) {
                event.getServletContext().setAttribute("color", new ZColor("ROUGE"));
                CountContext.incremente();
        }
}
```

Fichier ContextAttributeListener.java

Code : Java

```
package com.sdz.test;

import javax.servlet.ServletContextAttributeEvent;
import javax.servlet.ServletContextAttributeListener;

public class ContextAttributeListener implements ServletContextAttributeListener{

        public void attributeAdded(ServletContextAttributeEvent event) {
                CountContextAttribute.incremente();
        }
        public void attributeRemoved(ServletContextAttributeEvent event) {}
        public void attributeReplaced(ServletContextAttributeEvent event) {}
}
```

Fichier RequestListener.java

Code : Java

```
package com.sdz.test;

import javax.servlet.ServletRequestEvent;
import javax.servlet.ServletRequestListener;

public class RequestListener implements ServletRequestListener{

        public void requestDestroyed(ServletRequestEvent event) {
                CountRequest.incremente();
        }
        public void requestInitialized(ServletRequestEvent event) {}
}
```

Il ne manque plus que la servlet de test ainsi que le fichier **web.xml**.
J'ai modifié un peu la servlet afin d'isoler notre test. J'ai aussi ajouté les listeners dans le fichier **web.xml**.

Fichier Index.java

Code : Java

```
package com.sdz.test;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class Index extends HttpServlet {

        public void doGet(HttpServletRequest request, HttpServletResponse response)
                                                throws IOException, ServletException{
                response.setContentType("text/html");
                PrintWriter out = response.getWriter();

                out.println("<p>Nombre d'objet(s) ServletContext :
" + CountContext.getCount() + "</p>");
                out.println("<p>Nombre d'attribut(s) du contexte :
" + CountContextAttribute.getCount() + "</p>");
                out.println("<p>Nombre d'objet(s) HttpServletRequest :
" + CountRequest.getCount() + "</p>");
        }
}
```

Fichier web.xml

Code : XML

```
<web-app>

	<servlet>
		<servlet-class>com.sdz.test.Index</servlet-class>
		<servlet-name>IndexServlet</servlet-name>
	</servlet>

	<servlet-mapping>
		<servlet-name>IndexServlet</servlet-name>
		<url-pattern>/</url-pattern>
	</servlet-mapping>

	<listener>
		<listener-class>com.sdz.test.ContextListener</listener-class>
		<listener-
class>com.sdz.test.ContextAttributeListener</listener-class>
		<listener-class>com.sdz.test.RequestListener</listener-class>
	</listener>

</web-app>
```

Ceci fait, il ne vous reste plus qu'à relancer Tomcat et de retourner sur notre test. Vous devriez obtenir ceci (j'ai appuyé quelquefois sur **F5**) :

Nombre d'objet(s) ServletContext : 1

Nombre d'attribut(s) du contexte : 1

Nombre d'objet(s) HttpServletRequest : 3

Notre système de listeners fonctionne à merveille, vous avez appris à écouter les événements d'initialisation et à les utiliser pour paramétrer votre application : mes compliments !

Bon, je crois que vous en avez assez vu pour l'instant. Rendez-vous au QCM !

Voilà, maintenant vous êtes incollables sur la vie d'une servlet !
Vous pourrez même écrire un livre là-dessus.

Eh bien ne vous en déplaise, le conteneur vous cache encore pas mal de choses !
Il se passe des choses que vous ignorez toujours lors de l'initialisation de vos servlets... Nous aurons l'occasion d'y revenir un peu plus tard.

Par contre, dans le chapitre qui suit, vous verrez que le conteneur prend soin de vos JSP tout autant que de vos servlets... Et pour cause...

Partie encore en cours d'édition...
Revenez y faire un tour.

Ce tuto est encore en cours d'édition mais j'espère sincèrement qu'il vous aura permis de découvrir les méandres de JEE avec convivialité et humour !