Créez votre application web avec Java EE

Par Coyote

www.siteduzero.com

Licence Creative Commons BY-NC-SA 2.0
Dernière mise à jour le 24/06/2012

Sommaire

Sommaire 1
Partager 2
Créez votre application web avec Java EE 4
Comment lire ce cours ? 4
Partie 1 : Introduction au Java EE 5
Internet et le web 5
Internet n'est pas le web ! 5
Comment ça marche 5
Les langages du web 6
Le Java EE mis à nu ! 7
Principes de fonctionnement 7
Le modèle MVC : en théorie 8
Le modèle MVC : en pratique 9
Outils et environnement de développement 11
L'IDE Eclipse 11
Présentation 11
Version 11
Le serveur Tomcat 12
Présentation 12
Installation 12
Création du projet web avec Eclipse 15
Structure d'une application Java EE 25
Structure standard 26
Votre première page web 26
Partie 2 : Premiers pas 31
La servlet 31
Derrière les rideaux 31
Retour sur HTTP 31
Pendant ce temps là, sur le serveur... 32
Création 32
Mise en place 35
Mise en service 38
Servlet avec vue... 42
Introduction aux JSP 43
Nature d'une JSP 43
Mise en place d'une JSP 44
Création de la vue 44
Cycle de vie d'une JSP 46
Mise en relation avec notre servlet 48
Transmission de données 49
Données issues du serveur : les attributs 50
Données issues du client : les paramètres 51
Le JavaBean 54
Objectifs 55
Structure 55
Mise en place 56
La technologie JSP (1/2) 62
Les balises 63
Les directives 64
La portée des objets 67
Les actions standard 69
L'action standard useBean 69
L'action standard getProperty 70
L'action standard setProperty 70
L'action standard forward 71
La technologie JSP (2/2) 72
Expression Language 73
Présentation 73
Les tests 73
Les JavaBeans 75
Désactiver l'évaluation des expressions EL 77
Les objets implicites 79
Les objets de la technologie JSP 79
Les objets de la technologie EL 81
Des problèmes de vue ? 87
Nettoyons notre exemple 87
Complétons notre exemple... 89
Le point sur ce qu'il nous manque 93
Documentation 93
Liens utiles 94
TP Fil rouge - Étape 1 94
Objectifs 95
Contexte 95

Fonctionnalités ... 95
Contraintes ... 95
Conseils ... 98
À propos des formulaires ... 98
Le modèle ... 99
Les contrôleurs ... 99
Les vues ... 99
Création du projet ... 99
Illustration du comportement attendu ... 100
Exemples de rendu du comportement attendu ... 101
Correction ... 104
Le code des beans ... 104
Le code des servlets ... 106
Le code des JSP ... 109
Partie 3 : Une bonne vue grâce à la JSTL ... 112
Objectifs et configuration ... 112
C'est sa raison d'être... ♪♫ ... 112
Lisibilité du code produit ... 112
Moins de code à écrire ... 113
Vous avez dit MVC ? ... 113
À retenir ... 114
Plusieurs versions ... 114
Configuration ... 114
Configuration de la JSTL ... 114
La bibliothèque Core ... 117
Les variables et expressions ... 118
Affichage d'une expression ... 118
Gestion d'une variable ... 119
Les conditions ... 122
Une condition simple ... 122
Des conditions multiples ... 123
Les boucles ... 123
Boucle "classique" ... 123
Itération sur une collection ... 125
Itération sur une chaîne de caractères ... 128
Ce que la JSTL ne permet pas (encore) de faire ... 128
Les liens ... 128
Liens ... 128
Redirection ... 131
Imports ... 132
JSTL core : exercice d'application ... 134
Les bases de l'exercice ... 134
Correction ... 136
La bibliothèque xml ... 139
La syntaxe XPath ... 140
Le langage XPath ... 140
Les actions de base ... 141
Récupérer et analyser un document ... 141
Afficher une expression ... 144
Créer une variable ... 145
Les conditions ... 145
Les conditions ... 145
Les boucles ... 146
Les boucles ... 146
Les tranformations ... 147
Transformations ... 147
JSTL xml : exercice d'application ... 150
Les bases de l'exercice ... 150
Correction ... 151
Faisons le point ! ... 152
Reprenons notre exemple ... 153
Quelques conseils ... 155
Utilisation de constantes ... 155
Inclure automatiquement la JSTL Core à toutes vos JSP ... 157
Formater proprement et automatiquement votre code avec Eclipse ... 158
Documentation ... 163
Liens utiles ... 163
TP Fil rouge - Étape 2 ... 165
Objectifs ... 165
Utilisation de la JSTL ... 165
Application des bonnes pratiques ... 165
Exemples de rendus ... 165
Conseils ... 167
Utilisation de la JSTL ... 167
Application des bonnes pratiques ... 167
Correction ... 167
Code des servlets ... 168
Code des JSP ... 171
Partie 4 : Une application interactive ! ... 176
Formulaires : le b.a.-ba ... 177
Mise en place ... 177

JSP & CSS ... 184
La servlet ... 187
L'envoi des données ... 188
Contrôle : côté servlet ... 189
Affichage : côté JSP ... 193
Formulaires : à la mode MVC ... 201
Analyse de notre conception ... 201
Création du modèle ... 202
Reprise de la servlet ... 205
Reprise de la JSP ... 206
TP Fil rouge - Étape 3 ... 208
Objectifs ... 209
Fonctionnalités ... 209
Exemples de rendus ... 209
Conseils ... 211
Correction ... 211
Code des objets métier ... 211
Code des servlets ... 217
Code des JSP ... 219
La session : connectez vos clients ... 223
Le formulaire ... 224
Le principe de la session ... 224
Le modèle ... 227
La servlet ... 229
Les vérifications ... 231
Test du formulaire de connexion ... 231
Test de la destruction de session ... 233
Derrière les rideaux ... 239
La théorie : principe de fonctionnement ... 239
La pratique : scrutons nos requêtes et réponses ... 240
Le bilan ... 253
Le filtre : créez un espace membre ... 253
Restreindre l'accès à une page ... 254
Les pages d'exemple ... 254
La servlet de contrôle ... 255
Test du système ... 257
Le problème ... 258
Le principe du filtre ... 258
Généralités ... 258
Fonctionnement ... 259
Cycle de vie ... 260
Restreindre l'accès à un ensemble de pages ... 261
Restreindre un répertoire ... 261
Restreindre l'application entière ... 267
Désactiver le filtre ... 273
Le cookie : le navigateur vous ouvre ses portes ... 275
Le principe du cookie ... 275
Côté HTTP ... 275
Côté Java EE ... 275
Souvenez-vous de vos clients ! ... 275
Reprise de la servlet ... 276
Reprise de la JSP ... 281
Vérifications ... 282
À propos de la sécurité ... 286
Avancement du cours ... 286
Et après ? ... 286

Créez votre application web avec Java EE

Par

Coyote

Mise à jour : 23/06/2012
Difficulté : Intermédiaire Durée d'étude : 1 mois
(cc) BY-NC-SA
23 073 visites depuis 7 jours, classé 14/782

La création d'applications web avec **Java EE** semble compliquée à beaucoup de débutants. Une énorme nébuleuse de sigles en tout genre gravite autour de la plate-forme, un nombre conséquent de technologies et d'approches différentes existent : servlet, JSP, Javabean, MVC, JDBC, JNDI, EJB, JPA, JMS, JSF, Struts, Spring, Tomcat, Glassfish, JBoss, WebSphere, WebLogic... La liste n'en finit pas, et pour un novice ne pas étouffer sous une telle avalanche est bien souvent mission impossible !

Soyons honnêtes, ce tutoriel ne vous expliquera pas le fonctionnement et l'utilisation de toutes ces technologies. Car ça aussi, c'est mission impossible ! Il faudrait autant de tutos...

Non, ce cours a pour objectif de guider vos premiers pas dans l'univers Java EE : après quelques explications sur les concepts généraux et les bonnes pratiques en vigueur, vous allez entrer dans le vif du sujet et découvrir comment créer un projet web, en y ajoutant de la complexité au fur et à mesure que le cours avancera. À la fin du cours, vous serez capables de créer une application web qui respecte les standards reconnus dans le domaine et vous disposerez des bases nécessaires pour utiliser la plupart des technologies se basant sur Java EE.

Comment lire ce cours ?

Un contenu conséquent est prévu, mais je ne vais volontairement pas être exhaustif : les technologies abordées sont très vastes, et l'objectif du cours est de vous apprendre à créer une application. Si je vous réécrivais la documentation de la plate-forme Java EE en français, ça serait tout simplement imbuvable. Je vais ainsi fortement insister sur des points non documentés et des pratiques que je juge importantes, et être plus expéditif sur certains points, pour lesquels je me contenterai de vous présenter les bases et de vous renvoyer vers les documentations et sources officielles pour plus d'informations. Je vous invite donc à ne pas vous limiter à la seule lecture de ce cours, et à parcourir chacun des liens que j'ai mis en place tout au long des chapitres.

Enfin, avant d'attaquer sachez que ce cours ne part pas totalement de zéro : il vous faut des bases en Java afin de ne pas vous sentir largués dès les premiers chapitres. Ainsi, si vous n'êtes pas encore familier avec le langage, vous pouvez lire les parties 1 et 2 du tutoriel sur le Java du Site du Zéro.

Partie 1 : Introduction au Java EE

Dans cette courte première partie, nous allons poser le décor : quelles sont les briques de base d'une application Java EE, comment elles interagissent, quels outils utiliser pour développer un projet...

Internet et le web

Avant de nous plonger dans l'univers Java EE, commençons par nous pencher un instant sur ce qu'est le web, et sur ce qu'il n'est pas. Simples rappels pour certains d'entre vous, découverte pour d'autres, nous allons ici expliquer ce qui se passe dans les coulisses lorsque l'on accède à un site web depuis son navigateur. Nous aborderons enfin brièvement les autres langages existant, et les raisons qui nous poussent à choisir Java EE.

Internet n'est pas le web !

Avant tout, il ne faut pas confondre l'internet et le web :

- l'internet est le réseau, le support physique de l'information. Pour faire simple, c'est un ensemble de machines, de câbles et d'éléments réseaux en tout genre éparpillés sur la surface du globe ;
- le web constitue une partie seulement du contenu accessible sur l'internet. Vous connaissez et utilisez d'autres contenus, comme le courrier électronique ou encore la messagerie instantanée.

Un site web est un ensemble constitué de pages web (elles-mêmes faites de fichiers HTML, CSS, Javascript, etc.). Lorsqu'on développe puis publie un site web, on met en réalité en ligne du contenu sur internet. On distingue deux types de sites :

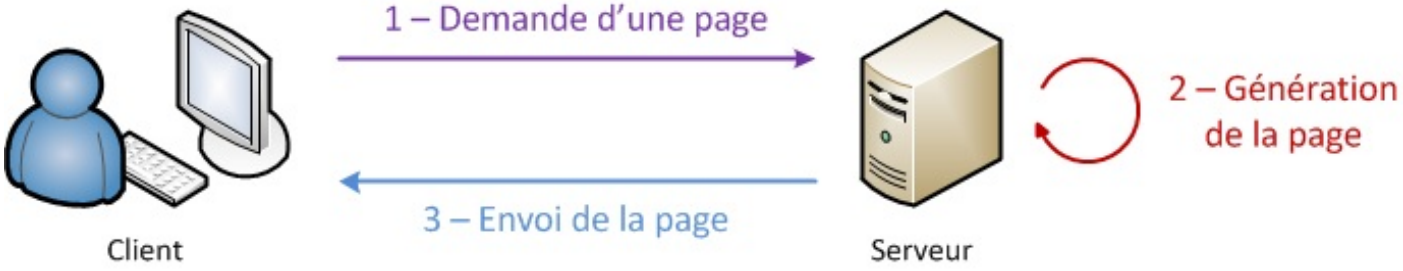
- **les sites internet statiques** : ce sont des sites dont le contenu est "fixe", il n'est modifiable que par le propriétaire du site. Ils sont réalisés à l'aide des technologies HTML, CSS et Javascript uniquement.
- **les sites internet dynamiques** : ce sont des sites dont le contenu est "dynamique",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

Si vous n'êtes pas à l'aise avec l'environnement web ni avec les technologies nécessaires à la création de sites statiques, commencez-donc par vous y initier en suivant le cours HTML5 / CSS3. Une fois tout cela bien assimilé, vous êtes prêts pour vous lancer !

Comment ça marche

Lorsqu'un utilisateur consulte un site, ce qui se passe derrière les rideaux est un simple échange entre un client et un serveur :

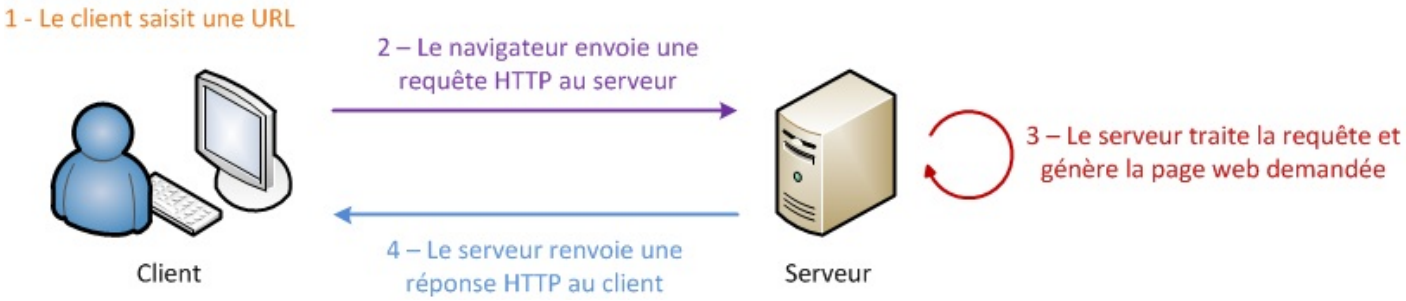
- **le client** : dans la plupart des cas, c'est le navigateur installé sur votre ordinateur. Retenez que ce n'est pas le seul moyen d'accéder au web, mais c'est celui qui nous intéresse dans ce cours.
- **le serveur** : c'est la machine sur laquelle le site est hébergé, où les fichiers sont stockés et les pages web générées.



Echange dynamique client <-> serveur

La communication qui s'effectue entre le client et le serveur est régie par des règles bien définies : le **protocole HTTP**. Entrons donc un peu plus dans le détail, et regardons de quoi est constitué un échange simple :

1. l'utilisateur saisit une URL dans la barre d'adresses de son navigateur ;
2. le navigateur envoie alors une **requête HTTP** au serveur pour lui demander la page correspondante ;
3. le serveur reçoit cette requête, l'interprète et génère alors une page web qu'il va renvoyer au client par le biais d'une **réponse HTTP** ;
4. le navigateur reçoit via cette réponse la page web finale, qu'il affiche alors à l'utilisateur.



Echange dynamique HTTP client <-> serveur

Ce qu'il faut comprendre et retenir de tout ça :

- les données sont échangées entre le client et le serveur via le protocole HTTP ;
- le client ne comprend que les langages de présentation de l'information, en d'autres termes les technologies HTML, CSS et Javascript ;
- les pages sont générées sur le serveur de manière dynamique, à partir du code source du site.

Les langages du web

Nous venons de le voir dans le dernier paragraphe, le client ne fait que recevoir des pages web, les afficher à l'utilisateur et transmettre ses actions au serveur. Vous savez déjà que les langages utilisés pour mettre en forme les données et les afficher à l'utilisateur sont le HTML, le CSS et éventuellement le Javascript. Ceux-ci ont une caractéristique commune importante : **ils sont tous interprétés par le navigateur**, directement sur la machine client. D'ailleurs, le client est uniquement capable de comprendre ces quelques langages, rien de plus !

Eh bien le serveur aussi dispose de technologies bien à lui, que lui seul est capable de comprendre : une batterie complète ayant pour objectif final de générer les pages web à envoyer au client, avec tous les traitements que cela peut impliquer au passage : analyse des données reçues via HTTP, transformation des données, enregistrement des données dans une base de données ou des fichiers, intégration des données dans le design...

Seulement à la différence du couple HTML & CSS qui est un standard incontournable pour la mise en forme des pages web, il existe plusieurs technologies capables de traiter les informations sur le serveur. Java EE est l'une d'entre elles, mais il en existe d'autres : PHP, .NET, Django et Ruby on Rails, pour ne citer que les principales. Toutes offrent sensiblement les mêmes possibilités, mais toutes utilisent un langage et un environnement bien à elles !

Comment choisir la technologie la mieux adaptée à son projet ?

C'est en effet une très bonne question : qu'est-ce qui permet de se décider parmi cet éventail de possibilités ? C'est un débat presque sans fin. Toutefois, dans la vie réelle le choix est bien souvent influencé voire dicté par :

- votre propre expérience : si vous avez déjà développé en Java, Python ou C# auparavant, il semble prudent de vous orienter respectivement vers Java EE, Django et .NET ;
- vos besoins : rapidité de développement, faible utilisation des ressources sur le serveur, réactivité de la communauté soutenant la technologie, ampleur de la documentation disponible en ligne, coût, etc.

Quoi qu'il en soit, peu importent les raisons qui vous ont poussé à lire ce cours, nous sommes bien là pour apprendre le Java EE !

Nous voilà maintenant prêts à faire un premier pas vers le Java EE. Dans le prochain chapitre, nous allons découvrir de quoi est faite une application web, et comment elle interagit avec le protocole HTTP.

Le Java EE mis à nu !

Le Java Enterprise Edition, comme son nom l'indique, a été créé pour le développement d'applications d'entreprises. Nous nous y attarderons dans le chapitre suivant, mais sachez d'ores et déjà que ses spécifications ont été pensées afin notamment de faciliter le travail en équipe sur un même projet : l'application est découpée en couches, et le serveur sur lequel tourne l'application est lui-même découpé en plusieurs niveaux. Pour faire simple, Java EE fournit un ensemble d'extensions au Java standard afin de faciliter la création d'applications centralisées.

Voyons comment tout cela s'agence !

Principes de fonctionnement

Nous venons de découvrir qu'afin de pouvoir communiquer entre eux, le client et le serveur doivent se parler via HTTP. Nous savons déjà que côté client, le navigateur s'en occupe. Côté serveur, qui s'en charge ? C'est un composant que l'on nomme logiquement **serveur HTTP**. Son travail est simple : il doit écouter tout ce qui arrive sur le port utilisé par le protocole HTTP, le port 80, et scruter chaque requête entrante. C'est tout ce qu'il fait, c'est en somme une interface de communication avec le protocole.

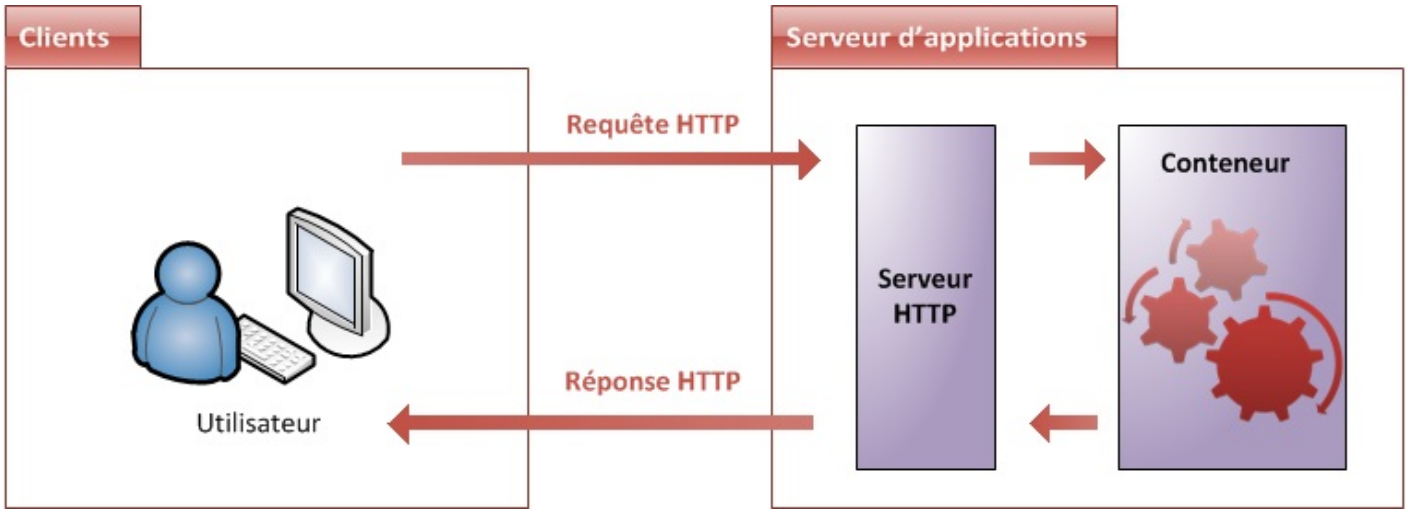
À titre informatif, voici les deux plus connus : Apache HTTP Server et IIS (Microsoft).

Cependant, nous n'allons directement utiliser ni l'un ni l'autre. Pourquoi ?

Être capable de discuter via HTTP c'est bien, mais notre serveur doit permettre d'effectuer d'autres tâches. En effet, une fois la requête HTTP lue et analysée, il faut encore traiter son contenu et éventuellement renvoyer une réponse au client en conséquence. Vous devez probablement déjà savoir que cette responsabilité vous incombe en grande partie : c'est le code que vous allez écrire qui va décider de quoi faire lorsque telle requête arrive ! Seulement comme je viens de vous l'annoncer, un serveur HTTP de base ne peut pas gérer votre application, ce n'est pas son travail.

Remarque : cette affirmation est en partie fausse, dans le sens où la plupart des serveurs HTTP sont devenus des serveurs webs à part entière, incluant des *plugins* qui les rendent capables de supporter des langages de scripts comme le PHP, l'ASP, etc.

Ainsi, nous avons besoin d'une solution plus globale : ce composant, qui va se charger d'exécuter votre code en plus de faire le travail du serveur HTTP, se nomme le **serveur d'applications**. Donner une définition exacte du terme est difficile : ce que nous pouvons en retenir, c'est qu'un tel serveur inclut un serveur HTTP, et y ajoute la gestion d'objets de diverses natures au travers d'un composant que nous allons pour le moment nommer **le conteneur**.



Concrètement, le serveur d'applications va :

- récupérer les requêtes HTTP issues des clients ;
- les mettre dans des boîtes, des objets, que votre code sera capable de manipuler ;
- faire passer ces objets dans la moulinette qu'est votre application, via le conteneur ;
- renvoyer des réponses HTTP aux clients, en se basant sur les objets retournés par votre code.

Là encore, il en existe plusieurs sur le marché, que l'on peut découper en deux secteurs :

- les solutions propriétaires et payantes : WebLogic et WebSphere, respectivement issues de chez Oracle et IBM, sont les références dans le domaine. Massivement utilisées dans les banques et la finance notamment, elles sont à la fois robustes, finement paramétrables et très coûteuses.
- les solutions libres et gratuites : Apache Tomcat, JBoss, GlassFish et Jonas en sont les principaux représentants.

Comment faire un choix parmi toutes ces solutions ?

Hormis les problématiques de coûts évidentes, d'autres paramètres peuvent influencer votre décision, citons par exemple la rapidité de chargement et d'exécution, ainsi que la quantité de technologies supportées. En ce qui nous concerne, nous partons de zéro : ainsi, un serveur d'applications basique, léger et gratuit fera très bien l'affaire. Cela tombe bien, il en existe justement un qui répond parfaitement à tous nos besoins : **Apache Tomcat**.

Pour information, c'est d'ailleurs souvent ce type de serveurs qui est utilisé lors des phases de développement de grands projets en entreprise. Le coût des licences des solutions propriétaires étant élevé, ce n'est que lors de la mise en service sur la machine finale (on parle alors de mise en production) que l'on change éventuellement pour une telle solution.

Avant de découvrir et prendre en main Tomcat, il nous reste encore quelques concepts clés à aborder !

Le modèle MVC : en théorie

Qu'est-ce qu'un modèle de conception ?

En anglais « design pattern », un modèle de conception (ou encore patron de conception) est une simple **bonne pratique**, qui répond à un problème de conception d'une application. C'est en quelque sorte une ligne de conduite qui permet de décrire les grandes lignes d'une solution. De tels modèles sont issus de l'expérience des concepteurs et développeurs d'applications : c'est en effet uniquement après une certaine période d'utilisation que peuvent être mises en évidence des pratiques plus efficaces que d'autres, pratiques qui sont alors structurées en modèles et considérées comme **standard**.

Maintenant que nous sommes au point sur ce qu'est un modèle, la seconde question à se poser concerne bien évidemment le Java EE.

Que recommandent les développeurs Java EE expérimentés ?

Il faut bien vous rendre compte qu'à l'origine, Java EE permet plus ou moins de coder son application comme on le souhaite : en d'autres termes, on peut coder n'importe comment ! Or on sait que dans Java EE, il y a "Entreprise", et que ça n'est pas là pour faire joli ! Le développement en entreprises implique entre autres :

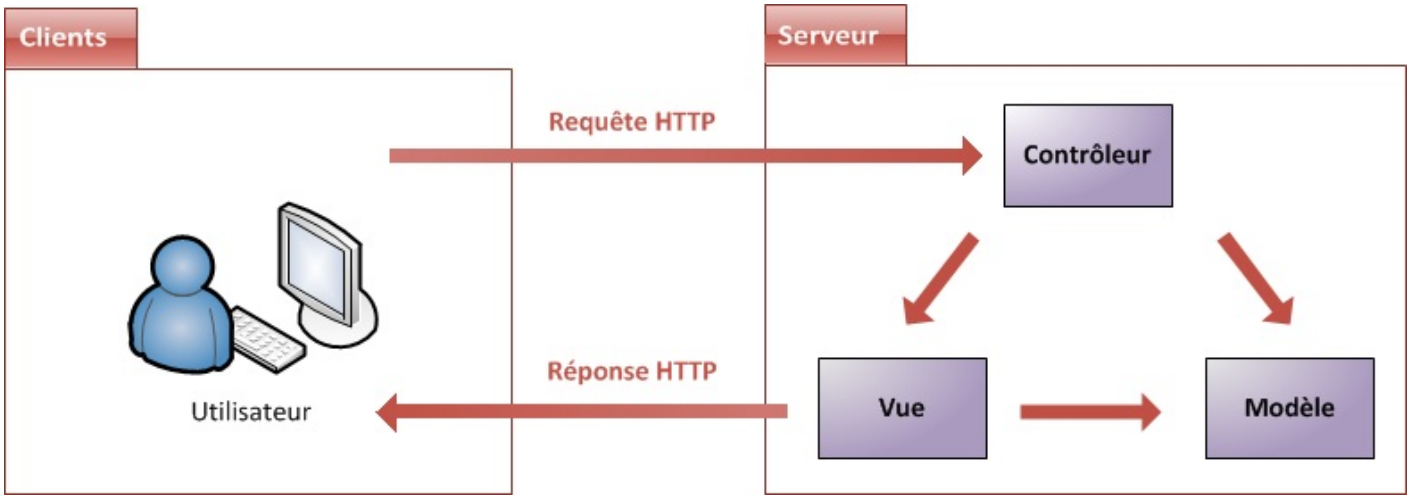
- que l'on puisse être amené à travailler à plusieurs contributeurs sur un même projet ou une même application (travail en équipe) ;
- que l'on puisse être amené à maintenir et corriger une application que l'on n'a pas créée soi-même ;
- que l'on puisse être amené à faire évoluer une application que l'on n'a pas créée soi-même.

Pour toutes ces raisons, il est nécessaire d'adopter une architecture plus ou moins standard, que tout développeur peut reconnaître, c'est-à-dire dans laquelle tout développeur sait se repérer.

Il a très vite été remarqué qu'un modèle permettait de répondre à ces besoins, et qu'il s'appliquait particulièrement bien à la conception d'applications Java EE : **le modèle MVC (Modèle-Vue-Contrôleur)**.
Il découpe littéralement l'application en couches distinctes, et de ce fait impacte très fortement l'organisation du code ! Voici dans les grandes lignes ce qu'impose MVC :

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

Ce modèle peut être représenté par le schéma suivant :



MVC

Ne vous faites pas de soucis si c'est encore flou dans votre esprit : nous reviendrons à maintes reprises sur ces concepts au fur et à mesure que nous progresserons dans notre apprentissage.

Le modèle MVC : en pratique

Le schéma précédent est très global, et ce pour vous permettre de bien visualiser l'ensemble du système. Tout cela est encore assez abstrait, et c'est volontaire ! En effet, chaque projet présente ses propres contraintes, et amène le développeur à faire des choix. Ainsi, on observe énormément de variantes dès que l'on entre un peu plus dans le détail de chacun de ces blocs.

Prenons l'exemple du sous-bloc représentant les données (donc à l'intérieur la couche Modèle) :

- Quel est le type de stockage dont a besoin votre application ?
- Quelle est l'envergure de votre application ?
- Disposez-vous d'un budget ?
- La quantité de donnée produite par votre application va-t-elle être amenée à fortement évoluer ?
- ...

La liste de questions est souvent longue, et réalisez bien que tous ces points sont autant de paramètres qui peuvent influencer la conception de votre application, et donc vos choix au niveau de l'architecture. Ainsi, détailler plus finement les blocs composant une application n'est faisable qu'au cas par cas, idem pour les relations entre ceux-ci.

Toutefois, nous pouvons d'ores et déjà étudier le Java EE "nu", sans frameworks ni fioritures. Voici une courte introduction de chacune des couches composant une application web suivant le modèle MVC :

Modèle : des traitements et des données

Dans le modèle, on trouve à la fois les données et les traitements à appliquer à ces données. Ce bloc contient donc des objets Java d'une part, qui peuvent contenir des attributs (données) et méthodes (traitements) qui leur sont propres, et un système capable de stocker des données d'autre part. Rien de bien transcendant ici, et la complexité du code ici dépendra bien évidemment de la complexité des traitements à effectuer par votre application.

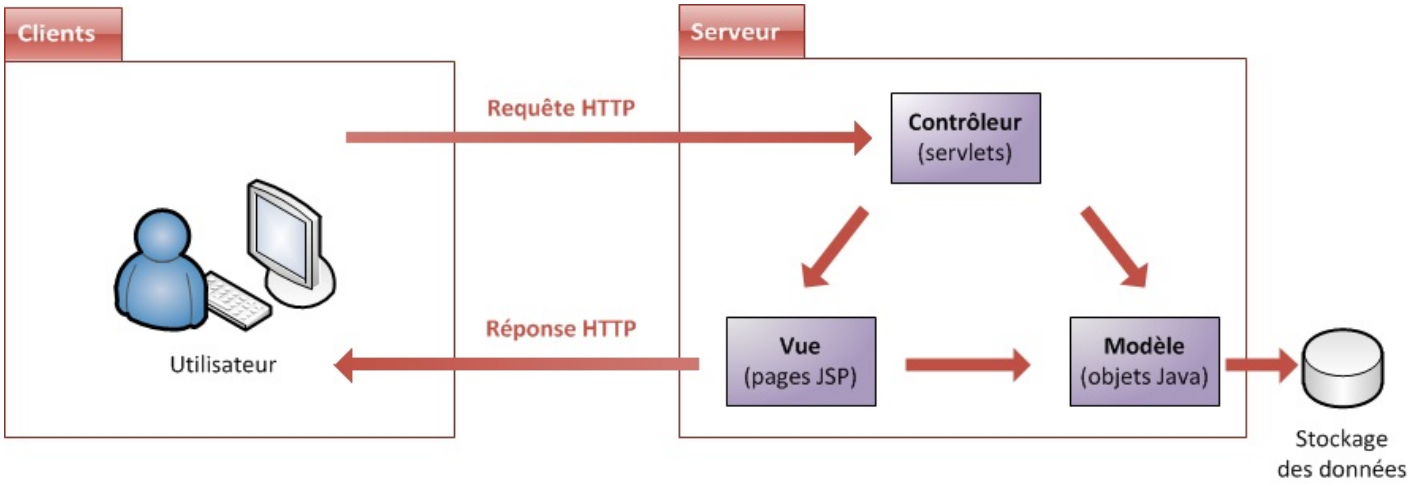
Vue : des pages JSP

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

Contrôleur : des servlets

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

Rien que dans ces quelques lignes, il y a beaucoup d'informations. Pas de panique, nous reviendrons sur tout cela en long, en large et en travers dans les parties suivantes de ce cours ! Afin de bien visualiser qui fait quoi, reprenons notre schéma en mettant des noms sur nos blocs :



Outils et environnement de développement

La création d'une application web avec Java EE s'effectue généralement à l'aide d'un **Environnement de Développement Intégré**, très souvent raccourci à l'anglaise *IDE* : c'est un logiciel destiné à faciliter grandement le développement dans son ensemble. S'il est possible pour certains projets Java de s'en passer, en ne se servant que d'un éditeur de texte pour écrire le code et d'une invite de commandes pour mettre en place l'application, ce n'est sérieusement plus envisageable pour la création d'une application web complexe. Nous allons donc dans ce chapitre apprendre à en utiliser un, et y intégrer notre serveur d'applications.

Si malgré mes conseils, votre côté *extrémiste-du-bloc-notes* prend le dessus et que vous souhaitez tout faire à la main, ne vous inquiétez pas, je prendrai le temps de détailler ce qui se trame en coulisses lorsque c'est important !

L'IDE Eclipse

Présentation

J'utiliserai l'IDE **Eclipse** tout au long de ce cours. Ce n'est pas le seul existant, c'est simplement celui que je maîtrise le mieux. Massivement utilisé en entreprise, c'est un outil puissant, gratuit, libre et multiplateforme. Les avantages d'un IDE dans le développement d'applications web Java EE sont multiples, et sans toutefois être exhaustif en voici une liste :

- intégration des outils nécessaires au développement et au déploiement d'une application ;
- paramétrage aisé et centralisé des composants d'une application ;
- multiples moyens de visualisation de l'architecture d'une application ;
- génération automatique de portions de codes ;
- assistance à la volée lors de l'écriture de code ;
- outils de débogage...

Version

Comme vous pouvez le constater en vous rendant sur la page de téléchargements du site, Eclipse est décliné en plusieurs versions. Nous avons bien entendu besoin de la version spécifique au développement Java EE :

Cliquez sur "Eclipse IDE for Java EE Developers", puis choisissez et téléchargez la version correspondant à votre système d'exploitation :

Une fois le logiciel téléchargé, installez-le de préférence dans un répertoire situé directement à la racine de votre disque dur, et dont le titre ne contient ni espaces ni caractères spéciaux. Ce n'est pas une obligation mais un simple conseil, qui vous évitera bien des ennuis par la suite. Je l'ai pour ma part installé dans un répertoire que j'ai nommé **eclipse** et placé à la racine de mon disque dur : on peut difficilement faire plus clair.

Pour ceux d'entre vous qui ont déjà sur leur poste une version "Eclipse for Java developers" et qui ne souhaitent pas télécharger et installer la version pour Java EE, sachez qu'il est possible - mais bien moins agréable - d'y ajouter des plugins afin d'y reproduire l'intégration de l'environnement Java EE. Si vous y tenez, voici les étapes à suivre depuis votre fenêtre Eclipse :

1. Allez dans Help > Install New Software.
2. Choisissez le site "Indigo - http://download.eclipse.org/releases/indigo".
3. Déroulez "Web, XML, and Java EE Development".
4. Cochez alors "JST Server Adapters" et "JST Server Adapters Extentions".

Ça résoudra une partie des problèmes que vous pourriez rencontrer par la suite en suivant ce cours.
Notez bien toutefois que je vous conseille de ne pas procéder ainsi, et de repartir d'une version vierge d'Eclipse pour Java EE.

Le serveur Tomcat

Présentation

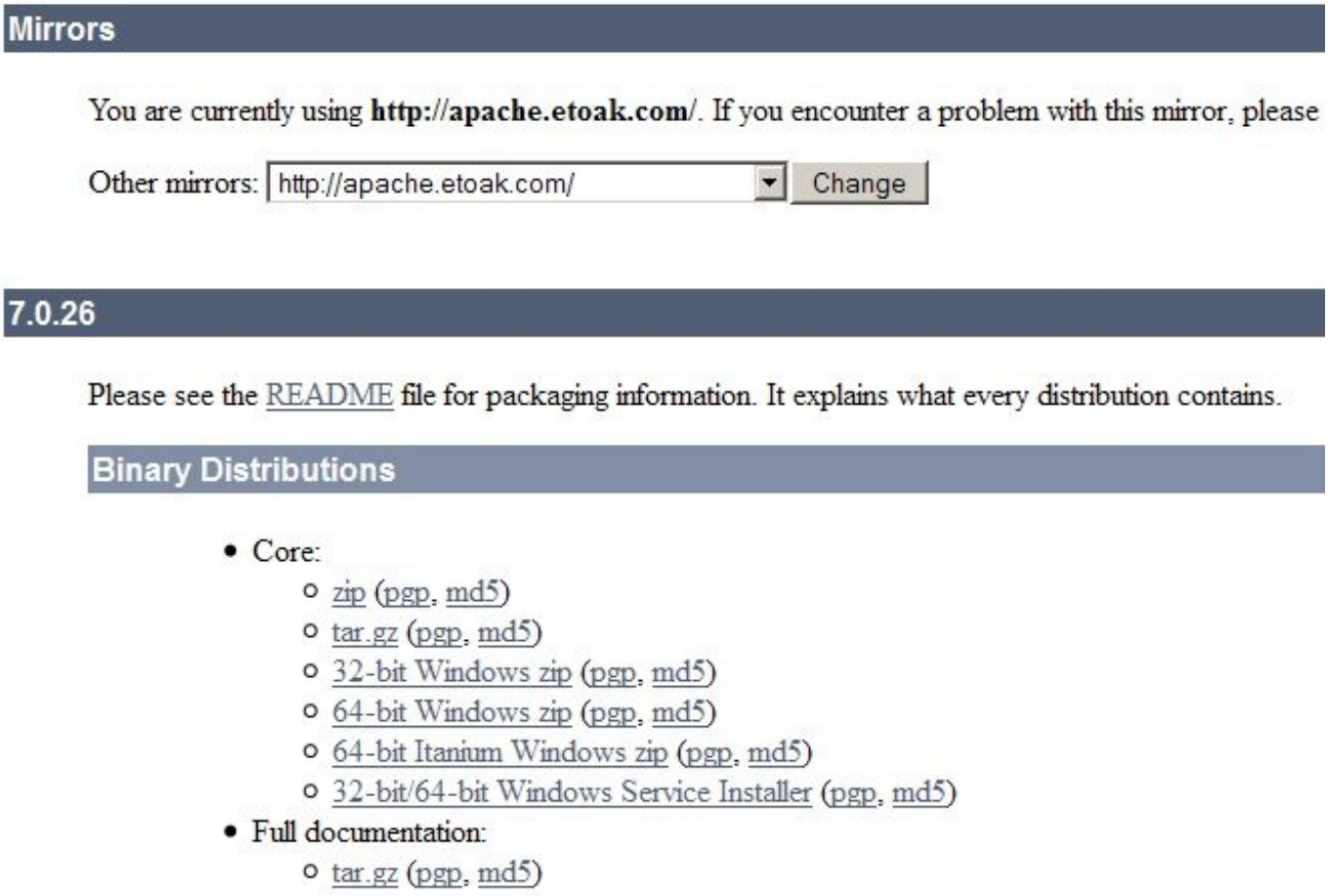
Nous l'avons découvert dans le second chapitre : pour faire fonctionner une application web Java EE, nous avons besoin de mettre en place un **serveur d'applications**. Il en existe beaucoup sur le marché : j'ai pour le début de ce cours choisi d'utiliser **Tomcat**, car c'est un serveur léger, gratuit, libre, multiplateforme et assez complet pour ce que nous allons aborder. On le rencontre d'ailleurs très souvent dans des projets en entreprise, en phase de développement comme en production.

Si vous souhaitez vous renseigner sur les autres serveurs existants et sur leurs différences, vous savez où chercher. Wikipédia en propose par ailleurs une liste non exhaustive.

Pour information, sachez que Tomcat tire sa légèreté du fait qu'il n'est en réalité que l'assemblage d'un **serveur web** (gestion des requêtes/réponses HTTP) et d'un **conteneur web** (nous parlerons en temps voulu de **conteneur de servlets**, et reviendrons sur ce que cela signifie concrètement). Pour le moment, retenez simplement que ce n'est pas un serveur d'applications Java EE au sens complet du terme, car il ne respecte pas entièrement ses spécifications et ne supporte pas toutes ses technologies.

Installation

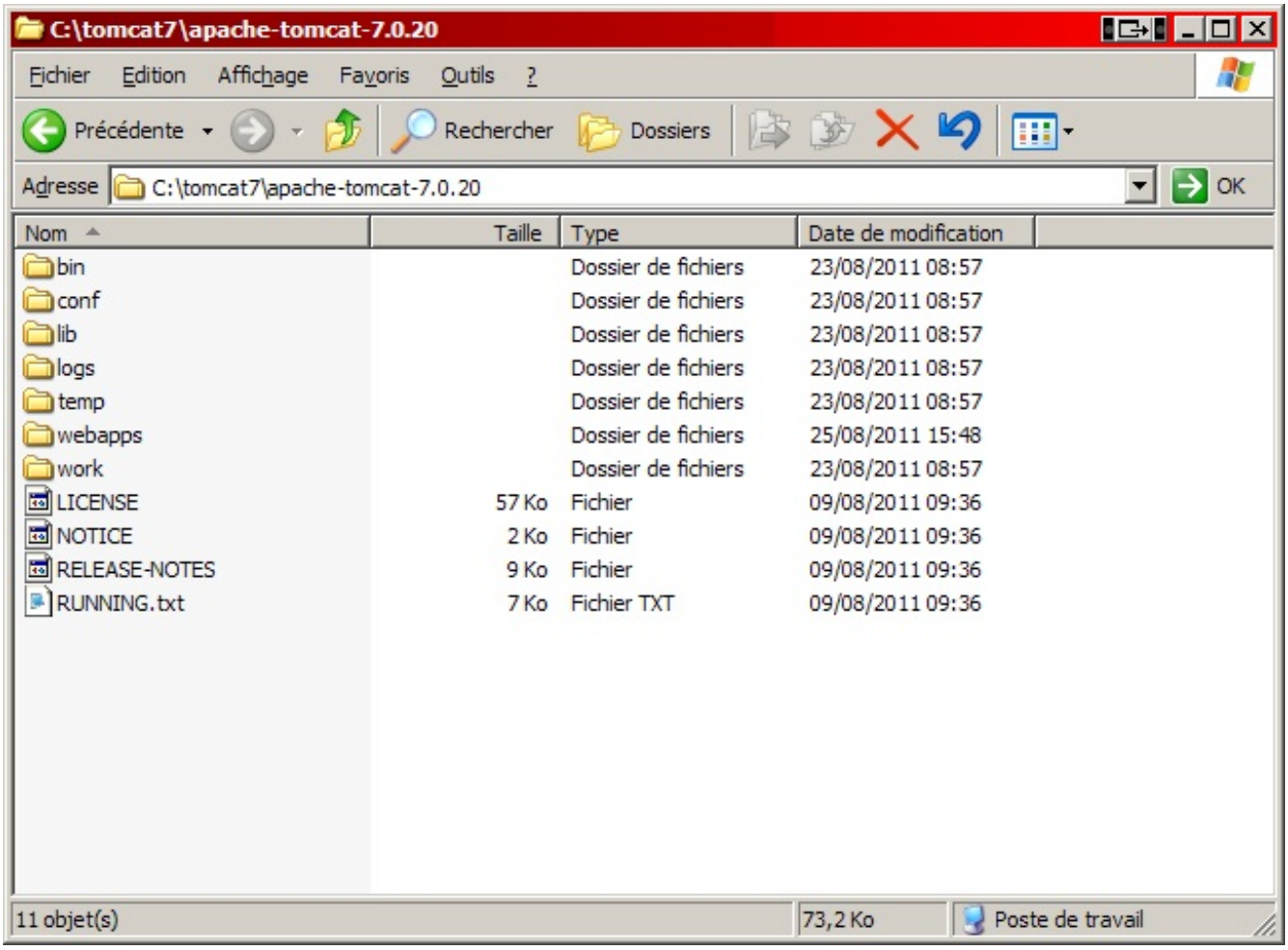
Nous allons utiliser la dernière version en date à ce jour, à savoir **Tomcat 7.0**. Rendez-vous sur la page de téléchargement de Tomcat, puis choisissez et téléchargez la version correspondant à votre système d'exploitation :



Page de téléchargement de Tomcat

Sous Windows

Récupérez la dernière version *Core* au format zip, puis décompressez son contenu dans le répertoire où vous souhaitez installer Tomcat. Au sujet du répertoire d'installation, même conseil que pour Eclipse : choisissez un chemin dont les noms de dossiers ne comportent pas d'espaces : pour ma part, je l'ai placé dans un dossier nommé **tomcat7** à la racine de mon disque. Un dossier nommé **apache-tomcat-7.0.xx** (les deux derniers numéros changeant selon la version que vous utiliserez) contient alors l'installation. Pour information, ce dossier est souvent référencé dans les cours et documentations par l'appellation *Tomcat Home*. Voici ce que j'obtiens sur mon poste :



Répertoire d'installation de Tomcat

Dans ce répertoire d'installation de Tomcat, vous trouverez un dossier nommé **webapps** : c'est ici que seront stockées par défaut vos applications. Pour ceux d'entre vous qui souhaiteraient jeter un oeil à ce qui se passe derrière les rideaux, vous trouverez dans le dossier **conf** les fichiers suivants :

- **server.xml** : contient les éléments de configuration du serveur.
- **context.xml** : contient les directives communes à toutes les applications web déployées sur le serveur.
- **tomcat-users.xml** : contient entre autres l'identifiant et le mot de passe permettant d'accéder à l'interface d'administration de votre serveur Tomcat.
- **web.xml** : contient les paramètres de configuration communs à toutes les applications web déployées sur le serveur.

Je ne m'attarde pas sur le contenu de chacun de ces fichiers : nous y effectuerons des modifications indirectement via l'interface d'Eclipse au cours des exemples à venir. Je n'aborde volontairement pas dans le détail la configuration fine d'un serveur Tomcat, ceci méritant sans aucun doute un tutoriel à part entière. Vous pouvez néanmoins trouver beaucoup d'informations sur Tomcat's Corner, et bien que ce site traite des versions plus anciennes, la plupart des concepts présentés sont toujours d'actualité. Je vous renvoie bien sûr à la documentation officielle de Tomcat 7, pour plus d'exactitude.

Sous Linux

Récupérez la dernière version Core au format tar.gz : une archive nommée apache-tomcat-7.0.xx.tar.gz est alors enregistrée sur votre poste, où xx correspond à la sous-version courante. Au moment où j'écris ces lignes, la version est la 7.0.20 : **pensez à adapter les commandes qui suivent à la version que vous téléchargez**. Décompressez ensuite son contenu dans le répertoire où vous souhaitez installer Tomcat. Par exemple :

Code : Console

```
cd /usr/local
```

```
mkdir tomcat
cd /usr/local/tomcat
cp ~/apache-tomcat-7.0.20.tar.gz .
tar -xvzf apache-tomcat-7.0.20.tar.gz
```

Un dossier nommé apache-tomcat-7.0.20 contient alors l'installation. Pour information, ce dossier est souvent référencé dans les cours et documentations par l'appellation *Tomcat Home*. Vérifiez alors que l'installation s'est bien effectuée :

Code : Console

```
cd /usr/local/tomcat/apache-tomcat-7.0.20
cd bin
./version.sh
```

Ce script montre alors que Tomcat 7.0 a été installé avec succès sur votre distribution Linux :

Code : Console

```
Server version: Apache Tomcat/7.0.20
Server built:   Aug 28 2011 15:13:02
Server number:  7.0.20.0
OS Name:        Linux
```

Sous Mas OS

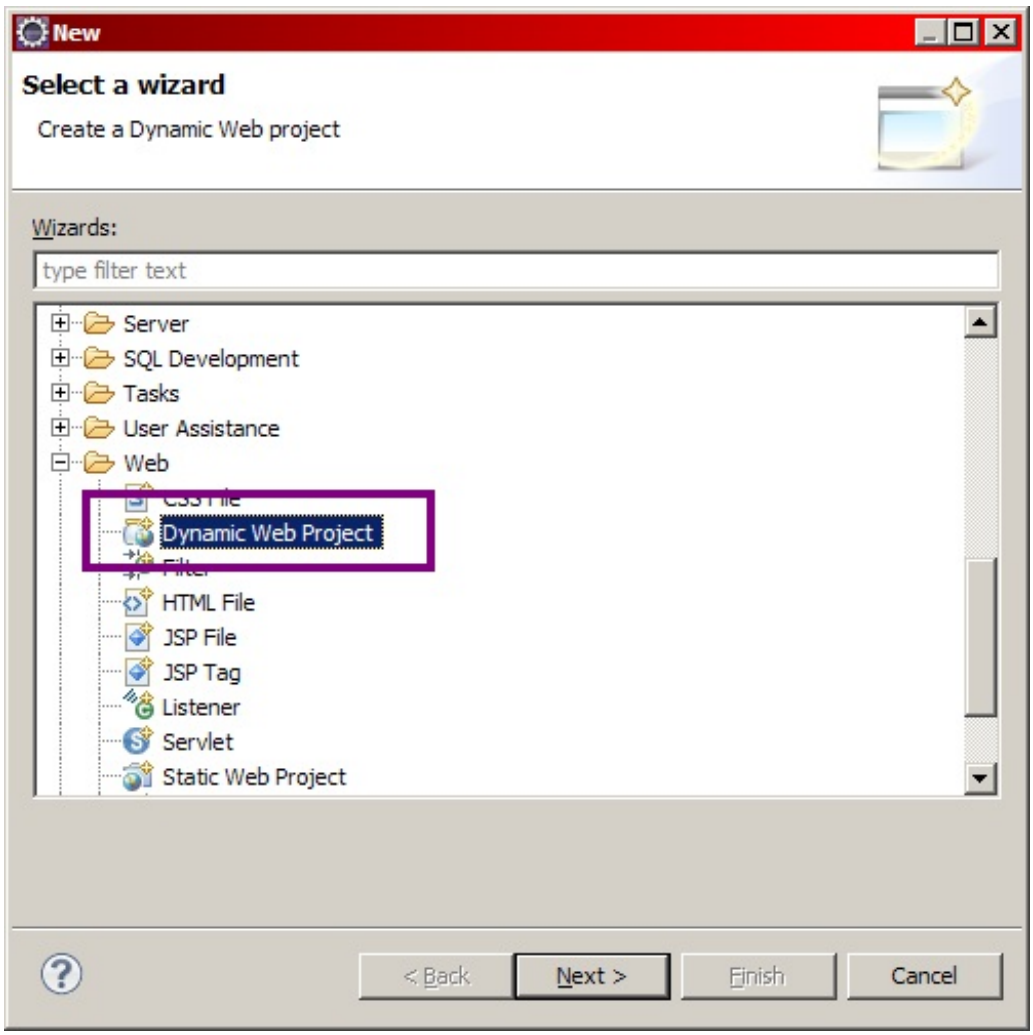
Je n'ai malheureusement pas à ma disposition de machine tournant sous Mac OS. Si vous êtes un aficionado de la marque à la pomme, voici deux liens qui expliquent comment installer Tomcat 7 sur OS X :

- Installation de Tomcat 7.0.x sur Mac OS X
- Installation sous Mac OS X Snow Leopard

Bonne lecture, et n'hésitez pas à me prévenir d'éventuels erreurs ou changements dans le procédé présenté, je modifierai cette section du chapitre en conséquence.

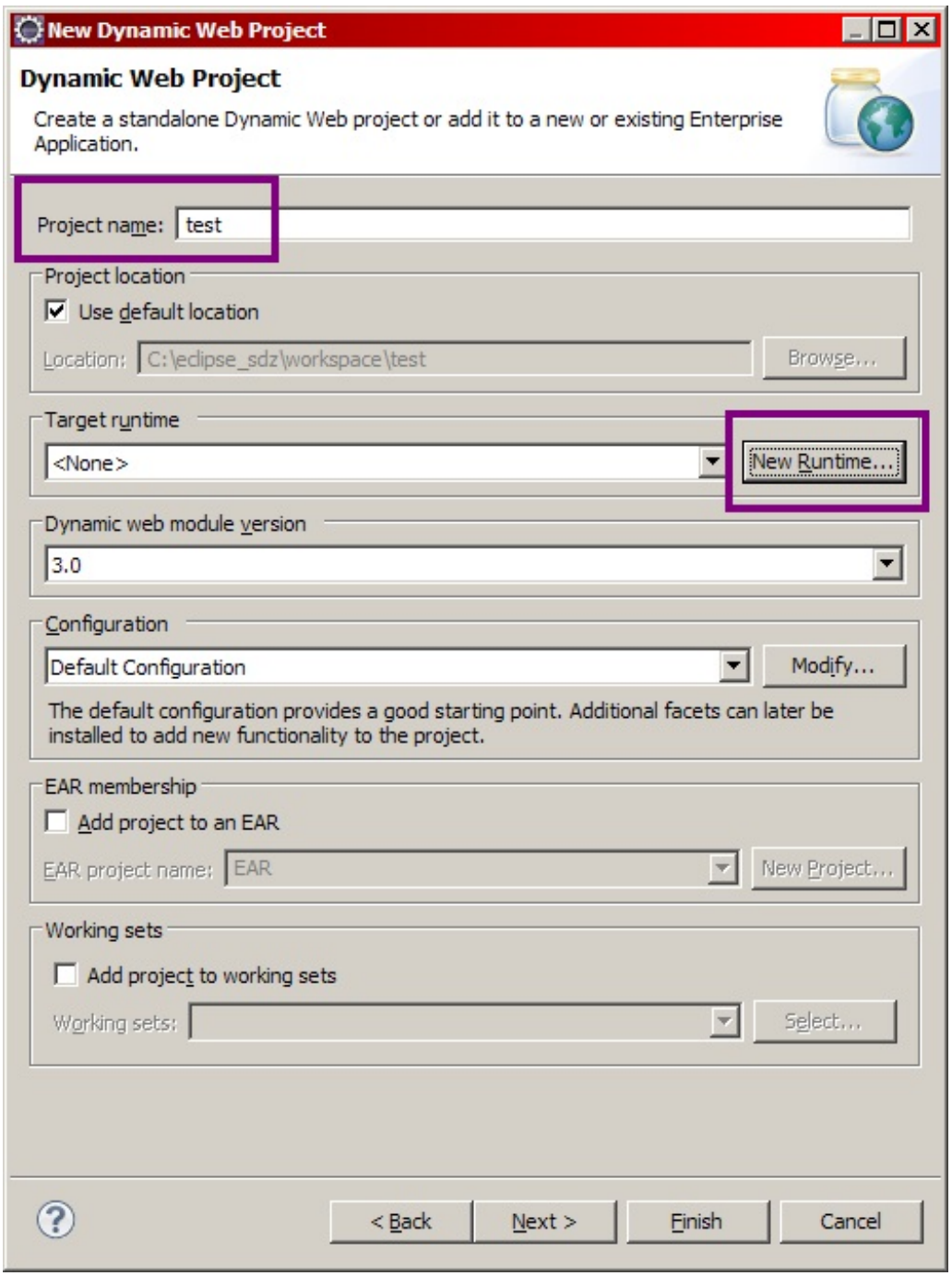
Création du projet web avec Eclipse

Depuis Eclipse, suivez le chemin suivant : **File > New > Project...**
Ceci peut d'ailleurs être raccourci en tapant au clavier Ctrl + N :



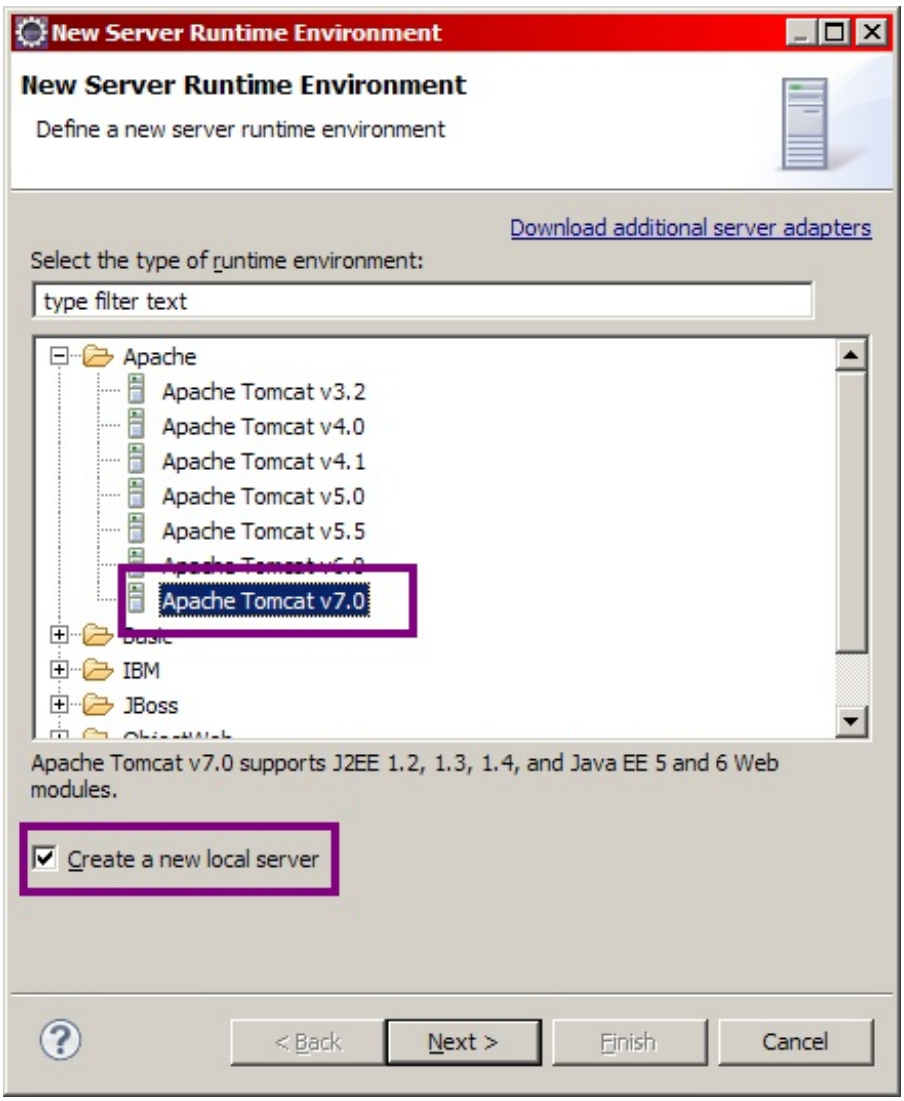
Nouveau projet web sous Eclipse

Sélectionnez alors **Dynamic Web Project** comme le montre l'image ci-dessus, puis cliquez sur `Next >`. J'appelle ici mon projet **test**. Remarquez ensuite le passage concernant le serveur :



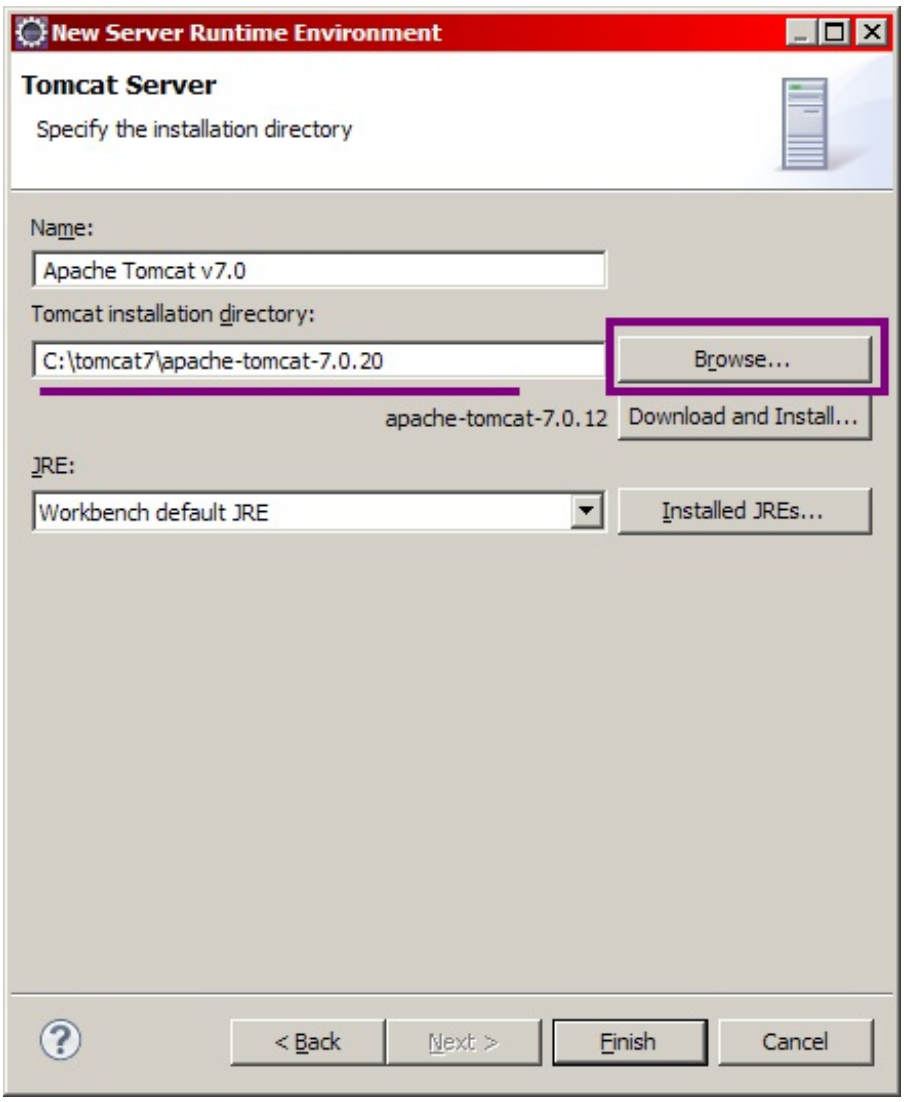
Mise en place de Tomcat - Étape 1

Cliquez sur le bouton `New Runtime...` et sélectionnez alors Apache Tomcat 7.0 dans la liste des possibilités :



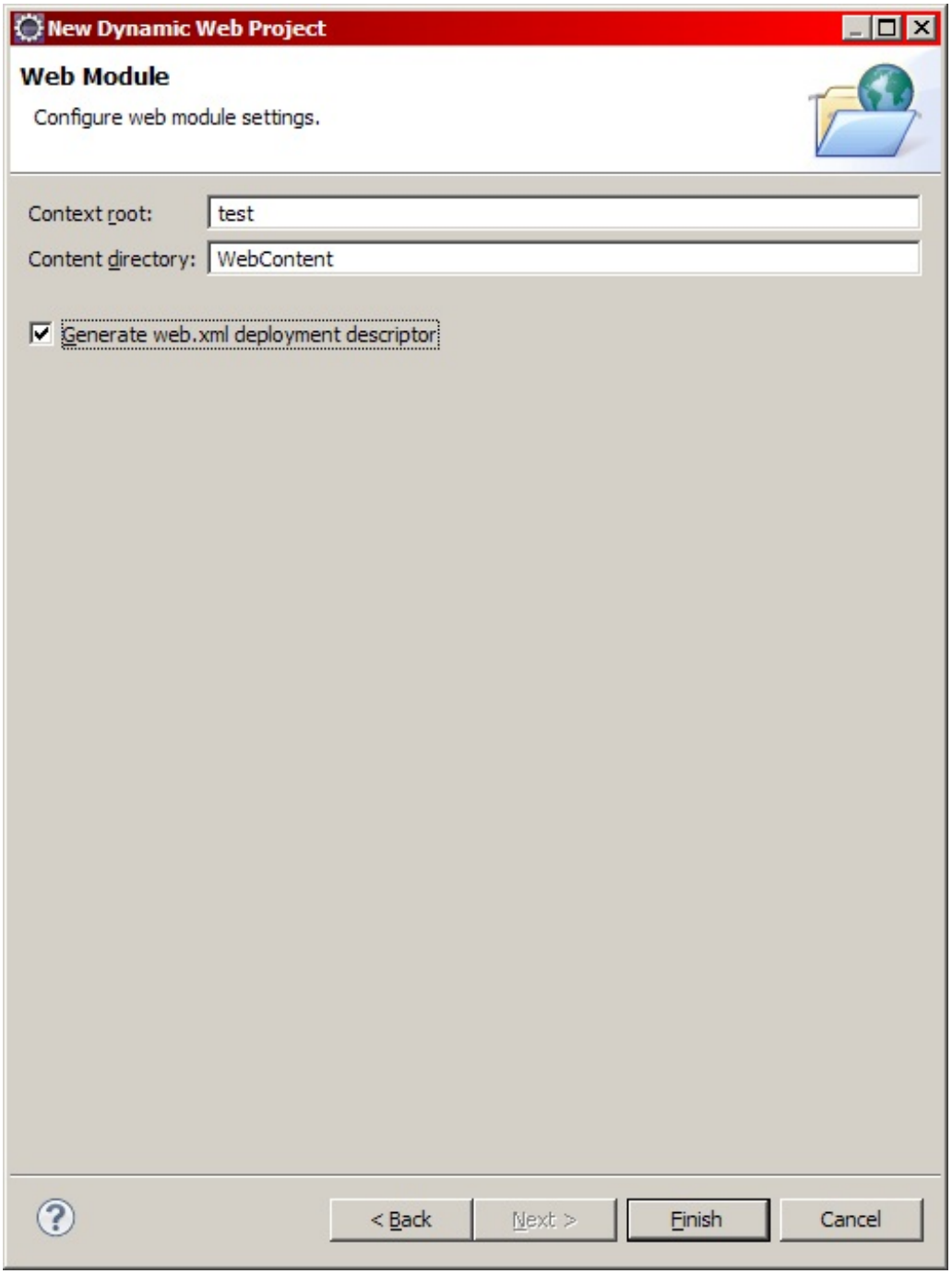
Mise en place de Tomcat - Étape 2

Cochez la case comme indiqué ci-dessus, ce qui signifie que nous allons en plus du projet créer localement une nouvelle instance d'un serveur, instance que nous utiliserons par la suite pour déployer notre application. Cliquez ensuite sur `Next >` et remplissez correctement les informations relatives à votre installation de Tomcat en allant chercher le répertoire d'installation de Tomcat sur votre poste. Les champs devraient alors ressembler à ceci, le répertoire d'installation et le numéro de version de Tomcat 7 pouvant être différents chez vous selon ce que vous avez choisi et installé :



Mise en place de Tomcat - Étape 3

Validez alors en cliquant sur `Finish`, puis cliquez deux fois sur `Next >`, jusqu'à obtenir cette fenêtre :



Mise en place de Tomcat - Étape 4

Avant d'aller plus loin, il est nécessaire de parler **contexte** !

Souvenez-vous, je vous ai déjà parlé d'un fichier **context.xml** associé à toutes les applications. Pour permettre plus de souplesse, il est possible de spécifier un contexte propre à chaque *webapp*. Comme je vous l'ai déjà dit, ces applications web sont empiriquement contenues dans le dossier... **webapps** de votre *Tomcat Home*. C'est ici que, par défaut, Tomcat ira chercher les applications qu'il doit gérer et déployer. Jusque là, vous suivez...

Le souci, et certains d'entre vous l'auront peut-être déjà compris, c'est que notre projet à nous, créé depuis Eclipse, se trouve dans un répertoire de notre *workspace* Eclipse : il n'est pas du tout dans ce fameux répertoire *webapps* de Tomcat. Pour que notre serveur prenne en compte notre future application, il va donc falloir arranger le coup ! Plusieurs solutions s'offrent alors à nous :

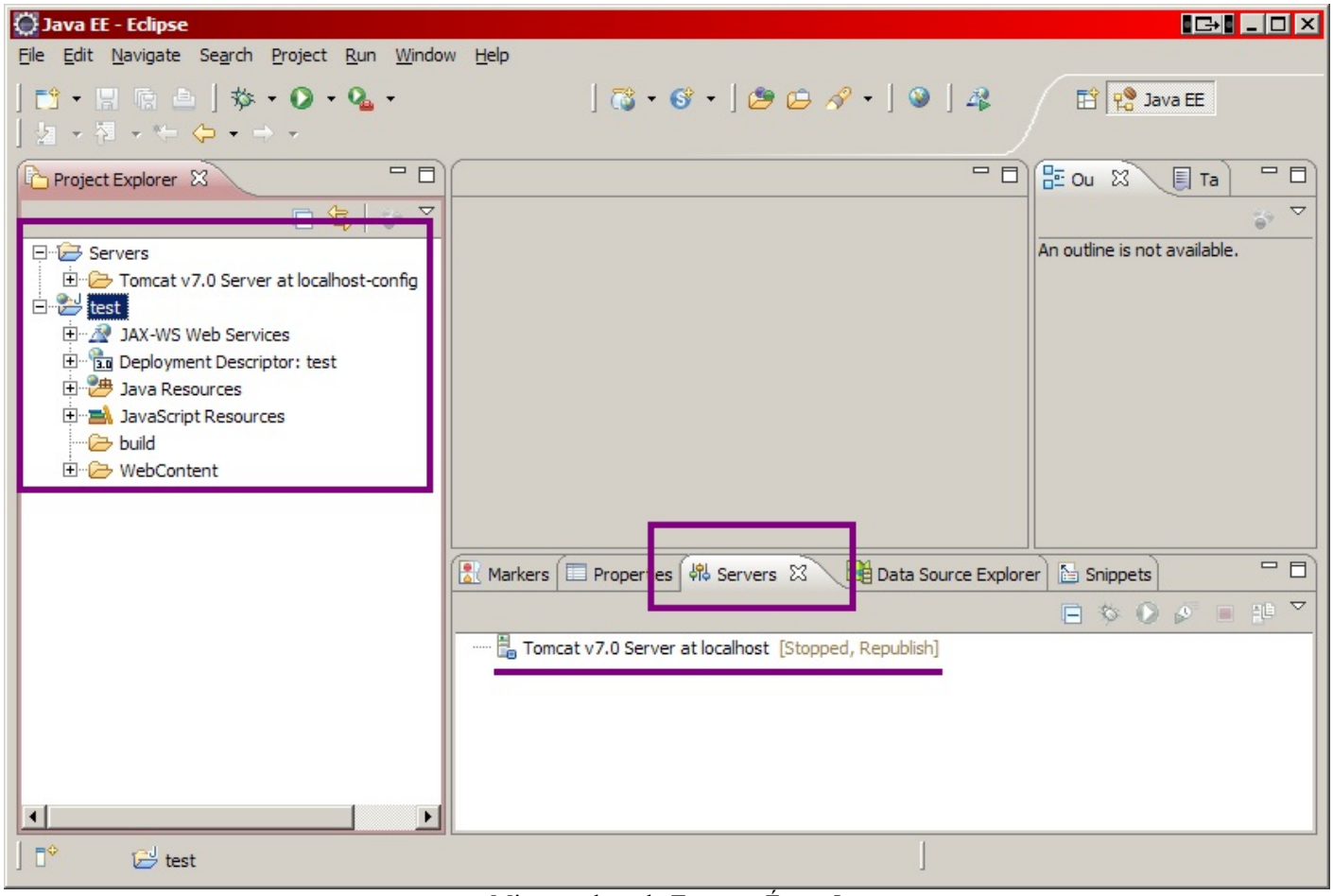
- créer un répertorie, du même nom que notre projet sous Eclipse, directement dans le dossier *webapps* de Tomcat, et y copier-coller nos fichiers, et ce à chaque modification de code ou configuration effectuée ;
- créer un nouveau projet depuis Eclipse, en utilisant directement le répertoire *webapps* de votre *Tomcat Home* comme *workspace* Eclipse ;
- modifier le **server.xml** ou le **context.xml** de votre Tomcat, afin qu'il sache où chercher ;

- utiliser les propriétés d'un projet Web dynamique sous Eclipse.

Étant donnée la dernière fenêtre qui s'est affichée, vous avez probablement deviné sur quelle solution notre choix va se porter. Je vous conseille bien évidemment ici d'utiliser la quatrième et dernière solution. Conservez le nom de vote projet sous Eclipse comme contexte de déploiement sur votre serveur Tomcat ("*Context root*" sur l'image précédente), afin de rester cohérent. Utiliser les paramètres ci-dessus permet alors de ne pas avoir à modifier vous-même le contexte de votre serveur, ou encore de ne pas avoir à utiliser le dossier *webapps* de votre serveur Tomcat en guise de *workspace*. Toute modification sur vos futures pages et classes sera ainsi automatiquement prise en compte par votre serveur Tomcat, qui s'occupera de recharger le contexte à chaque modification sauvegardée, lorsque le serveur sera lancé.

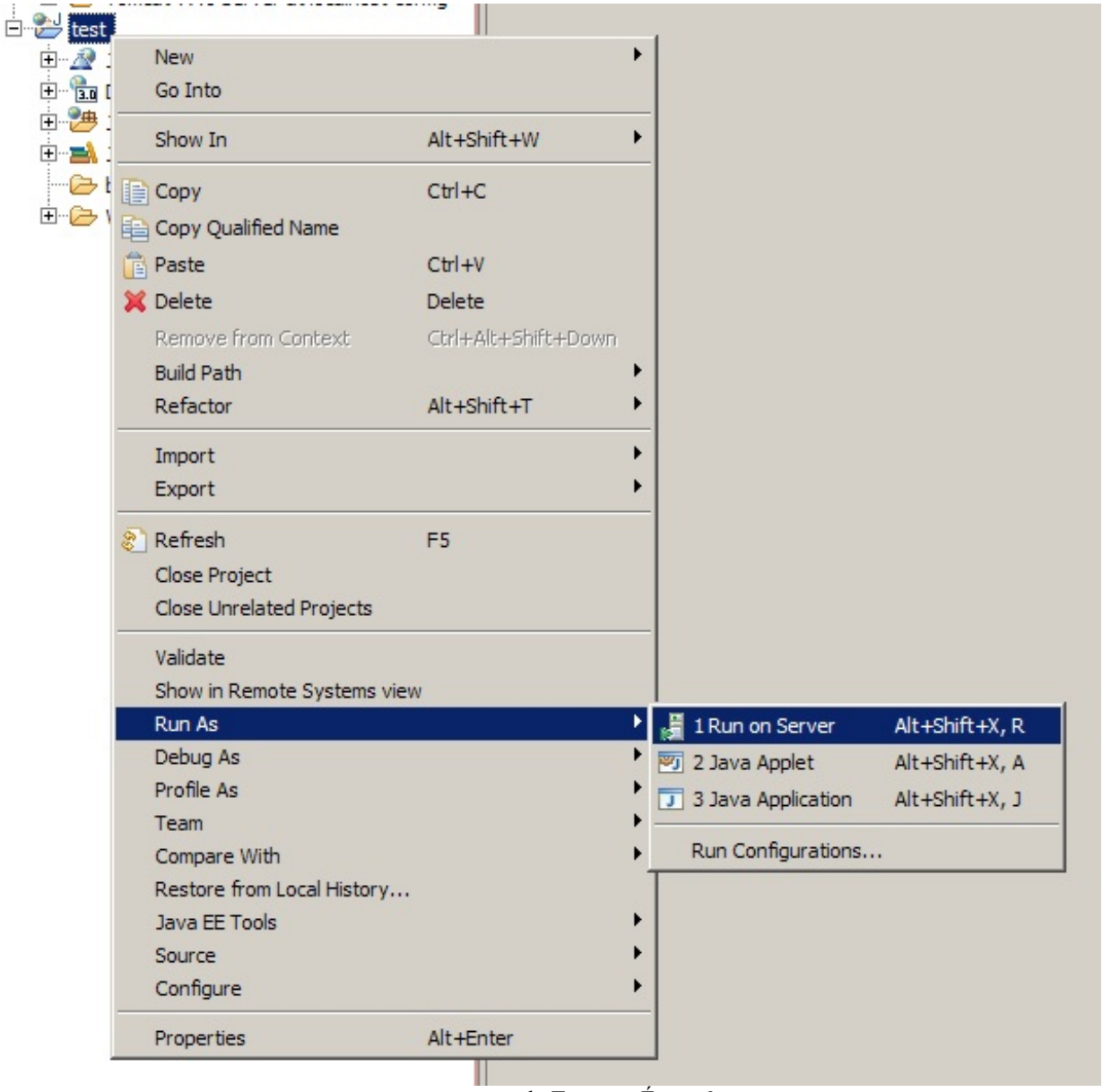
Comme diraient les têtes-à-claques, *isn't it amazing?*

Voici maintenant ce à quoi doit ressembler votre fenêtre Eclipse :



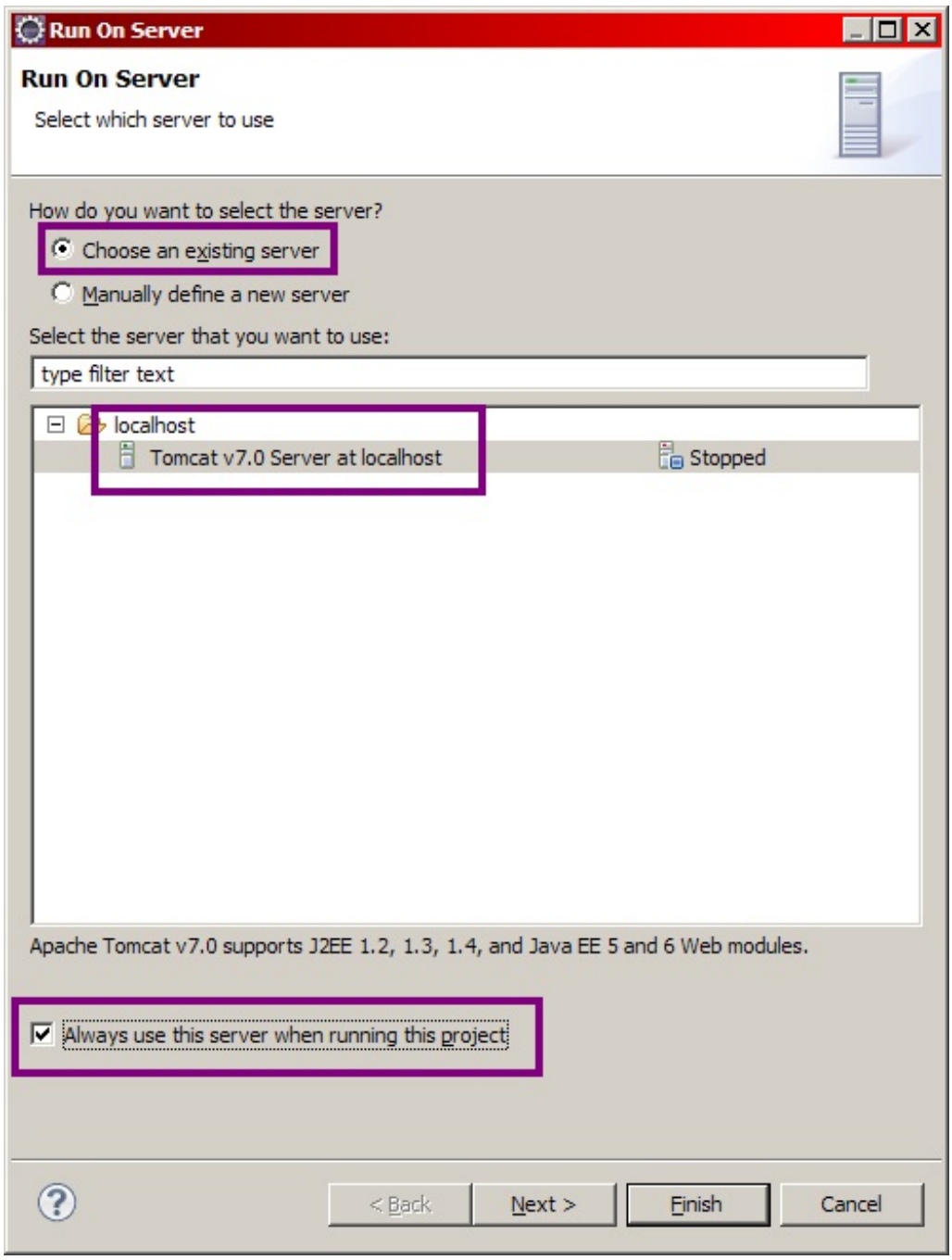
Mise en place de Tomcat - Étape 5

Vous noterez l'apparition d'une entrée **Tomcat v7.0** dans l'onglet **Servers**, et de l'arborescence de votre projet **test** dans le volet de gauche.
Faites maintenant un clic droit sur le titre de votre projet dans l'arborescence Eclipse, et suivez **Run As > Run on Server** :



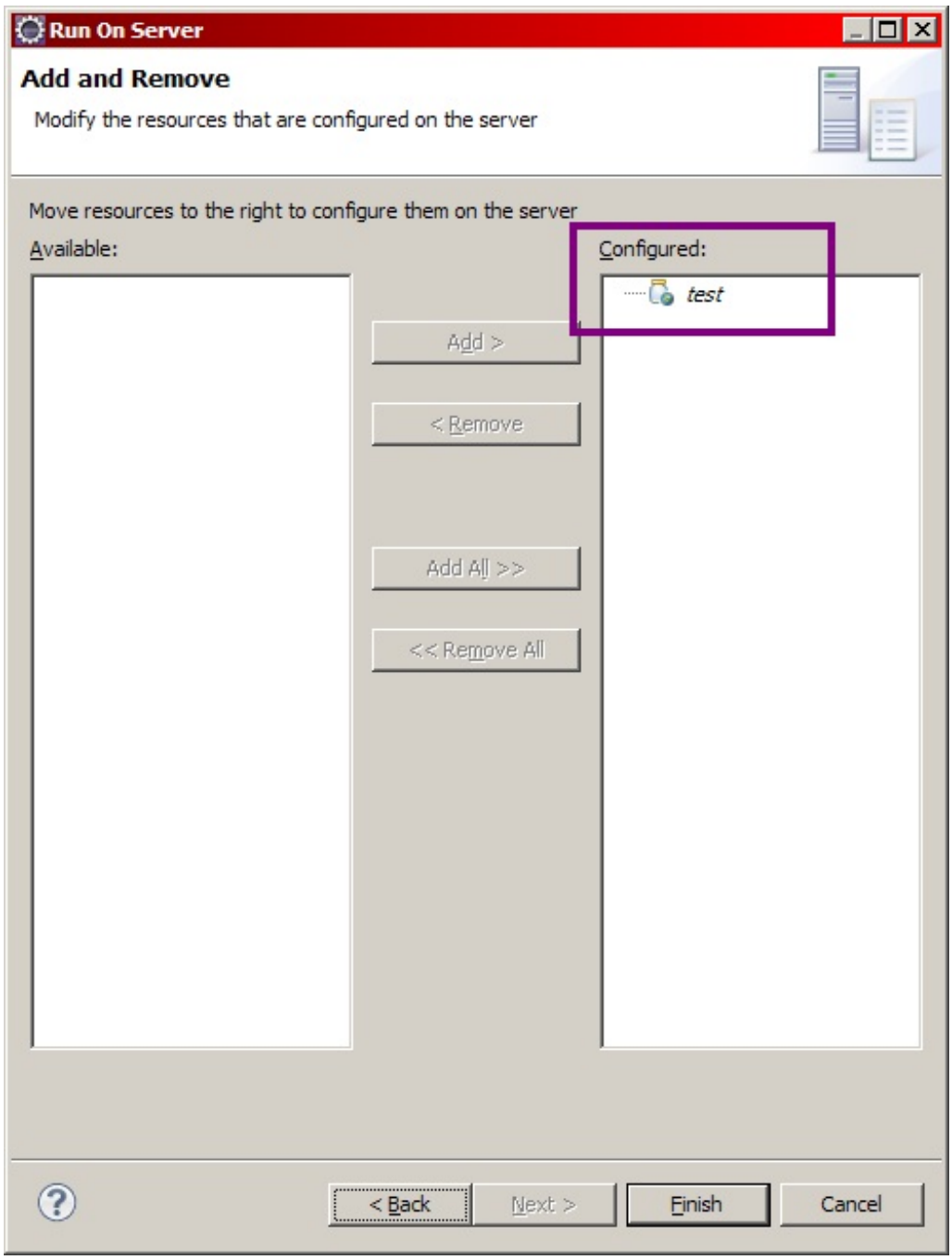
Mise en place de Tomcat - Étape 6

Dans la fenêtre qui s'ouvre alors, nous allons sélectionner le serveur Tomcat que nous venons de mettre en place lors de la création de notre projet web, et préciser que l'on souhaite associer par défaut notre projet à ce serveur :



Mise en place de Tomcat - Étape 7

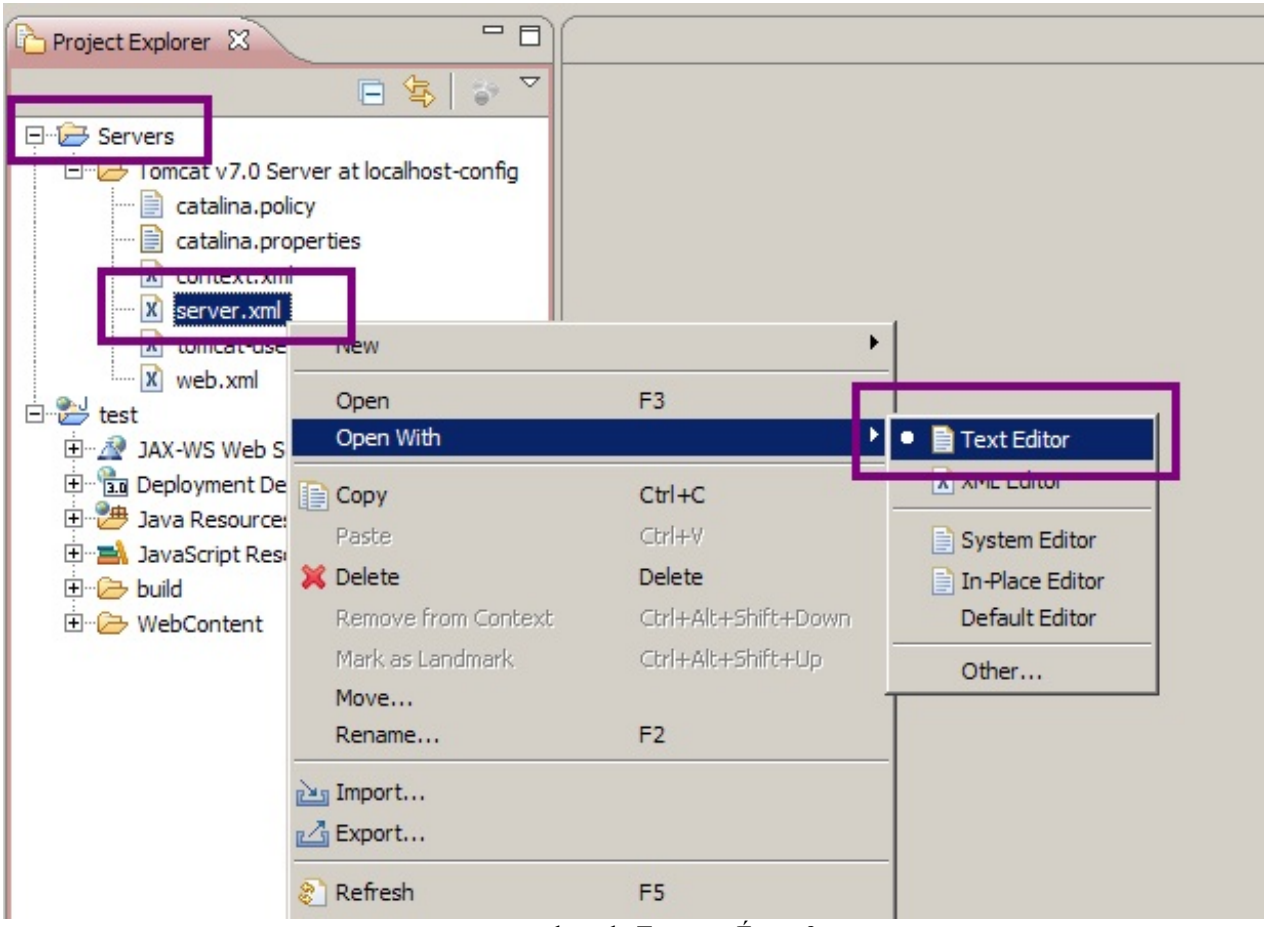
Cliquez alors sur `Next >`, puis vérifiez que votre nouveau projet est bien pris en compte par votre serveur :



Mise en place de Tomcat - Étape 8

Validez enfin en cliquant sur `Finish`, et voilà la mise en place de votre projet et de son serveur **terminée** !

Pour la petite histoire, une section est ajoutée dans le fichier **server.xml** de votre instance de Tomcat, qui est maintenant accessible depuis le dossier **Servers** de votre arborescence Eclipse :



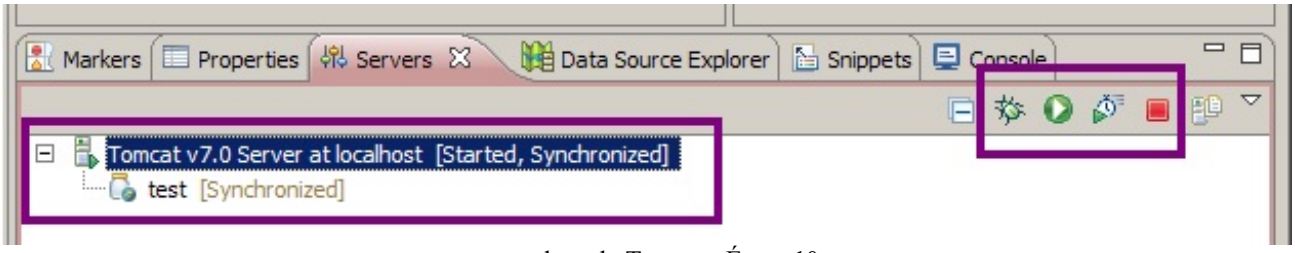
Mise en place de Tomcat - Étape 9

Si vous êtes curieux, éditez-le ! Vous verrez qu'il contient effectivement en fin de fichier une section de ce type :

Code : XML

```
<Context docBase="test" path="/test" reloadable="true"
source="org.eclipse.jst.jee.server:test"/>
```

Dorénavant, pour piloter votre serveur Tomcat il vous suffira de vous rendre dans l'onglet **Servers** en bas de votre fenêtre Eclipse, et d'utiliser un des boutons selon le besoin (redémarrage, arrêt, debug) :



Mise en place de Tomcat - Étape 10

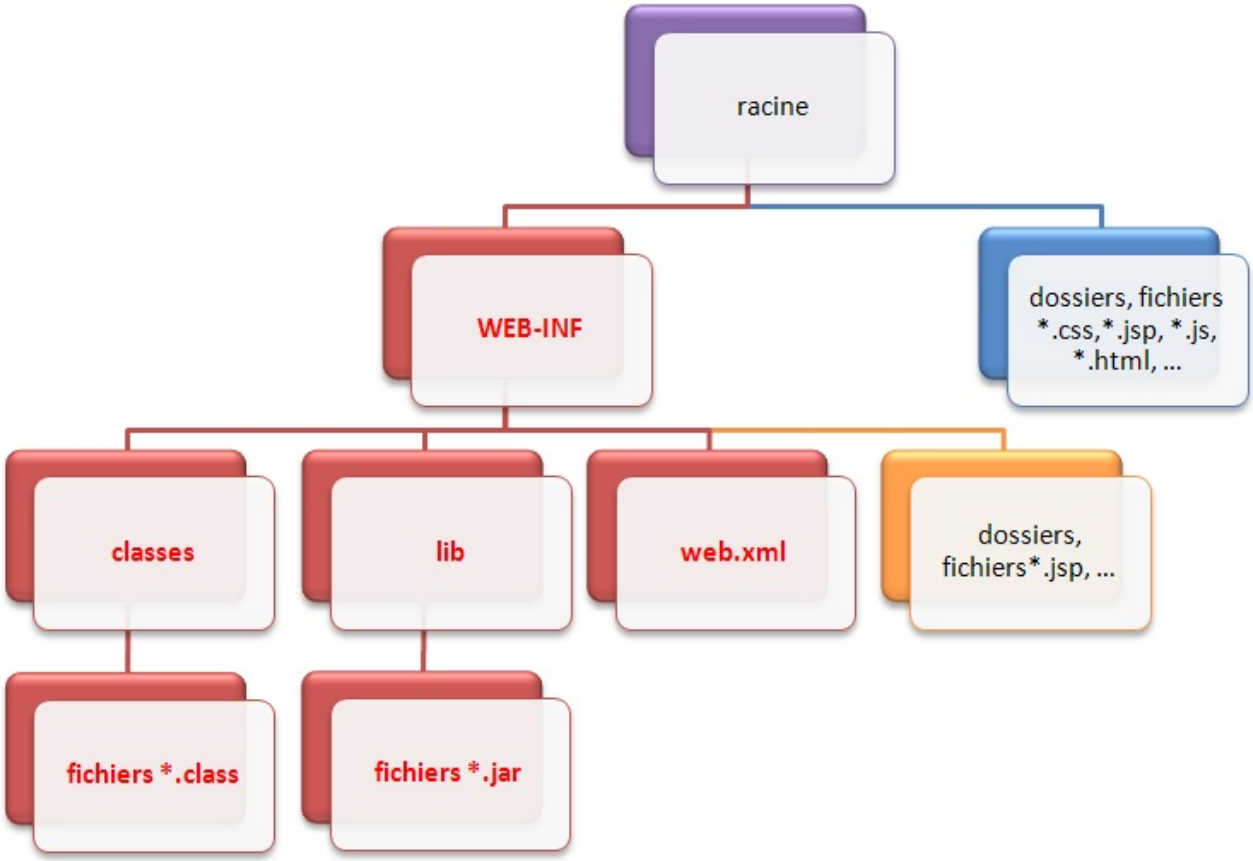
Sachez pour finir que cette manipulation n'est pas limitée à Tomcat uniquement. Vous pouvez utiliser d'autres types de serveurs, cela ne pose pas de problèmes. De même, une fois que vous avez correctement paramétré un serveur Tomcat depuis Eclipse, vous n'êtes pas forcés de recréer localement un nouveau serveur pour chacun de vos projets, vous pouvez très bien réutiliser la même instance de Tomcat en y déployant plusieurs applications web différentes.

Si vous êtes arrivés jusqu'ici, c'est que votre instance de serveur Tomcat est fonctionnelle et que vous pouvez la piloter depuis Eclipse. Voyons maintenant où placer notre premier essai, et comment y accéder.

Structure d'une application Java EE

Structure standard

Toute application web Java EE doit respecter une structure de dossiers standard, qui est définie dans les spécifications de la plate-forme :



Structure des fichiers d'une application web JSP/Servlet

Quelques précisions :

- La racine de l'application, en violet sur le schéma, est le dossier qui porte le nom de votre projet et qui contient l'intégralité des dossiers et fichiers de l'application.
- Le dossier nommé **WEB-INF** est un dossier spécial. Il doit obligatoirement exister et être placé juste sous la racine de l'application. Il doit à son tour obligatoirement contenir :
  - le fichier de configuration de l'application (web.xml)
  - un dossier nommé **classes**, qui contient à son tour les classes compilées (fichiers .class)
  - un dossier nommé **lib**, qui contient à son tour les bibliothèques nécessaires au projet (archives .jar)

  Bref, tous les dossiers et fichiers marqués en rouge sur le schéma doivent obligatoirement être nommés et placés comme indiqué sur le schéma.
- Les fichiers et dossiers persos placés directement sous la racine, en bleu sur le schéma, sont publics et donc accessibles directement par le client via leurs URL. **(*)**
- Les fichiers et dossiers persos placés sous le répertoire **WEB-INF**, en orange sur le schéma, sont privés et ne sont donc pas accessibles directement par le client. **(*)**

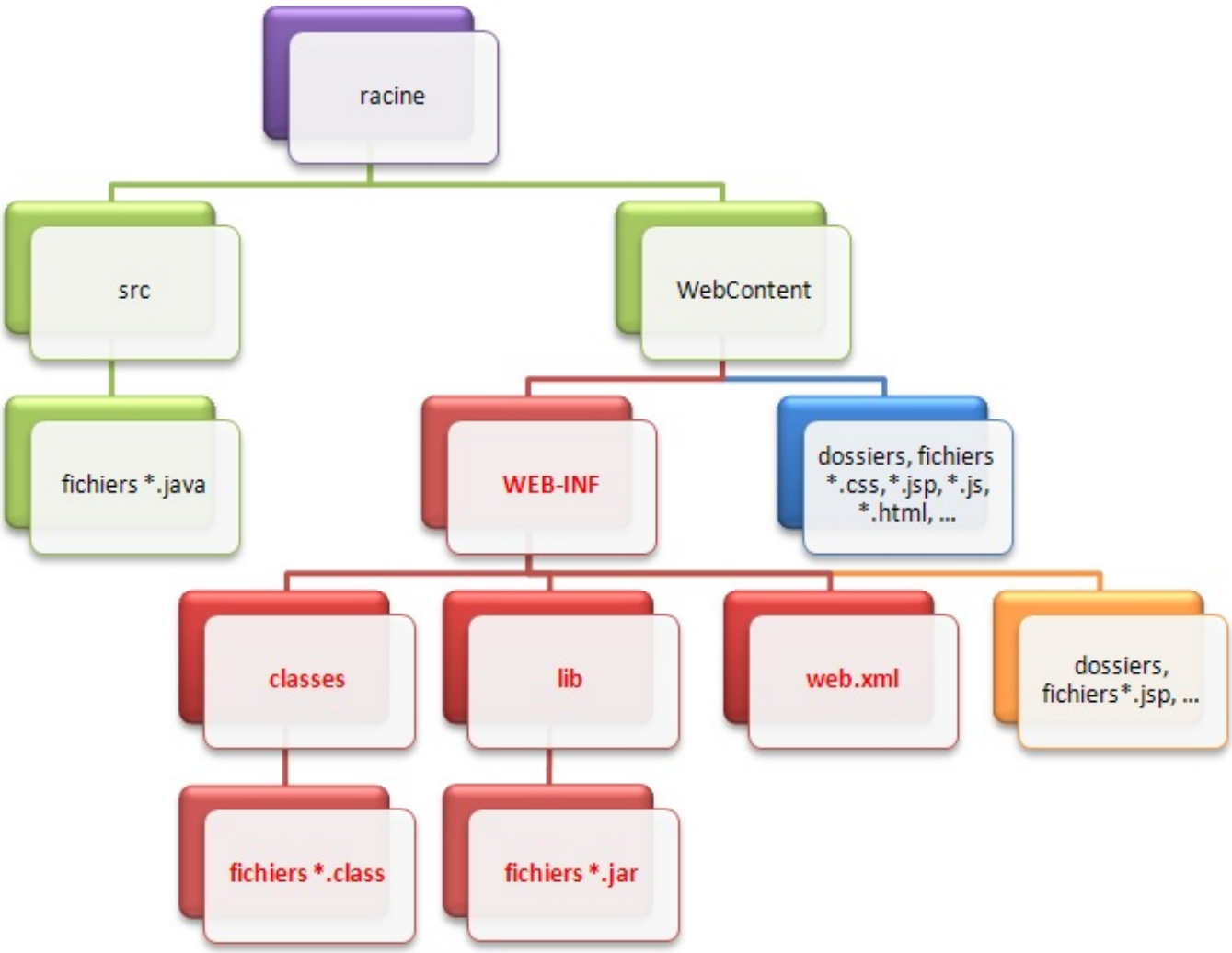
***(*)** Nous reviendrons en temps voulu sur le caractère privé du dossier **WEB-INF**, et sur la distinction avec les dossiers publics.*

Voilà tout concernant la structure officielle : si votre application n'est pas organisée de cette manière, le serveur d'applications ne sera pas capable de la déployer ni de la faire fonctionner correctement.

Votre première page web

Eclipse, ce fourbe !

Ce que vous devez savoir avant de continuer, c'est qu'Eclipse joue souvent au fourbe, en adaptant certaines spécificités à son mode de fonctionnement. En l'occurrence, Eclipse modifie comme suit la structure d'une application Java EE :



Structure des fichiers d'une application web sous Eclipse

Comme vous pouvez le voir en vert sur le schéma, Eclipse déplace la structure standard de l'application sous un dossier nommé **WebContent**, et ajoute sous la racine un dossier **src** qui contiendra le code source de vos classes (les fichiers .java). En outre, je ne les ai pas représentés ici, mais sachez qu'Eclipse ajoute également sous la racine quelques fichiers de configuration qui lui permettront, via une tambouille interne, de gérer correctement l'application !

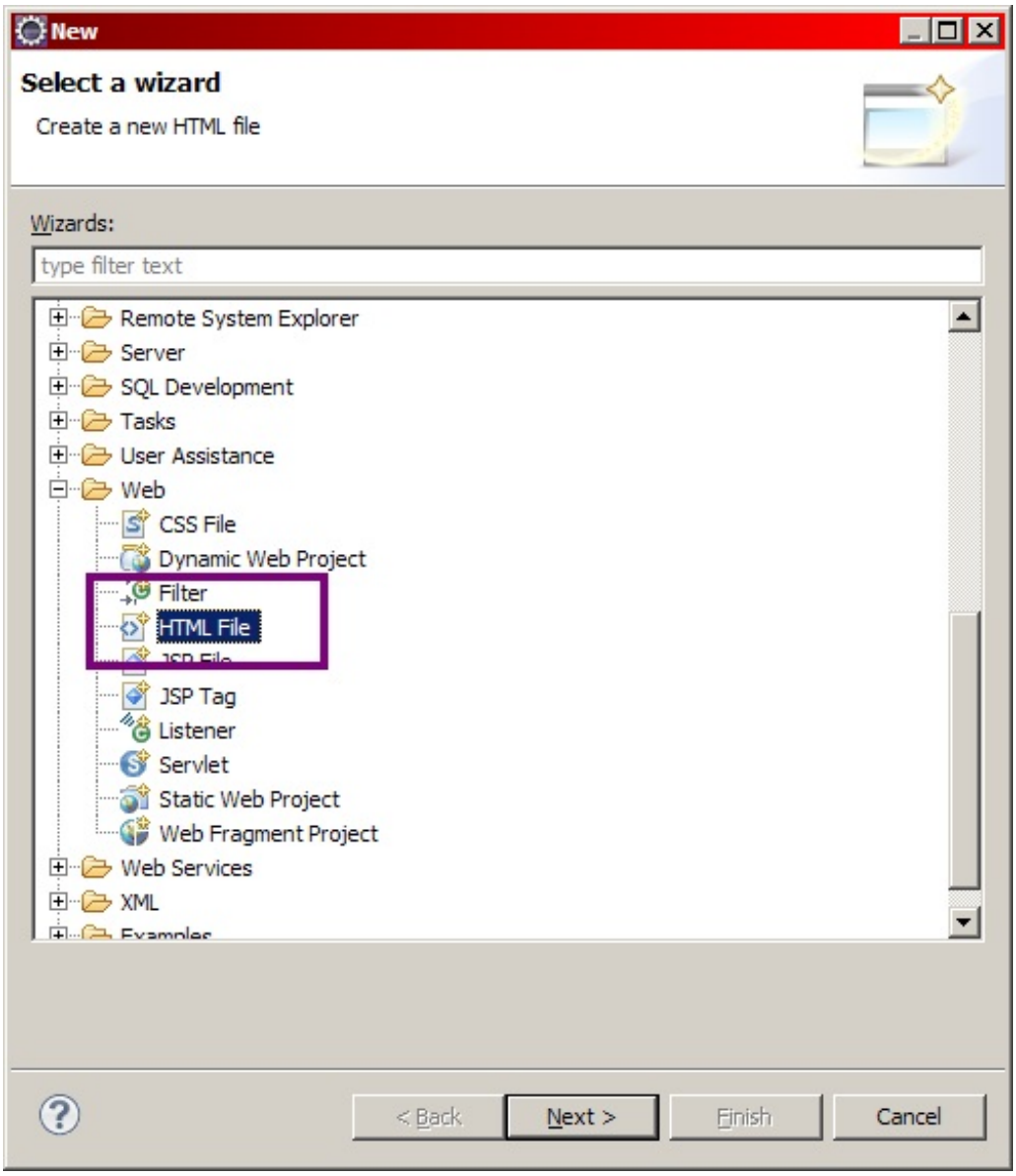
Attendez... Je viens de vous dire que si notre application n'était pas correctement structurée, notre serveur d'applications ne saurait pas la gérer. Si Eclipse vient mettre son nez dans cette histoire, comment notre application va-t-elle pouvoir fonctionner ?

Eh bien comme je viens de vous l'annoncer, Eclipse se débrouille via une tambouille interne pour que la structure qu'il a modifiée soit malgré tout utilisable sur le serveur d'applications que nous lui avons intégré. Ceci implique donc deux choses très importantes :

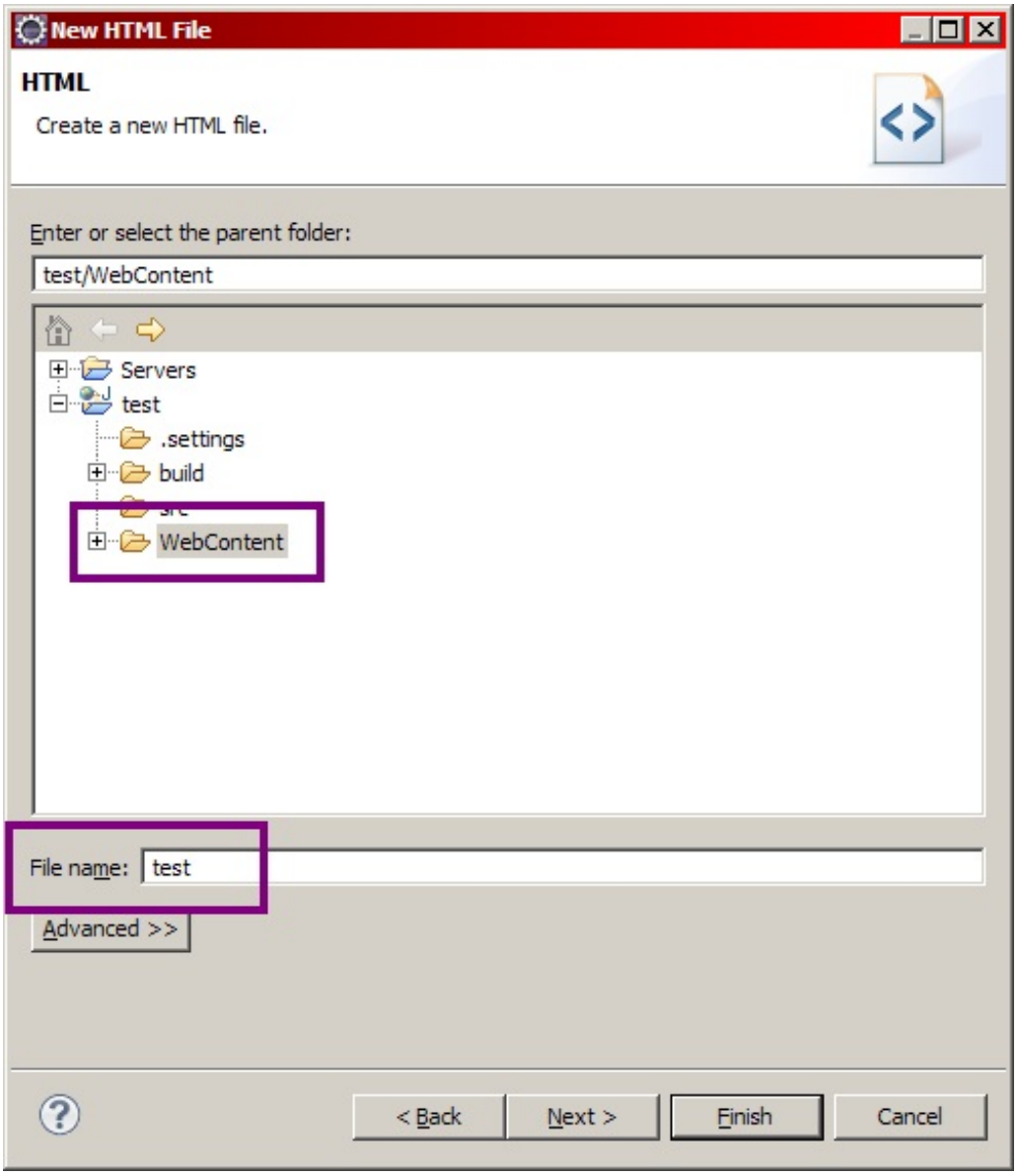
- le dossier **WebContent** n'existe légitimement qu'au sein d'Eclipse. Si vous développez sans IDE, ce répertoire ne doit pas exister et votre application doit impérativement suivre la structure standard présentée précédemment ;
- pour cette même raison, si vous souhaitez utiliser votre application en dehors de l'IDE, il faudra obligatoirement utiliser l'outil d'export proposé par Eclipse. Réaliser un simple copié/collé des dossiers ne fonctionnera pas en dehors d'Eclipse ! Là encore, nous y reviendrons plus tard.

Création d'une page web

Vous avez maintenant en mains toutes les informations pour bien débuter. Votre projet dynamique fraîchement créé, vous pouvez maintenant placer votre première page HTML dans son dossier public, c'est-à-dire sous le dossier **WebContent** d'Eclipse (voir le bloc bleu sur notre schéma). Pour cela, tapez une nouvelle fois Ctrl + N au clavier, puis cherchez **HTML File** dans le dossier **Web** de l'arborescence qui apparaît alors. Sélectionnez ensuite le dossier parent, en l'occurrence donc le dossier **WebContent** de votre projet, puis donnez un nom à votre page et validez. Je nomme ici ma page *test.html* :



Création d'une page HTML dans votre projet



Saisie du dossier parent et du nom de la page HTML

Une page HTML est donc apparue dans votre projet, sous le répertoire **WebContent**. Remplacez alors le code automatiquement généré par Eclipse dans votre page par ce code HTML basique :

Code : HTML - test.html

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Test</title>
    </head>
    <body>
        <p>Ceci est une page HTML.</p>
    </body>
</html>
```

Vous pouvez maintenant tenter d'accéder à votre page web fraîchement créée. Pour ce faire, lancez le serveur Tomcat, via le bouton lancement de tomcat si vous avez bien suivi la manière que je vous ai présentée précédemment. Ouvrez ensuite votre navigateur préféré, et entrez l'URL suivante afin d'accéder à votre serveur :

Code : URL

```
http://localhost:8080/test/test.html
```

Votre page s'affiche alors sous vos yeux... déçus !?

C'est quoi toute cette histoire ? Tout un flanc pour afficher 3 mots ?

Patience, patience... Notre serveur étant maintenant fonctionnel, nous voici prêts à entrer dans le vif du sujet.
Nous sommes maintenant prêts pour développer notre première application web. Allons-y !

Partie 2 : Premiers pas

Le travail sérieux commence : au programme, découverte & création de votre première servlet, de votre première page JSP et de votre premier JavaBean, et apprentissage du langage JSP ! Cette partie se terminera enfin par un récapitulatif de ce que nous serons alors capables de faire, et de ce qui nous manquera encore.

La servlet

Nous y voilà enfin ! Nous allons commencer par découvrir ce qu'est une servlet, son rôle au sein de l'application et comment elle doit être mise en place.
J'adopte volontairement pour ce chapitre un rythme assez lent, afin que vous preniez bien conscience des fondements de cette technologie.

Pour ceux qui trouveraient cela barbant, comprenez bien que c'est important de commencer par là et rassurez-vous, nous ne nous soucierons bientôt plus de tous ces détails !

Derrière les rideaux

Retour sur HTTP

Avant d'étudier le code d'une servlet, nous devons nous pencher un instant sur le fonctionnement du protocole HTTP. Pour le moment, nous avons simplement appris que c'était le langage qu'utilisaient le client et le serveur pour s'échanger des informations. Il nous faudrait idéalement un chapitre entier pour l'étudier en détails, mais nous ne sommes pas là pour ça ! Je vais donc tâcher de faire court...

Si nous observions d'un peu plus près ce langage, nous remarquerions alors qu'il ne comprend que quelques mots, appelés **méthodes HTTP**. Ce sont les mots qu'utilise le navigateur pour poser des questions au serveur. Mieux encore, je vous annonce d'emblée que nous ne nous intéresserons qu'à trois de ces mots : **GET**, **POST** et **HEAD**.

GET

C'est la méthode utilisée par le client pour récupérer une ressource web du serveur via une URL. Par exemple, lorsque vous tapez **www.siteduzero.com** dans la barre d'adresse de votre navigateur et que vous validez, votre navigateur envoie une requête GET pour récupérer la page correspondant à cette adresse et le serveur la lui renvoie. La même chose se passe lorsque vous cliquez sur un lien.

Lorsqu'il reçoit une telle demande, le serveur ne fait pas que retourner la ressource demandée, il en profite pour l'accompagner d'informations diverses à son sujet, dans ce qui s'appelle les **en-têtes** ou ***headers*** HTTP : typiquement, on y trouve des informations comme la longueur des données renvoyées ou encore la date d'envoi.

Enfin, sachez qu'il est possible de transmettre des données au serveur lorsque l'on effectue une requête GET, au travers de paramètres directement placés après l'URL (paramètres nommés les *query strings*) ou de cookies placés dans les en-têtes de la requête : nous reviendrons en temps voulu sur ces deux manières de faire. La limite de ce système, c'est que comme la taille d'une URL est limitée, **on ne peut pas utiliser cette méthode pour envoyer des données volumineuses au serveur**, comme par exemple un fichier.

Les gens qui ont écrit la norme décrivant le protocole HTTP ont émis des **recommandations d'usage**, que les développeurs sont libres de suivre ou non. Celles-ci précisent que via cette méthode GET, il est uniquement possible de **récupérer ou lire** des informations, sans que cela ait un quelconque impact sur la ressource demandée : ainsi, une requête GET est censée pouvoir être répétée indéfiniment sans risques pour la ressource concernée.

POST

La taille du corps du message d'une requête POST n'est pas limitée, c'est donc cette méthode qu'il faut utiliser pour soumettre au serveur des données de tailles variables, ou que l'on sait volumineuses. Parfait pour envoyer des fichiers par exemple.

Toujours selon les recommandations d'usage, cette méthode doit être utilisée pour réaliser les opérations qui ont un effet sur la ressource, et qui ne peuvent par conséquent pas être répétées sans l'autorisation explicite de l'utilisateur. Vous avez probablement déjà reçu de votre navigateur un message d'alerte après avoir actualisé une page web, vous prévenant qu'un rafraichissement de la page entrainera un renvoi des informations : eh bien c'est simplement parce que la page que vous souhaitez recharger a été récupérée via la méthode POST, et le navigateur vous demande confirmation avant de renvoyer à

nouveau la requête.

HEAD

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

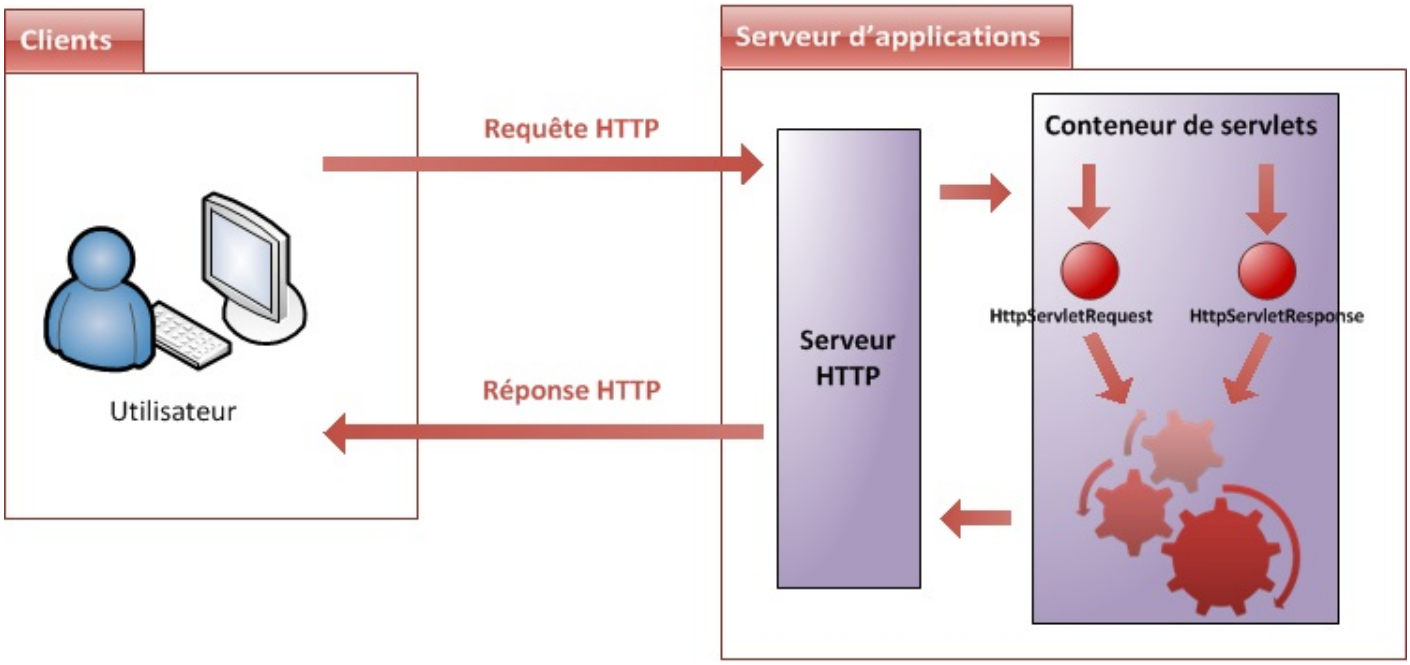
Pendant ce temps là, sur le serveur...

Rappelez-vous notre schéma global : la requête HTTP part du client et arrive sur notre serveur.
L'élément qui entre en jeu est alors le **serveur HTTP** (on parle également de **serveur web**), qui ne fait qu'écouter les requêtes HTTP sur un certain port, en général le port 80.

Que fait-il lorsqu'une requête lui parvient ?

Nous savons déjà qu'il la transmet à un autre élément, que nous avons jusqu'à présent qualifié de conteneur : il s'agit en réalité d'un **conteneur de servlets**, également nommé **conteneur web**. Celui-ci va alors créer deux nouveaux objets :

- `HttpServletRequest` : cet objet contient la requête HTTP, et donne accès à toutes ses informations, telles que les en-têtes (*headers*) et le corps de la requête.
- `HttpServletResponse` : cet objet initialise la réponse HTTP qui sera renvoyée au client, et permet de la personnaliser, en initialisant par exemple les en-têtes et le corps (nous verrons comment par la suite).



Et ensuite ? Que fait-il de ce couple d'objets ?

Eh bien à ce moment précis, c'est votre code qui va entrer en jeu (représenté par la série de rouages sur le schéma). En effet, le conteneur de servlets va les transmettre à votre application, et plus précisément aux servlets et filtres que vous avez éventuellement mis en place. Le cheminement de la requête dans votre code commence à peine, et nous devons déjà nous arrêter : qu'est-ce qu'une **servlet** ?

Création

Une servlet est en réalité une simple classe Java, qui a la particularité de **permettre le traitement de requêtes et la personnalisation de réponses**. Pour faire simple, dans la très grande majorité des cas une servlet n'est rien d'autre qu'une classe

capable de recevoir une requête HTTP envoyée depuis le navigateur de l'utilisateur, et de lui renvoyer une réponse HTTP. C'est tout !

En principe, une servlet dans son sens générique est capable de gérer n'importe quel type de requête, mais dans les faits il s'agit principalement de requêtes HTTP. Ainsi, l'usage veut qu'on ne s'embête pas à préciser "servlet HTTP" lorsque l'on parle de ces dernières, et il est donc extrêmement commun d'entendre parler de servlets alors qu'il s'agit bien en réalité de servlets HTTP. Dans la suite de ce cours, je ferai de même.

Un des avantages de la plate-forme Java EE est sa documentation : très fournie et offrant un bon niveau de détails, la Javadoc permet en un rien de temps de se renseigner sur une classe, une interface ou un package de l'API Java EE. Tout au long de ce cours, je mettrai à votre disposition des liens vers les documentations des objets importants, afin que vous puissiez facilement par vous-mêmes compléter votre apprentissage et vous familiariser avec ce système de documentation.

Regardons donc ce qu'elle contient au chapitre concernant le package servlet : on y trouve une quarantaine de classes et interfaces, parmi lesquelles **l'interface nommée Servlet**. En regardant celle-ci de plus près, on apprend alors qu'elle est **l'interface mère que toute servlet doit obligatoirement implémenter**.

Mieux encore, on apprend en lisant sa description qu'il existe déjà des classes de base qui l'implémentent, et qu'il nous suffit donc d'hériter d'une de ces classes pour créer une servlet :

javax.servlet

Interface Servlet

All Known Subinterfaces:
HttpJspPage, JspPage

All Known Implementing Classes:
FacesServlet, GenericServlet, HttpServlet

```
public interface Servlet
```

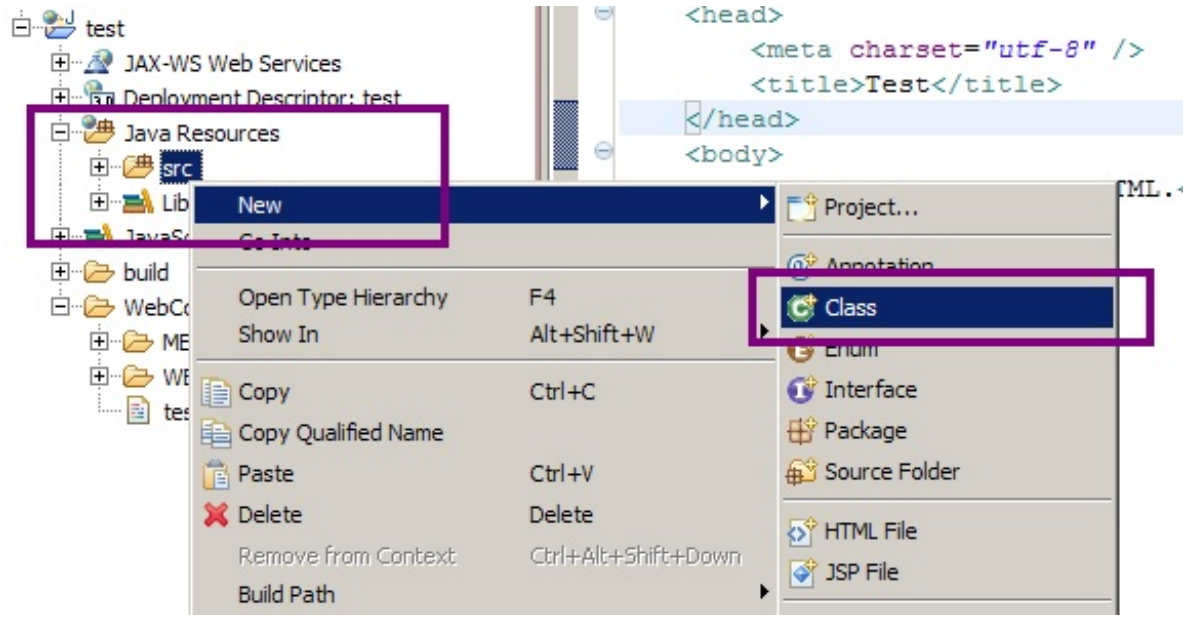
Defines methods that all servlets must implement.

A servlet is a small Java program that runs within a Web server. Servlets receive and respond to requests from Web clients, usually across HTTI Transfer Protocol.

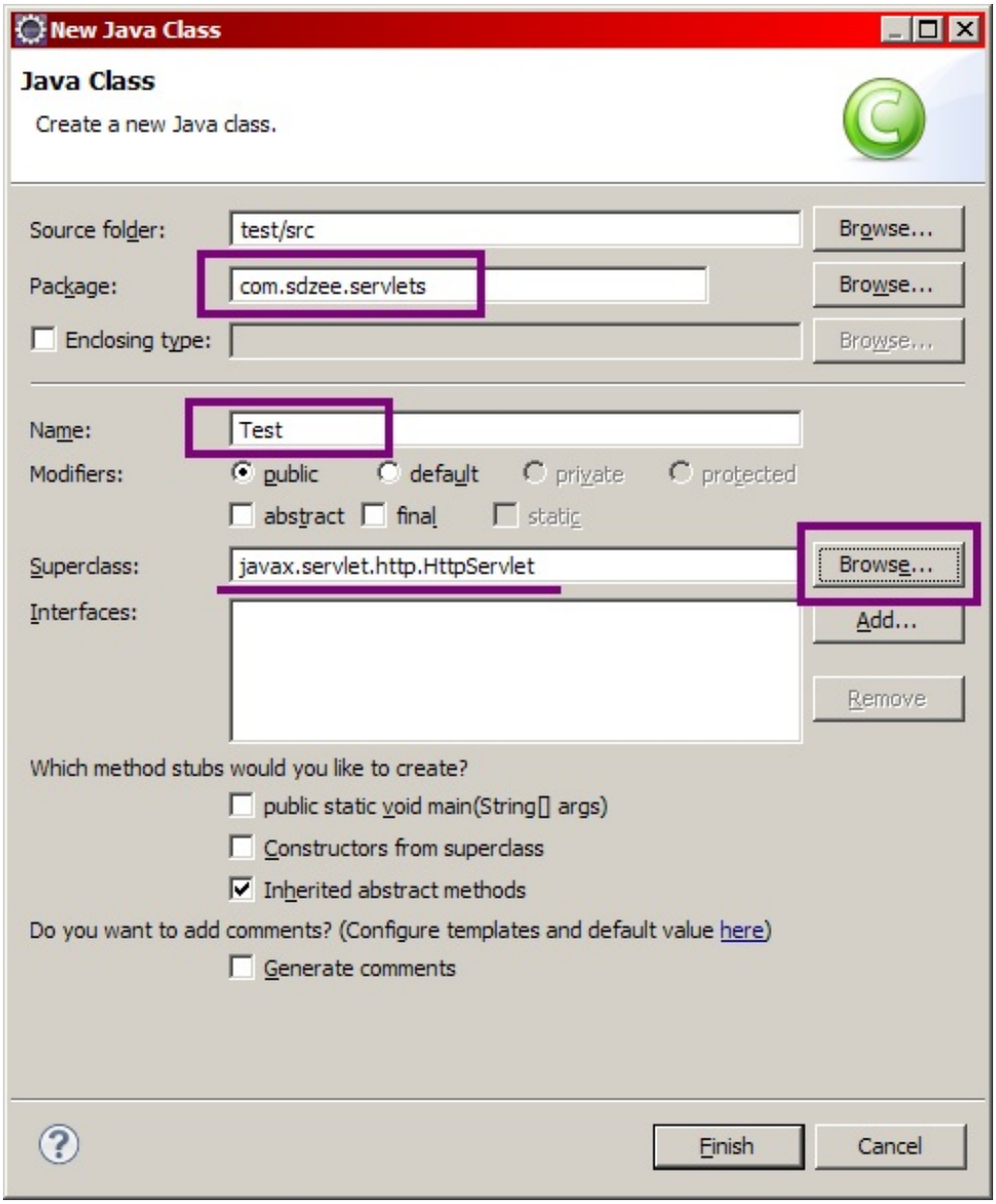
To implement this interface, you can write a generic servlet that extends `javax.servlet.GenericServlet` or an HTTP servlet that extends `javax.servlet.http.HttpServlet`.

Javadoc de l'interface Servlet

Nous souhaitons traiter des requêtes HTTP, nous allons donc faire hériter notre servlet de la classe `HttpServlet` !
De retour sur votre projet Eclipse, faites un clic-droit sur le répertoire **src**, puis choisissez **New > Class**. Renseignez alors la fenêtre qui s'ouvre comme suit :



text



Création d'une servlet

Renseignez le champ **package** par un package de votre choix : pour notre projet, j'ai choisi de le nommer `com.sdzee.servlets` !
Renseignez le nom de la servlet, puis cliquez ensuite sur le bouton Browse... afin de définir de quelle classe doit hériter notre

servlet, puis allez chercher la classe `HttpServlet` et validez. Voici le code que vous obtenez alors automatiquement :

Code : Java - com.sdzee.servlets.Test

```
package com.sdzee.servlets;

import javax.servlet.http.HttpServlet;

public class Test extends HttpServlet {

}
```

Rien d'extraordinaire pour le moment, notre servlet étant absolument vide. D'ailleurs puisqu'elle ne fait encore rien, sautons sur l'occasion pour prendre le temps de regarder ce que contient cette classe `HttpServlet` héritée, afin de voir un peu ce qui se passe derrière. La Javadoc nous donne des informations utiles concernant le fonctionnement de cette classe : pour commencer c'est une classe abstraite, ce qui signifie qu'on ne pourra pas l'utiliser telle quelle et qu'il sera nécessaire de passer par une servlet qui en hérite. On apprend ensuite que la classe propose **les méthodes Java nécessaires au traitement des requêtes et réponses HTTP** ! Ainsi, on y trouve les méthodes :

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

Comment la classe fait-elle pour associer chaque type de requête HTTP à la méthode Java qui lui correspond ?

Vous n'avez pas à vous en soucier, ceci est géré automatiquement par sa méthode `service()` : c'est elle qui se charge de lire l'objet `HttpServletRequest` et de distribuer la requête HTTP à la méthode doXXX() correspondante.

Ce qu'il faut retenir pour le moment :

- une servlet HTTP **doit hériter** de la classe abstraite **HttpServlet** ;
- une servlet **doit implémenter** au moins une des méthodes doXXX(), afin d'être capable de traiter une requête entrante.

Puisque ce sont elles qui prennent en charge les requêtes entrantes, **les servlets vont être les points d'entrée de notre application web, c'est par elles que tout va passer**. Contrairement au Java SE, il n'existe en Java EE pas de point d'entrée unique prédéfini comme pourrait l'être la méthode main()...

Mise en place

Vous le savez, les servlets jouent un rôle très particulier dans une application. Je vous ai parlé d'aiguilleurs en introduction, on peut encore les voir comme des gendarmes : si les requêtes étaient des véhicules, les servlets seraient chargées de faire la circulation sur le gigantesque carrefour qu'est votre application ! Eh bien pour obtenir cette autorité et être reconnues en tant que telles, les servlets nécessitent un traitement de faveur : il va falloir les enregistrer auprès de notre application.

Revenons à notre exemple. Maintenant que nous avons codé notre première servlet, il nous faut donc un moyen de faire comprendre à notre application que notre servlet existe, à la fois pour lui donner l'autorité sur les requêtes et pour la rendre accessible au public ! Lorsque nous avions mis en place une page HTML statique dans le chapitre précédent, le problème ne se posait pas : nous accédions directement à la page en question via une URL directe pointant vers le fichier depuis notre navigateur.

Mais dans le cas d'une servlet - qui rappelons-le est une classe Java - comment faire ?

Concrètement, il va falloir configurer quelque part le fait que notre servlet va être associée à une URL. Ainsi lorsque le client la saisira, la requête HTTP sera automatiquement aiguillée par notre conteneur de servlet vers la bonne servlet, celle qui est en charge de répondre à cette requête. Ce "quelque part" se présente sous la forme d'un simple fichier texte : le fichier **web.xml**.

C'est le cœur de votre application : ici vont se trouver tous les paramètres qui contrôlent son cycle de vie. Nous n'allons pas apprendre d'une traite toutes les options intéressantes, mais y aller par étapes. Commençons donc par apprendre à lier notre servlet à une URL : après tous les efforts que nous avons fournis, c'est le minimum syndical que nous sommes en droit de lui

demander !

Ce fichier de configuration doit impérativement se nommer **web.xml** et se situer juste sous le répertoire **/WEB-INF** de votre application. Si vous avez suivi à la lettre la procédure de création de notre projet web, alors ce fichier est déjà présent : éditez-le, et supprimez le contenu généré par défaut. Si jamais le fichier est absent de votre arborescence, créez simplement un nouveau fichier XML en veillant bien à le placer sous le répertoire **/WEB-INF** et à le nommer **web.xml**. Voici la structure à vide du fichier :

Code : XML - Fichier /WEB-INF/web.xml vide

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

</web-app>
```

L'intégralité de son contenu devra être placé entre les balises **<web-app>** et **</web-app>**.

La mise en place d'une servlet se déroule en deux étapes : nous devons d'abord déclarer la servlet, puis lui faire correspondre une URL.

Définition de la servlet

La première chose à faire est de déclarer notre servlet : en quelque sorte il s'agit de lui donner une carte d'identité, un moyen pour le serveur de la reconnaître. Pour ce faire, il faut ajouter une section au fichier qui se présente ainsi sous sa forme minimale :

Code : XML - Déclaration de notre servlet

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

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

- **<servlet-name>** permet de donner un nom à une servlet. C'est ensuite via ce nom qu'on fera référence à la servlet en question. Ici, j'ai nommé notre servlet **Test**.
- **<servlet-class>** sert à préciser le chemin de la classe de la servlet dans votre application. Ici, notre classe a bien pour nom **Test** et se situe bien dans le package **com.sdzee.servlets**.

Bonne pratique : gardez un nom de classe et un nom de servlet identiques. Bien que ce ne soit en théorie pas nécessaire, cela vous évitera des ennuis ou confusions par la suite.

Il est par ailleurs possible d'insérer au sein de la définition d'une servlet d'autres balises facultatives :

Code : XML - Déclaration de notre servlet avec options

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

 <description>Ma première servlet de test.</description>

 <init-param>
  <param-name>auteur</param-name>
  <param-value>Coyote</param-value>
 </init-param>

 <load-on-startup>1</load-on-startup>
</servlet>
```

On découvre ici trois nouveaux blocs :

- **<description>** permet de décrire plus amplement le rôle de la servlet. Cette description n'a aucune utilité technique et n'est visible que dans ce fichier.
- **<init-param>** permet de préciser des paramètres qui seront accessibles à la servlet lors de son chargement. Nous y reviendrons en détails plus tard dans ce cours.
- **<load-on-startup>** permet de forcer le chargement de la servlet dès le démarrage du serveur. Nous reviendrons sur cet aspect un peu plus loin dans ce chapitre.

Mapping de la servlet

Il faut ensuite faire correspondre notre servlet fraîchement déclarée à une URL, afin qu'elle soit joignable par les clients :

Code : XML - Mapping de notre servlet sur l'URL relative /toto

```
<servlet-mapping>
 <servlet-name>Test</servlet-name>
 <url-pattern>/toto</url-pattern>
</servlet-mapping>
```

La balise responsable de la définition du mapping se nomme logiquement **<servlet-mapping>**, et les deux balises obligatoires de cette section sont là encore très explicites :

- **<servlet-name>** permet de préciser le nom de la servlet à laquelle faire référence. Cette information doit correspondre avec le nom défini dans la précédente déclaration de la servlet.
- **<url-pattern>** permet de préciser la ou les URL relatives au travers desquelles la servlet sera accessible. Ici, ça sera **/toto** !

Pourquoi un "pattern" et pas simplement une URL ?

En effet il s'agit bien d'un pattern, c'est-à-dire un modèle, et pas nécessairement d'une URL fixe. Ainsi, on peut choisir de rendre notre servlet responsable du traitement des requêtes issues d'une seule URL, ou bien d'un groupe d'URL. Vous n'imaginez pour le moment peut-être pas de cas qui impliqueraient qu'une servlet doive traiter les requêtes issues de plusieurs URL, mais rassurez-vous nous ferons la lumière sur ce type d'utilisation dans la partie suivante de ce cours. De même, nous découvrirons qu'il est tout à fait possible de déclarer plusieurs sections **<servlet-mapping>** pour une même section **<servlet>** dans le fichier **web.xml**.

Que signifie "URL relative" ?

Cela veut dire que l'URL ou le pattern que vous renseignez dans le champ **<url-pattern>** est basé sur **le contexte de votre application**. Dans notre cas, souvenez-vous du contexte de déploiement que nous avons précisé lorsque nous avons créé notre projet web : nous l'avions appelé **test**. Nous en déduisons donc que notre **<url-pattern>**/toto**</url-pattern>** fait référence à l'URL absolue /test/toto.

Nous y voilà, notre servlet est maintenant joignable par le client via l'URL suivante :

Code : URL

```
http://localhost:8080/test/toto
```

Pour information, le code final de notre fichier **web.xml** est donc :

Code : XML - /WEB-INF/web.xml

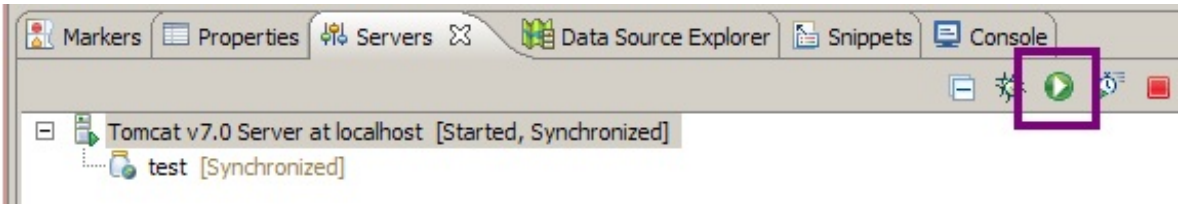
```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
 <servlet>
  <servlet-name>Test</servlet-name>
  <servlet-class>com.sdzee.servlets.Test</servlet-class>
 </servlet>

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

L'ordre des sections de déclaration au sein du fichier est important : il est impératif de définir une servlet avant de spécifier son mapping.

Mise en service

Do you « GET » it?

Nous venons de créer un fichier de configuration pour notre application, nous devons donc redémarrer notre serveur pour que ces modifications soient prises en compte. Il suffit pour cela de cliquer sur le bouton "start" de l'onglet **Servers** :



Bouton de redémarrage du serveur Tomcat dans Eclipse

Faisons le test, et observons ce que nous affiche notre navigateur lorsque nous tentons d'accéder à l'URL que nous venons de mapper sur notre servlet :

Méthode HTTP non supportée

Nous voici devant notre premier code de statut HTTP. En l'occurrence, c'est à la fois une bonne et une mauvaise nouvelle :

- une bonne nouvelle car cela signifie que notre mapping a fonctionné et que notre serveur a bien contacté notre servlet !
- une mauvaise nouvelle car notre serveur nous retourne le code d'erreur 405 et nous précise que la méthode GET n'est pas supportée par la servlet que nous avons associée à l'URL...

Par qui a été générée cette page d'erreur ?

Tout est parti du conteneur de servlets. D'ailleurs, ce dernier effectue pas mal de choses dans l'ombre, sans vous le dire ! Dans ce cas précis, il a :

1. reçu la requête HTTP depuis le serveur web
2. généré un couple d'objets requête/réponse
3. parcouru le fichier web.xml de votre application à la recherche d'une entrée correspondant à l'URL contenue dans l'objet

requête
4. trouvé et identifié la servlet que vous y avez déclarée
5. contacté votre servlet et transmis la paire d'objets requête/réponse

Dans ce cas, pourquoi cette page d'erreur a-t-elle été générée ?

Nous avons pourtant bien fait hériter notre servlet de la classe `HttpServlet`, notre servlet doit pouvoir interagir avec HTTP ! Qu'est-ce qui cloche ? Eh bien nous avons oublié une chose importante : afin que notre servlet soit capable de traiter une requête HTTP de type GET, il faut y implémenter une méthode... `doGet()` ! Souvenez-vous, je vous ai déjà expliqué que la méthode `service()` de la classe `HttpServlet` s'occupera alors elle-même de transmettre la requête GET entrante vers la méthode `doGet()` de notre servlet.... Ça vous revient ?

Maintenant, comment cette page d'erreur a-t-elle été générée ?

C'est la méthode `doGet()` de la classe mère `HttpServlet` qui est en cause. Ou plutôt, disons que c'est grâce à elle ! En effet, le comportement par défaut des méthodes doXXX() de la classe `HttpServlet` est de renvoyer un code d'erreur HTTP 405 ! Donc si le développeur a bien fait son travail, pas de problème : c'est bien la méthode doXXX() de la servlet qui sera appelée. Par contre, s'il a mal fait son travail et a oublié de surcharger la méthode doXXX() voulue, alors c'est la méthode de la classe mère `HttpServlet` qui sera appelée, et un code d'erreur sera gentiment et automatiquement renvoyé au client. Ainsi, la classe mère s'assure toujours que sa classe fille - votre servlet ! - surcharge bien la méthode doXXX() correspondant à la méthode HTTP traitée !

Par ailleurs, votre conteneur de servlets est également capable de générer lui-même des codes d'erreur HTTP. Par exemple, lorsqu'il parcourt le fichier web.xml de votre application à la recherche d'une entrée correspondant à l'URL envoyée par le client, et qu'il ne trouve rien, c'est lui qui va se charger de générer le fameux code d'erreur 404 !

Nous voilà maintenant au courant de ce qu'il nous reste à faire : il nous suffit de surcharger la méthode `doGet()` de la classe `HttpServlet` dans notre servlet Test. Voici donc le code de notre servlet :

Code : Java - Surcharge de la méthode doGet() dans notre servlet Test

```
package com.sdzee.servlets;

import java.io.IOException;

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

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

 }
}
```

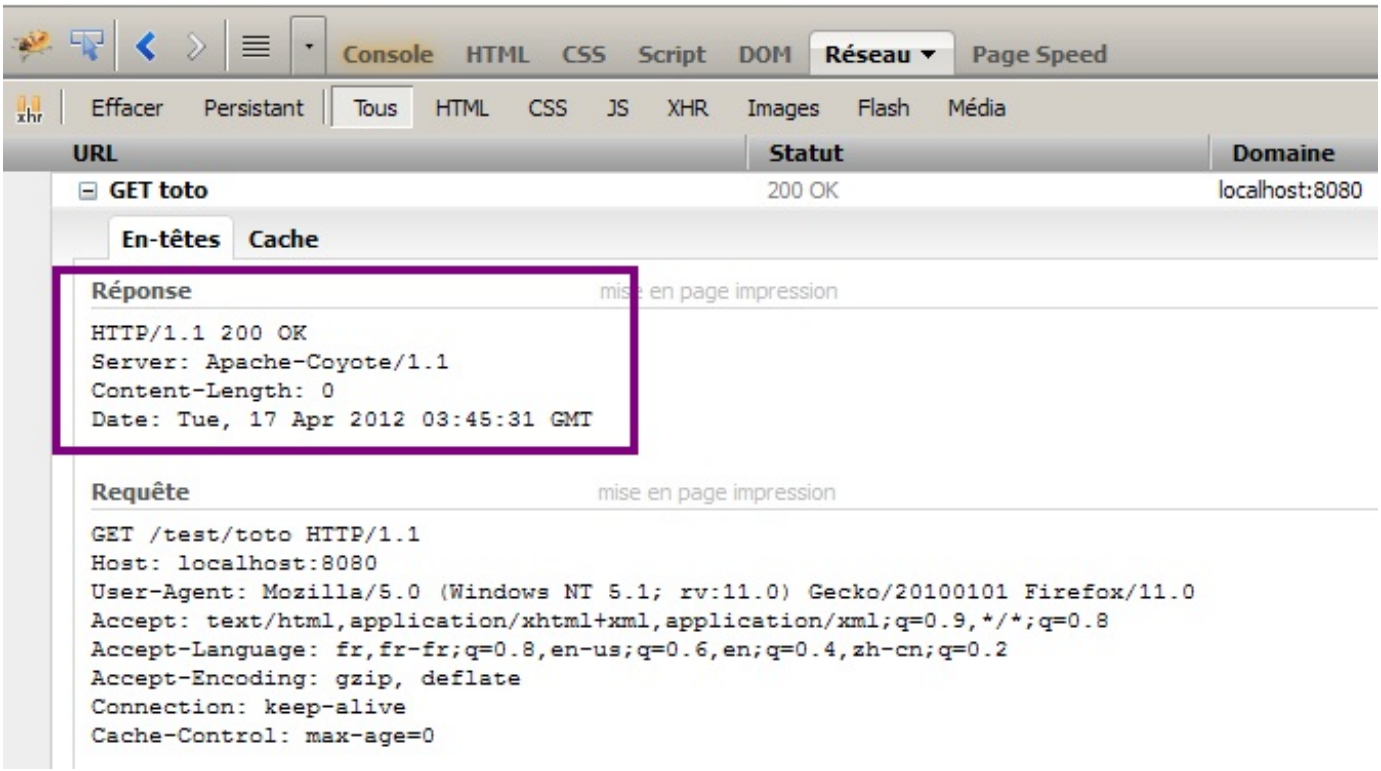
Comme vous pouvez le constater, l'ajout de cette seule méthode vide fait intervenir plusieurs imports qui définissent les objets et exceptions présents dans la signature de la méthode : `HttpServletRequest`, `HttpServletResponse`, `ServletException` et `IOException`.

Réessayons alors de contacter notre servlet via notre URL : tout se passe comme prévu, le message d'erreur HTTP disparaît. Cela dit, notre servlet ne fait strictement rien de la requête HTTP reçue : le navigateur nous affiche alors une page... blanche !

Comment le client sait-il que la requête est arrivée à bon port ?

C'est une très bonne remarque. En effet, si votre navigateur vous affiche une simple page blanche, c'est parce qu'il considère la

requête comme terminée avec succès : si ce n'était pas le cas, il vous afficherait un des codes et messages d'erreur HTTP... Si vous utilisez le navigateur Firefox, vous pouvez utiliser l'onglet Réseau de l'outil Firebug pour visualiser qu'effectivement, une réponse HTTP est bien reçue par votre navigateur (si vous utilisez le navigateur Chrome, vous pouvez accéder à un outil similaire en appuyant sur F12) :



En-têtes de la réponse HTTP avec Firebug

On y observe :

- un code HTTP 200 OK, qui signifie que la requête s'est effectuée avec succès ;
- la longueur des données contenues dans la réponse (*Content-Length*) : 0...

Eh bien encore une fois, c'est le conteneur de servlets qui a fait le boulot sans vous prévenir ! Quand il a généré la paire d'objets requête/réponse, il a initialisé le statut de la réponse avec une valeur par défaut : 200. C'est-à-dire que par défaut, le conteneur de servlets crée un objet réponse qui stipule que tout s'est bien passé. Ensuite, il transmet cet objet à votre servlet, qui est alors libre de le modifier à sa guise. Lorsqu'il reçoit à nouveau l'objet en retour, si le code de statut n'a pas été modifié par la servlet, c'est que tout s'est bien passé. En d'autres termes, le conteneur de servlets adopte une certaine philosophie : pas de nouvelles, bonne nouvelle !

Le serveur retourne donc toujours une réponse au client, peu importe ce que fait notre servlet avec la requête ! Dans notre cas, la servlet n'effectue aucune modification sur l'objet HttpServletResponse, et par conséquent n'y insère aucune donnée et n'y modifie aucune en-tête. D'où la longueur initialisée à zéro dans l'en-tête de la réponse, le code de statut initialisé à 200... et la page blanche en guise de résultat final !

Cycle de vie d'une servlet

Dans certains cas, il peut s'avérer utile de connaître les rouages qui se cachent derrière une servlet. Toutefois, je ne souhaite pas vous embrouiller dès maintenant : vous n'en êtes qu'aux balbutiements de votre apprentissage et n'avez pas assez d'expérience pour intervenir proprement sur l'initialisation d'une servlet. Je ne vais par conséquent qu'aborder rapidement son cycle de vie au sein du conteneur, à travers ce court aparté. Nous lèverons le voile sur toute cette histoire dans un chapitre en annexe de ce cours, et en profiterons pour utiliser le puissant outil de debug d'Eclipse !

Quand une servlet est demandée pour la première fois ou quand l'application web démarre, le conteneur de servlets va créer une instance de celle-ci et la garder en mémoire pendant toute l'existence de l'application. **La même instance sera réutilisée pour**

chaque requête entrante dont les URL correspondent au pattern d'URL défini pour la servlet. Dans notre exemple, aussi longtemps que notre serveur restera en ligne, tous nos appels vers l'URL **/test/toto** seront dirigés vers la même et unique instance de notre servlet, générée par Tomcat lors du tout premier appel.

En fin de compte, l'instance d'une servlet est créée lors du premier appel à cette servlet, ou bien dès le démarrage du serveur ?

Ceci dépend en grande partie du serveur d'applications utilisé. Dans notre cas, avec Tomcat c'est par défaut au premier appel d'une servlet que son unique instance est créée.

Toutefois, ce mode de fonctionnement est configurable. Plus tôt dans ce chapitre, je vous expliquais comment déclarer une servlet dans le fichier web.xml, et j'en ai profité pour vous présenter une balise facultative : **<load-on-startup>N</load-on-startup>**, où N doit être un entier positif. Si dans la déclaration d'une servlet vous ajoutez une telle ligne, alors vous ordonnez au serveur de charger l'instance de la servlet en question directement pendant le chargement de l'application.

Le chiffre N correspond à la priorité que vous souhaitez donner au chargement de votre servlet. Dans notre projet nous n'utilisons pour le moment qu'une seule servlet, donc nous pouvons bien marquer n'importe quel chiffre supérieur ou égal à zéro, ça ne changera rien. Mais dans le cas d'une application contenant beaucoup de servlets, cela permet de définir quelle servlet doit être chargée en premier. L'ordre est établi du plus petit au plus grand : la ou les servlets ayant un load-on-startup initialisé à zéro sont les premières à être chargées, puis 1, 2, 3, etc.

Voilà tout pour cet aparté. En ce qui nous concerne, nous n'utiliserons pas cette option de chargement dans nos projets, le chargement des servlets lors de leur première sollicitation nous ira très bien !

Envoyer des données au client

Avec tout cela, nous n'avons encore rien envoyé à notre client, alors qu'en mettant en place une simple page HTML nous avions affiché du texte dans le navigateur du client en un rien de temps. Patience, les réponses vont venir... Utilisons notre servlet pour reproduire la page HTML statique que nous avions créée lors de la mise en place de Tomcat. Comme je vous l'ai expliqué dans le paragraphe précédent, pour envoyer des données au client il va falloir manipuler l'objet `HttpServletResponse`. Regardons d'abord ce qu'il est nécessaire d'inclure à notre méthode `doGet()`, et analysons tout cela ensuite :

Code : Java

```
public void doGet( HttpServletRequest request, HttpServletResponse
response ) throws ServletException, IOException{
 response.setContentType("text/html");
 PrintWriter out = response.getWriter();
 out.println("<!DOCTYPE html>");
 out.println("<html>");
 out.println("<head>");
 out.println("<meta charset=\"utf-8\" />");
 out.println("<title>Test</title>");
 out.println("</head>");
 out.println("<body>");
 out.println("<p>Ceci est une page générée depuis une
servlet.</p>");
 out.println("</body>");
 out.println("</html>");
}
```

Comment procédons-nous :

1. Nous commençons par modifier l'en-tête **Content-Type** de la réponse HTTP, pour préciser au client que nous allons lui envoyer du texte au sein d'une page HTML, en faisant appel à la méthode `setContentType()` de l'objet `HttpServletResponse`.
2. Nous récupérons ensuite un objet `PrintWriter` qui va nous permettre d'envoyer du texte au client, via la méthode `getWriter()` de l'objet `HttpServletResponse`. Vous devrez donc importer `java.io.PrintWriter` dans votre servlet.
3. Nous écrivons alors du texte dans la réponse via la méthode `println()` de l'objet `PrintWriter`.

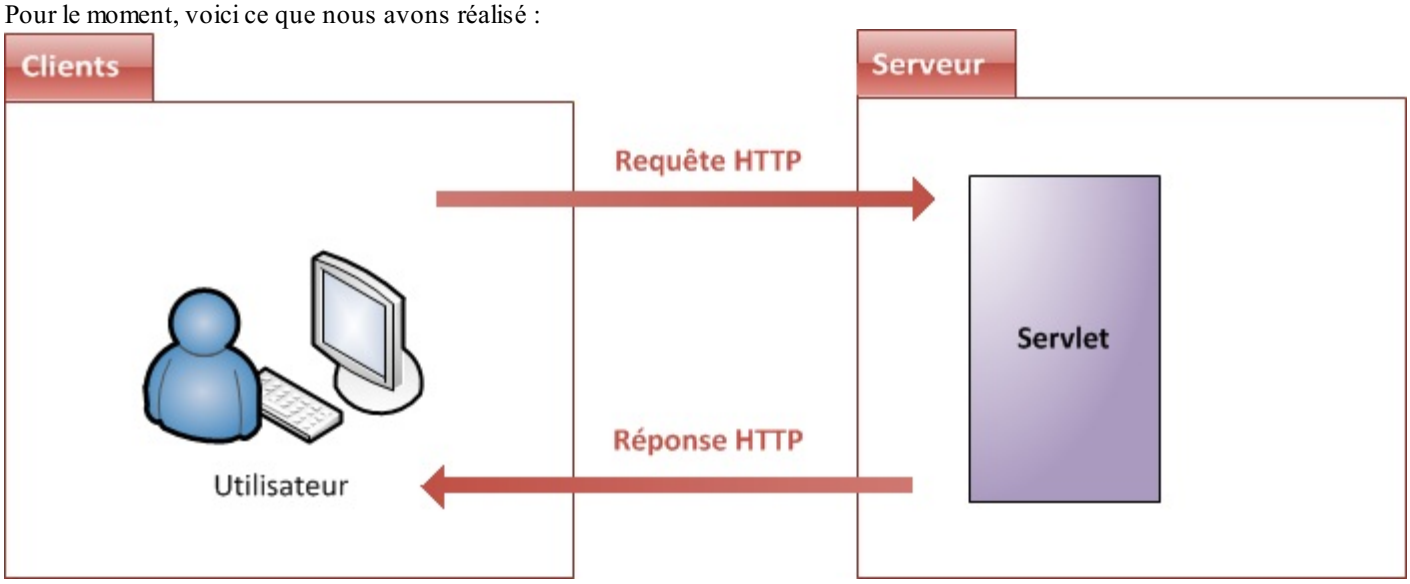
Enregistrez, testez et vous verrez enfin la page s'afficher dans votre navigateur : ça y est, vous savez maintenant utiliser une

servlet et transmettre des données au client.

Rien que pour reproduire ce court et pauvre exemple, il nous a fallu 10 appels à `out.println()` ! Lorsque nous nous attaquerons à des pages web un peu plus complexes que ce simple exemple, allons-nous devoir écrire tout notre code HTML à l'intérieur de ces méthodes `println()` ?

Non, bien sûr que non ! Vous vous imaginez un peu l'horreur si c'était le cas ?! Si vous avez suivi le topo sur MVC, vous vous souvenez d'ailleurs que **la servlet n'est pas censée s'occuper de l'affichage, c'est la vue qui doit s'en charger** ! Et c'est bien pour ça que je ne vous ai rien fait envoyer d'autre que cette simple page d'exemple HTML... Toutefois, même si nous ne procèderons plus jamais ainsi pour la création de nos futures pages web, il était très important que nous découvrions comment cela se passe.

Pour le moment, voici ce que nous avons réalisé :



Servlet seule

Note : dorénavant et afin d'alléger nos schémas, je ne représenterai plus le serveur HTTP en amont du conteneur. Ici, le bloc intitulé "Serveur" correspond en réalité au conteneur de servlets.

Pour information, nous nous resservirons plus tard de cette technique d'envoi direct de données depuis une servlet, lorsque nous manipulerons des fichiers.

La leçon à retenir en cette fin de chapitre est claire : le langage Java n'est pas du tout adapté à la rédaction de pages web ! Notre dernier exemple en est une excellente preuve, et il nous faut nous orienter vers quelque chose de plus efficace.

Il est maintenant grand temps de revenir au modèle MVC : l'affichage de contenu HTML n'ayant rien à faire dans le contrôleur (notre servlet), nous allons créer une vue et la mettre en relation avec notre servlet.

Servlet avec vue...

Le modèle MVC nous conseille de placer tout ce qui touche à l'affichage final (texte, mise en forme, etc.) dans une couche à part : la vue. Nous avons dans la première partie de ce cours rapidement survolé comment ceci se concrétisait en Java EE : la technologie utilisée pour réaliser une vue est **la page JSP**. Nous allons dans ce chapitre découvrir comment fonctionne une telle page, et apprendre à en mettre une en place au sein de notre embryon d'application.

Introduction aux JSP

À quoi ressemble une page JSP ?

C'est un document qui à première vue ressemble beaucoup à une page HTML, mais qui en réalité en diffère par plusieurs aspects :

- l'extension d'une telle page devient **.jsp** et non plus .html ;
- une telle page peut contenir des balises HTML, mais également des **balises JSP** qui appellent de manière transparente du code Java ;
- contrairement à une page HTML statique directement renvoyée au client, **une page JSP est exécutée côté serveur**, et génère alors une page renvoyée au client.

L'intérêt est de rendre possible la création de pages **dynamiques** : puisqu'il y a une étape de génération sur le serveur, il devient possible de faire varier l'affichage et d'interagir avec l'utilisateur, en fonction notamment de la requête et des données reçues !

Bien que la syntaxe d'une page JSP soit très proche de celle d'une page HTML, il est théoriquement possible de générer n'importe quel type de format avec une page JSP : du HTML donc, mais tout aussi bien du CSS, du XML, du texte brut, etc. Dans notre cas, dans la très grande majorité des cas d'utilisation il s'agira de pages HTML destinées à l'affichage des données de l'application sur le navigateur du client.

Ne vous fiez pas au titre de ce sous-chapitre, nous n'allons pour le moment pas nous intéresser à la technologie JSP en elle-même, ceci faisant l'objet des chapitres suivants. Nous allons nous limiter à l'étude de ce qu'est une JSP, de la manière dont elle est interprétée par notre serveur et comment elle s'insère dans notre application.

Nature d'une JSP

Quoi ?

Les pages JSP sont une des technologies de la plate-forme Java EE les plus puissantes, simples à utiliser et à mettre en place. Elles se présentent sous la forme d'un simple fichier au format texte, contenant des balises respectant une syntaxe à part entière. Le langage JSP combine à la fois les technologies HTML, XML, servlet et JavaBeans (nous reviendrons sur ce terme plus tard, pour le moment retenez simplement que c'est un objet Java) en une seule solution permettant aux développeurs de créer des vues dynamiques.

Pourquoi ?

Pour commencer, mettons noir sur blanc les raisons de l'existence de cette technologie :

- La technologie servlet est trop difficile d'accès et ne convient pas à la génération du code de présentation : nous l'avons souligné en fin de chapitre précédent, écrire une page web en langage Java est horriblement pénible. Il est nécessaire de disposer d'une technologie qui joue le rôle de **simplification de l'API servlet** : les pages JSP sont en quelque sorte une abstraction "haut-niveau" de la technologie servlet.
- **Le modèle MVC recommande une séparation nette entre le code de contrôle et la présentation.** Il est théoriquement envisageable d'utiliser certaines servlets pour effectuer le contrôle, et d'autres pour effectuer l'affichage, mais nous rejoignons alors le point précédent : la servlet n'est pas adaptée à la prise en charge de l'affichage...
- **Le modèle MVC recommande une séparation nette entre le code métier et la présentation** : dans le modèle on doit trouver le code Java responsable de la génération des éléments dynamiques, et dans la vue on doit simplement trouver l'interface utilisateur ! Ceci afin notamment de permettre aux développeurs et designers de travailler facilement sur la vue, sans avoir à y faire intervenir directement du code Java.

Comment ?

On peut résumer la technologie JSP en **une technologie offrant les capacités dynamiques des servlets tout en permettant une approche naturelle pour la création de contenu statiques.** Ceci est rendu possible par :

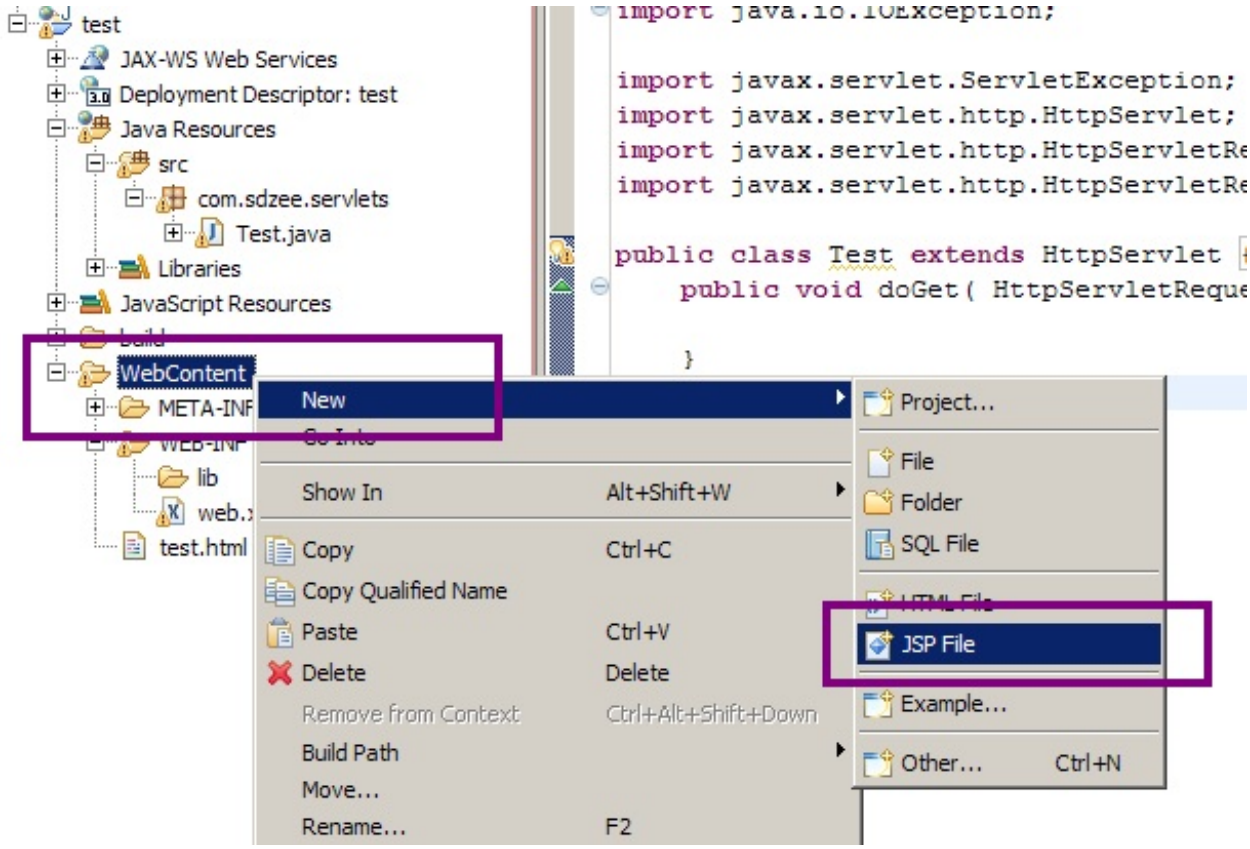
- **un langage dédié** : les pages JSP sont des documents au format texte, à l'opposé des classes Java que sont les servlets, qui décrivent indirectement comment traiter une requête et construire une réponse. Elles contiennent des balises qui combinent à la fois simplicité et puissance, via une syntaxe simple, semblable au HTML et donc aisément compréhensibles par un humain ;
- **la simplicité d'accès aux objets Java** : des balises du langage rendent l'utilisation directe d'objets au sein d'une page très aisée ;
- **des mécanismes permettant l'extension du langage** utilisé au sein des pages JSP : il est possible de mettre en place des balises qui n'existent pas dans le langage JSP, afin d'augmenter les fonctionnalités accessibles. Pas de panique, ça paraît complexe a priori mais nous y reviendrons calmement dans la partie concernant la JSTL, et tout cela n'aura bientôt plus aucun secret pour vous !

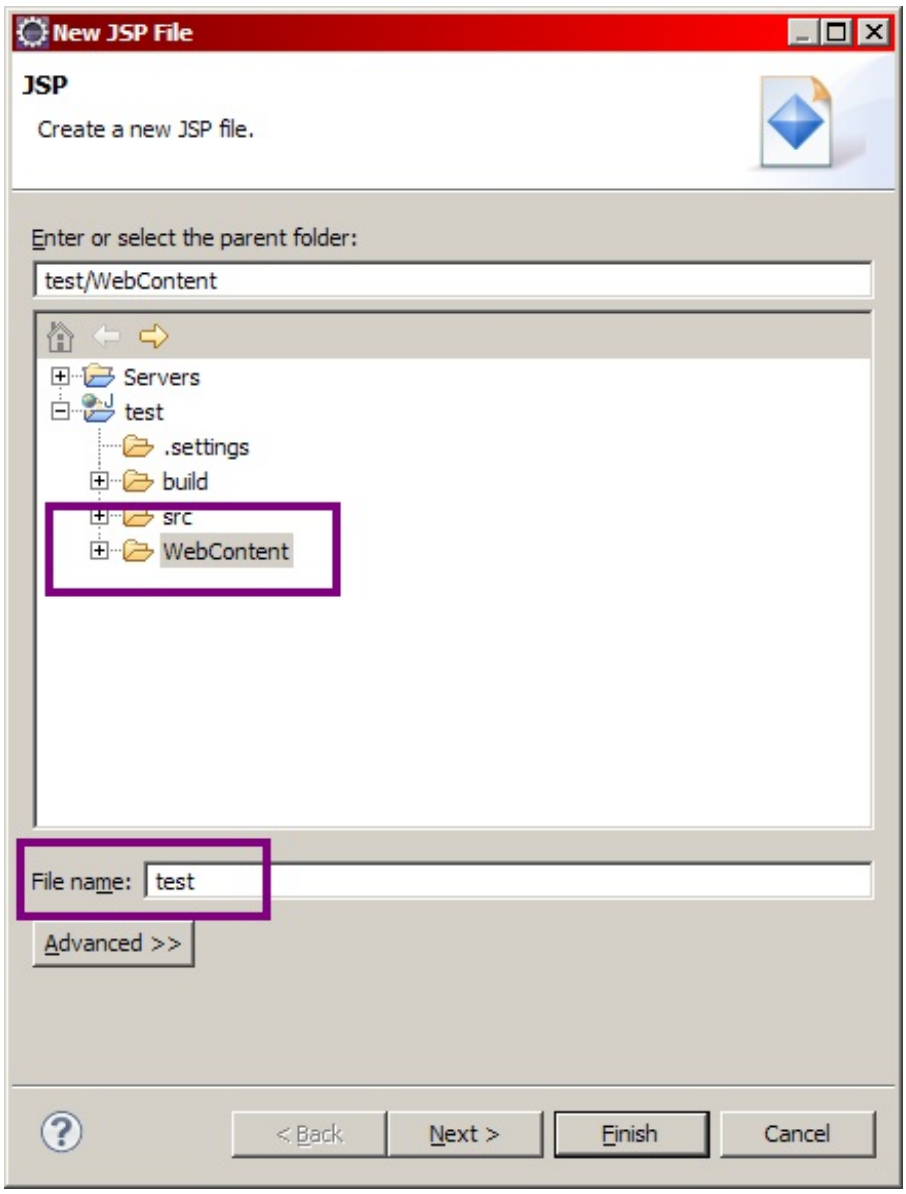
Bon, assez gambergé ! Maintenant que nous avons une bonne idée de ce que sont les pages JSP, rentrons dans le concret en étudiant leur vie au sein d'une application !

Mise en place d'une JSP

Création de la vue

Le contexte étant posé, nous pouvons maintenant créer notre première page JSP. Pour ce faire, depuis votre projet Eclipse faites un clic-droit sur le dossier **WebContent** de votre projet, puis choisissez **New > JSP File**, et dans la fenêtre qui apparaît renseignez le nom de votre page JSP :





Une page JSP par défaut est alors générée par Eclipse : supprimez tout son contenu, et remplacez-le par notre modèle d'exemple :

Code : HTML - test.jsp

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Test</title>
    </head>

    <body>
        <p>Ceci est une page générée depuis une JSP.</p>
    </body>
</html>
```

Rendez vous ensuite dans la barre d'adresse de votre navigateur, et entrez l'URL correspondant à la page que vous venez de créer :

Code : URL

```
http://localhost:8080/test/test.jsp
```

Nous obtenons alors le même résultat qu'avec notre servlet auparavant : le navigateur nous affiche bien notre page.
Tout va bien donc, notre JSP est bien fonctionnelle !

Cycle de vie d'une JSP

En théorie

Tout tient en une seule phrase : quand une JSP est demandée pour la première fois, ou quand l'application web démarre, le conteneur de servlets va **vérifier, traduire puis compiler la page JSP en une classe héritant de HttpServlet**, et l'utiliser durant l'existence de l'application.

Cela signifie-t-il qu'une JSP est littéralement transformée en servlet par le serveur ?

C'est exactement ce qui se passe. Lors de la demande d'une JSP, le moteur de servlets va exécuter la classe JSP auparavant traduite et compilée et envoyer la sortie générée (typiquement, une page HTML/CSS/JS) depuis le serveur vers le client à travers le réseau, sortie qui sera alors affichée dans son navigateur !

Pourquoi ?

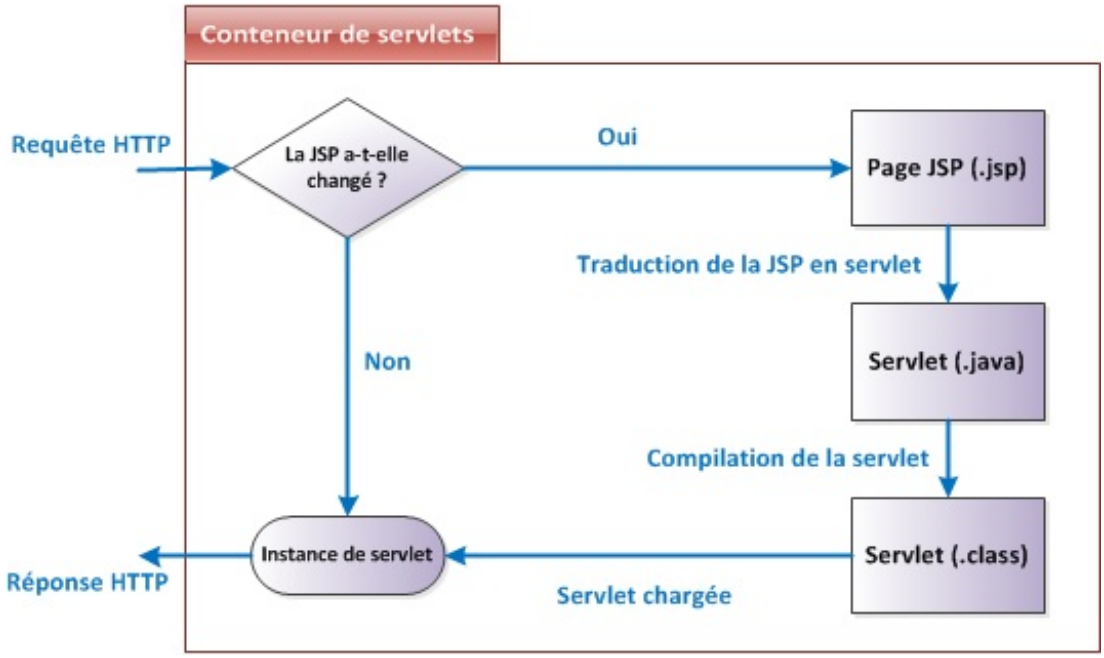
Je vous l'ai déjà dit, la technologie JSP consiste en une véritable abstraction de la technologie servlet : cela signifie concrètement que **les JSP permettent au développeur de faire du Java sans avoir à écrire de code Java** ! Bien que cela paraisse magique, rassurez-vous il n'y a pas de miracles : vous pouvez voir le code JSP écrit par le développeur comme une succession de raccourcis en tous genres, qui dans les coulisses appellent en réalité des portions de code Java toutes prêtes !

Que se passe-t-il si le contenu d'une page JSP est modifié ? Que devient la servlet auto-générée correspondante ?

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

- le conteneur vérifie si la JSP a déjà été traduite et compilée en une servlet :
  - si non, il vérifie la syntaxe de la page, la traduit en une servlet (du code Java) et la compile en une classe exécutable prête à l'emploi.
  - si oui, il vérifie que l'âge de la JSP et de la servlet sont identiques :
    - si non, cela signifie que la JSP est plus récente que la servlet et donc qu'il y a eu modification, le conteneur effectue alors à nouveau les tâches de vérification, traduction et compilation.
- il charge ensuite la classe générée, en crée une instance et l'exécute pour traiter la requête.

J'ai représenté cette suite de décisions dans le schéma suivant, afin de vous faciliter la compréhension du cycle :



De tout cela, il faut retenir que le processus initial de vérification/traduction/compilation n'est pas effectué à chaque appel ! La servlet générée et compilée étant **sauvegardée**, les appels suivants à la JSP sont beaucoup plus rapides : le conteneur se contente d'exécuter directement l'instance de la servlet stockée en mémoire.

En pratique

Avant de passer à la suite, revenons sur cette histoire de traduction en servlet. Je vous ai dit que le conteneur de servlets, en l'occurrence ici Tomcat, générait une servlet à partir de votre JSP. Eh bien sachez que vous pouvez trouver le code source ainsi généré dans **le répertoire de travail du serveur** : sous Tomcat, il s'agit du répertoire **/work**.

Qu'en est-il de notre première JSP ? Existe-t-il une servlet auto-générée depuis nos quelques lignes de texte ?

La réponse est oui, bien entendu ! Cette servlet est bien présente dans le répertoire de travail de Tomcat, seulement comme nous gérons notre serveur directement depuis Eclipse, par défaut ce dernier va en quelque sorte prendre la main sur Tomcat, et mettre tous les fichiers dans un répertoire à lui ! Le fourbe... Bref, voilà où se trouve ma servlet pour cet exemple :

Code : URI

```
C:\eclipse\workspace\.metadata\.plugins\org.eclipse.wst.server.core\tmp0\work\Cat
C:\eclipse\workspace\.metadata\.plugins\org.eclipse.wst.server.core\tmp0\work\Cat
```

Elle doit se situer sensiblement au même endroit chez vous, à un ou deux noms de dossier près selon la configuration que vous avez mise en place et bien entendu selon le système d'exploitation que vous utilisez. Vous remarquerez que Tomcat suffixe les servlets qu'il auto-génère à partir de pages JSP avec "_jsp".

Je vous conseille alors d'éditer le fichier .java et de consulter les sources générées, c'est un exercice très formateur pour vous que de tenter de comprendre ce qui y est fait : n'oubliez pas, la Javadoc est votre amie pour comprendre les méthodes qui vous sont encore inconnues. Ne prenez surtout pas peur devant ce qui s'apparente à un joyeux bordel, et passez faire un tour sur le forum Java si vous avez des questions précises sur ce qui s'y trouve !

Sans surprise, au milieu du code généré par Tomcat nous retrouvons bien des instructions très semblables à celles que nous avions dû écrire dans notre servlet dans le chapitre précédent, et qui correspondent cette fois à ce que nous avons écrit dans notre JSP :

Code : Java - Extrait de la servlet test_jsp.java auto-générée par Tomcat depuis notre page test.jsp

```
out.write("<!DOCTYPE html>\r\n");
out.write("<html>\r\n");
out.write(" <head>\r\n");
out.write(" <meta charset=\"utf-8\" />\r\n");
out.write(" <title>Test</title>\r\n");
out.write(" </head>\r\n");
out.write(" <body>\r\n");
out.write(" <p>Ceci est une page générée depuis une JSP.</p>\r\n");
out.write(" </body>\r\n");
out.write("</html>");
```

Retenez bien que c'est cette classe Java qui est compilée et exécutée lorsque votre JSP est appelée. Ce court aparté se termine ici, dorénavant nous ne nous préoccuperons plus de ce qui se trame dans les coulisses de notre serveur : nous aurons bien assez de travail avec nos JSP... et le reste !

Mise en relation avec notre servlet

Garder le contrôle

Dans l'exemple que nous venons de réaliser, nous nous sommes contentés d'afficher du texte statique, et avons visualisé le résultat en appelant directement la page JSP depuis son URL. C'était pratique pour le coup, mais dans une application Java EE il ne faut jamais procéder ainsi ! Pourquoi ? La réponse tient en trois lettres : MVC. Ce modèle de conception nous recommande en effet la mise en place d'un contrôleur, et nous allons donc tâcher de **toujours associer une servlet à une vue**.

Mais je viens de vous montrer qu'une JSP était de toute façon traduite en servlet... Quel est l'intérêt de mettre en place une autre servlet ?

Une JSP est en effet automatiquement traduite en servlet, mais attention à ne pas confondre : les contrôleurs du MVC sont bien représentés en Java EE par des servlets, mais cela ne signifie pas pour autant que toute servlet joue le rôle d'un contrôleur dans une application Java EE. En l'occurrence, les servlets résultant de la traduction des JSP dans une application n'ont pour rôle que de permettre la manipulation des requêtes et réponses HTTP. En aucun cas elles n'interviennent dans la couche de contrôle, elles agissent de manière transparente et font bien partie de la vue : ce sont simplement des traductions en un langage que comprend le serveur (le Java !) des vues présentes dans votre application (de simples fichiers textes contenant de la syntaxe JSP).

Dorénavant, nous allons donc systématiquement créer une servlet lorsque nous créerons une page JSP. Ça peut vous sembler pénible au début, mais c'est une bonne pratique à prendre dès maintenant : vous gardez ainsi le contrôle, en vous assurant qu'une vue ne sera jamais appelée par le client sans être passée à travers une servlet. **Souvenez-vous : la servlet est le point d'entrée de votre application !**

Nous allons même pousser le vice, et déplacer notre page JSP sous le répertoire **/WEB-INF**. Si vous vous souvenez de ce que je vous ai dit dans le chapitre sur la configuration de Tomcat, vous savez que ce dossier a une particularité qui nous intéresse : il cache automatiquement les ressources qu'il contient. En d'autres termes, une page présente sous ce répertoire n'est plus accessible directement par une URL côté client ! Il devient alors **nécessaire** de passer par une servlet côté serveur pour donner l'accès à cette page... Plus d'oubli possible !

Faites le test, essayez depuis une URL de joindre votre page JSP après l'avoir déplacée sous **/WEB-INF** : vous n'y arriverez pas !

Nous devons donc associer notre servlet à notre vue. Cette opération est réalisée depuis la servlet, ce qui est logique puisque c'est elle qui décide d'appeler la vue.
Reprenons notre servlet, vidons la méthode `doGet()` du contenu que nous avons depuis fait migrer dans la JSP, et regardons le code à mettre en place pour effectuer l'association :

Code : Java - com.sdzee.servlets.Test

```
...

public void doGet( HttpServletRequest request, HttpServletResponse
response ) throws ServletException, IOException {
 this.getServletContext().getRequestDispatcher( "/WEB-INF/test.jsp"
```

```
        ).forward( request, response );
        }
```

Analysons ce qui se passe :

- depuis notre instance de servlet (**this**), nous appelons la méthode `getServletContext()`. Celle-ci nous retourne alors un objet `ServletContext`, qui fait référence au contexte commun à toute l'application : celui-ci contient un ensemble de méthodes qui permettent à une servlet de communiquer avec le conteneur de servlet.
- celle qui nous intéresse ici est la méthode permettant de manipuler une ressource : `getRequestDispatcher()`, que nous appliquons à notre page JSP. Elle retourne un objet `RequestDispatcher`, qui agit ici comme une enveloppe autour de notre page JSP. Vous pouvez considérer cet objet comme la pierre angulaire de votre servlet : c'est grâce à lui que notre servlet est capable de faire suivre nos objets requête et réponse à une vue. Il est impératif d'y préciser le chemin complet vers la JSP, commençant obligatoirement par un / (voir l'avertissement et la précision ci-dessous).
- nous utilisons enfin ce dispatcher pour réexpédier la paire requête/réponse HTTP vers notre page JSP via sa méthode `forward()`.

De cette manière notre page JSP devient accessible au travers de la servlet : d'ailleurs, notre servlet ne faisant actuellement rien d'autre, son seul rôle est de transférer le couple requête reçue & réponse vers la JSP finale.

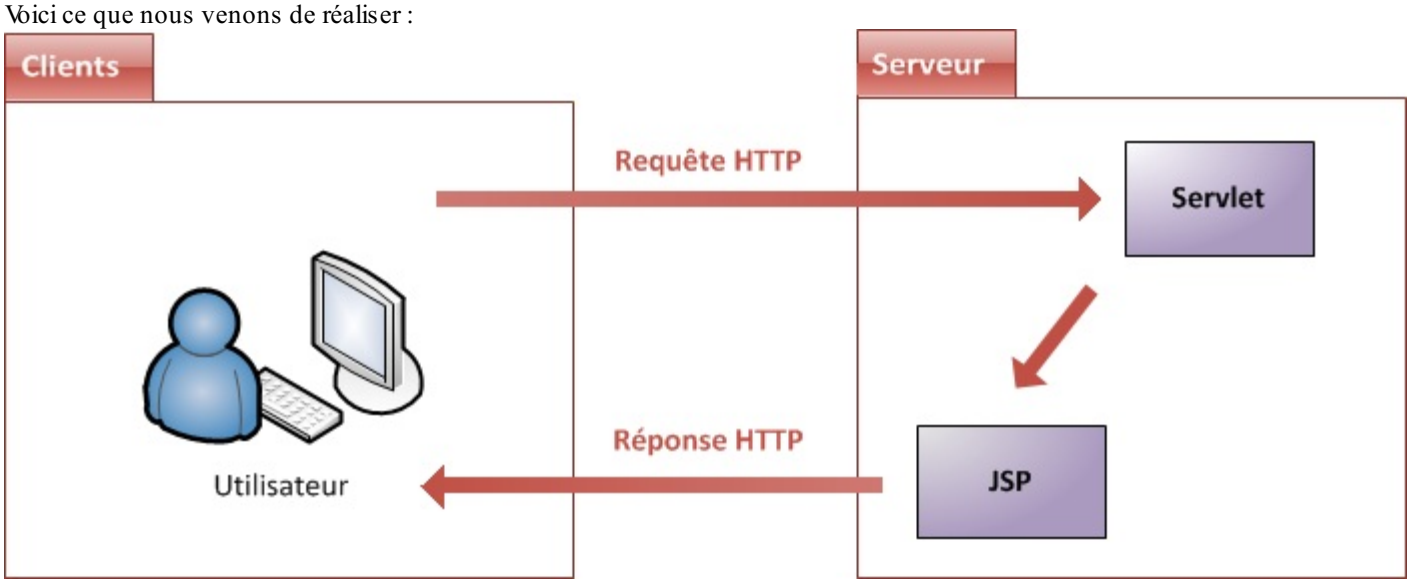
Ne soyez pas leurrés : comme je vous l'ai déjà dit lorsque je vous ai présenté la structure d'un projet, le dossier **WebContent** existe uniquement dans Eclipse ! Je vous ai déjà expliqué qu'il correspondait en réalité à la racine de l'application, et c'est donc pour ça qu'il faut bien écrire **/WEB-INF/test.jsp** en argument de la méthode `getRequestDispatcher()`, et non pas **/WebContent/WEB-INF/test.jsp** !

À retenir donc : nulle part dans votre code ne doit être mentionné le répertoire WebContent ! C'est une représentation de la racine de l'application, propre à Eclipse uniquement.

Si vous parcourez la documentation de l'objet `HttpServletRequest`, vous remarquerez qu'il contient lui aussi une méthode `getRequestDispatcher()` ! Toutefois, cette dernière présente une différence notable : elle peut prendre en argument un chemin relatif, alors que sa grande soeur n'accepte qu'un chemin complet. Je vous conseille, afin d'éviter de vous emmêler les crayons, de passer par la méthode que je vous présente ci-dessus. Quand vous serez plus à l'aise avec ces histoires de chemins relatifs et absolus, vous pourrez alors décider d'utiliser l'une ou l'autre de ces méthodes.

Faites à nouveau le test, en essayant cette fois d'appeler l'URL correspondant à votre servlet (souvenez-vous, c'est celle que nous avons mise en place dans le fichier web.xml, à savoir /test/toto) : tout fonctionne, notre requête est bien acheminée jusqu'à notre JSP, et en retour notre navigateur nous affiche bien le contenu de notre page !

Voici ce que nous venons de réaliser :



Notre projet commence à prendre forme. Dans le chapitre suivant, nous allons apprendre à faire circuler des données entre les différents acteurs de notre application !

Transmission de données

Nous nous sommes jusqu'à présent contentés d'afficher une page web au contenu figé, comme nous l'avions fait via une simple page HTML écrite en dur en tout début de cours. Seulement cette fois, notre contenu est présent dans une page JSP à laquelle nous avons associé une servlet, nous disposons ainsi de tout ce qui est nécessaire pour ajouter du dynamisme à notre projet. Il est donc grand temps d'apprendre à faire communiquer entre eux les différents éléments constituant notre application !

Données issues du serveur : les attributs

Transmettre des variables de la servlet à la JSP

Jusqu'à présent nous n'avons pas fait grand chose avec notre requête HTTP, autrement dit avec notre objet `HttpServletRequest` : nous nous sommes contentés de le transmettre à la JSP. Pourtant, vous avez dû vous en apercevoir lorsque vous avez parcouru sa documentation, ce dernier contient énormément de méthodes !

Puisque notre requête HTTP passe maintenant au travers de la servlet avant d'être transmise à la vue, profitons-en pour y apporter quelques modifications ! Utilisons donc notre servlet pour mettre en place un semblant de dynamisme dans notre application : créons une chaîne de caractères depuis notre servlet, et transmettons-la à notre vue pour affichage.

Code : Java - Transmission d'une variable

```
public void doGet( HttpServletRequest request, HttpServletResponse
response ) throws ServletException, IOException{
 String message = "Transmission de variables : OK !";
request.setAttribute( "test", message );
 this.getServletContext().getRequestDispatcher( "/WEB-INF/test.jsp"
).forward( request, response );
}
```

Comme vous pouvez le constater, le principe est très simple et tient en une ligne (surlignée dans le code) : il suffit d'appeler la méthode `setAttribute()` de l'objet requête pour y enregistrer un attribut ! Cette méthode prend en paramètre le nom que l'on souhaite donner à l'attribut suivi de l'objet en lui-même.

Ici, l'attribut que j'ai créé est une simple chaîne de caractères - un objet de type String - que j'ai choisi de nommer **test** lors de son enregistrement dans la requête.

Ne confondez pas le nom que vous donnez à votre objet au sein du code et le nom que vous donnez à l'attribut au sein de la requête.
Ici mon objet se nomme **message**, mais j'ai par la suite nommé **test** l'attribut qui contient cet objet dans la requête. Côté vue, c'est par ce nom d'attribut que vous pourrez accéder à votre objet !

C'est tout ce qu'il est nécessaire de faire côté servlet. Regardons maintenant comment récupérer et afficher l'objet côté vue :

Code : JSP - /WEB-INF/test.jsp

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Test</title>
    </head>
    <body>
        <p>Ceci est une page générée depuis une JSP.</p>
        <p>
<%
String attribut = (String) request.getAttribute("test");
out.println( attribut );
%>
        </p>
    </body>
</html>
```

Ne paniquez pas, vous ne connaissez pas cette notation et c'est bien normal puisque je ne vous l'ai pas encore présentée...

Voici donc une première information concernant la technologie JSP : elle permet d'inclure du code Java dans une page en entourant ce code des balises `<% et %>`. Ce sont des marqueurs qui délimitent les portions contenant du code Java du reste de la page, contenant ici simplement des balises HTML et du texte. À l'intérieur, tout se passe comme si on écrivait du code directement dans une servlet : on fait appel à la méthode `println()` de l'objet `PrintWriter` **out** pour afficher du contenu. **La seule différence réside dans le fait que depuis une JSP, il n'est plus nécessaire de spécifier le content-type de la réponse HTTP ni d'y récupérer l'objet PrintWriter, comme nous l'avions fait deux chapitres auparavant depuis notre servlet.** Ceci est rendu possible grâce à l'existence d'**objets implicites**, sur lesquels nous allons revenir très bientôt !

En ce qui concerne la récupération de l'attribut depuis la requête, je pense que vous êtes assez grands pour faire vous-mêmes l'analogie avec sa création : tout comme il suffit d'appeler `setAttribute()` pour créer un attribut dans une requête depuis une servlet, il suffit d'appeler la méthode `getAttribute()` pour en récupérer un depuis une JSP ! Ici, je récupère bien mon objet nommé **test** .

Nous avons ici transmis un simple objet String, mais il est possible de transmettre n'importe quel objet, comme un entier ou une liste par exemple.
Remarquez à ce sujet la nécessité de convertir (*cast*) l'objet récupéré dans la JSP au type souhaité, la méthode getAttribute() renvoyant un objet global de type Object.

Alors c'est ça une JSP ? Une autre page dans laquelle on remet une couche de Java ?

Non, bien sûr que non ! Écrire du Java dans une JSP n'a aucun sens : l'intérêt même de ces pages est justement de s'affranchir du langage Java ! À ce compte-là, autant n'utiliser qu'une servlet et ne pas mettre en place de JSP...

Cependant comme vous pouvez le voir, cela fonctionne très bien ainsi : ça confirme ce que je vous disais dans la première partie de ce cours, en Java EE rien n'impose au développeur de bien travailler, et il est possible de coder n'importe comment sans que cela n'impacte le fonctionnement de l'application. Voilà donc un premier exemple destiné à vous faire comprendre dès maintenant que mettre du code Java dans une page JSP, c'est mal.

Pour ce qui est de notre exemple, ne vous y méprenez pas : si je vous fais utiliser du code Java ici, c'est uniquement parce que nous n'avons pas encore découvert le langage JSP. D'ailleurs, autant vous prévenir tout de suite : à partir du chapitre suivant, **nous allons tout mettre en œuvre pour ne plus jamais écrire de Java directement dans une JSP !**

Données issues du client : les paramètres

Qu'est-ce qu'un paramètre de requête ?

Si vous êtes assidus, vous devez vous souvenir de la description que je vous ai faite de la méthode GET du protocole HTTP : elle permet au client de transmettre des données au serveur en les incluant directement dans l'URL, dans ce qui s'appelle les paramètres ou *query strings* en anglais. Eh bien c'est cela que nous allons apprendre à manipuler ici : nous allons rendre notre projet interactif, en autorisant le client à transmettre des informations au serveur.

La forme de l'URL

Les paramètres sont transmis au serveur directement via l'URL. Voici des exemples des différentes formes qu'une URL peut prendre :

Code : HTML

```
<!-- URL sans paramètres -->
/page.jsp

<!-- URL avec un paramètre nommé 'cat' et ayant pour valeur 'java'
-->
/page.jsp?cat=java

<!-- URL avec deux paramètres nommés 'lang' et 'admin', et ayant
pour valeur respectivement 'fr' et 'true' -->
/page.jsp?lang=fr&admin=true
```

Il y a peu de choses à retenir :

- le premier paramètre est séparé du reste de l'URL par le caractère ?
- les paramètres sont séparés entre eux par le caractère &
- une valeur est attribuée à chaque paramètre via l'opérateur =

Il n'existe pas d'autre moyen de déclarer des paramètres dans une requête GET, ceux-ci doivent impérativement apparaître en clair dans l'URL demandée. À ce propos, souvenez-vous de ce dont je vous avais averti lors de la présentation de cette méthode GET : la taille d'une URL étant limitée, la taille des données qu'il est ainsi possible d'envoyer est limitée également !

À vrai dire, la norme ne spécifie pas de limite à proprement parler, mais les navigateurs imposent d'eux mêmes une limite : par exemple, la longueur maximale d'une URL est de 2 083 caractères dans Internet Explorer 8. Au delà de ça, autrement dit si votre URL est si longue qu'elle contient plus de 2 000 caractères, ce navigateur ne saura pas gérer cette URL ! Vous disposez donc d'une certaine marge de manœuvre, pour des chaînes de caractères courtes comme dans notre exemple cette limite ne vous gêne absolument pas. Mais d'une manière générale et même si les navigateurs récents savent gérer des URL bien plus longues, **lorsque vous avez beaucoup de contenu à transmettre ou que vous ne connaissez pas à l'avance la taille des données qui vont être envoyées par le client, préférez la méthode POST.**

J'en profite enfin pour vous reparler des recommandations d'usage HTTP : lorsque vous envoyez des données au serveur et qu'elles vont avoir **un impact sur la ressource demandée**, il est là encore préférable de passer par la méthode POST du protocole, plutôt que par la méthode GET.

Que signifie "avoir un impact sur la ressource" ?

Eh bien cela veut dire "entraîner une modification sur la ressource", et en fin de compte tout dépend de ce que vous faites de ces données dans votre code. Prenons un exemple concret pour bien visualiser : imaginons une application proposant une page compte.jsp qui autoriserait des actions diverses sur le compte en banque de l'utilisateur. Ces actions ne se dérouleraient bien évidemment pas comme cela dans une vraie application bancaire, mais c'est simplement pour que l'exemple soit parlant :

- si le code attend un paramètre précisant le mois pour lequel l'utilisateur souhaite afficher la liste des entrées et sorties d'argent de son compte, par exemple compte.jsp?mois=avril, alors cela n'aura pas d'impact sur la ressource. En effet, nous pouvons bien renvoyer 10 fois la requête au serveur, notre code ne fera que ré-afficher les mêmes données à l'utilisateur sans les modifier.

- si par contre le code attend des paramètres précisant des informations nécessaires en vue de réaliser un transfert d'argent, par exemple compte.jsp?montant=100&destinataire=01K87B612, alors cela aura clairement un impact sur la ressource : en effet, si nous renvoyons 10 fois une telle requête, notre code va effectuer 10 fois le transfert !

Ainsi, si nous suivons les recommandations d'usage, nous pouvons utiliser une requête GET pour le premier cas, et devons utiliser une requête POST pour le second. Nous reviendrons sur les avantages de la méthode POST lorsque nous aborderons les formulaires, dans une des parties suivantes de ce cours.

N'importe quel client peut-il envoyer des paramètres à une application ?

Oui, effectivement. Par exemple, lorsque vous naviguez sur le site du zéro, rien ne vous empêche de rajouter des paramètres tout droit issus de votre imagination lors de l'appel de la page d'accueil du site : par exemple, www.siteduzero.com/?mascotte=zozor. Le site n'en fera rien car la page n'en tient pas compte, mais le serveur les recevra bien. C'est en partie pour cela, mais nous aurons tout le loisir d'y revenir par la suite, qu'il est impératif de bien vérifier le contenu des paramètres envoyés au serveur avant de les utiliser.

Récupération des paramètres par le serveur

Modifions notre exemple afin d'y inclure la gestion d'un paramètre nommé "auteur" :

Code : Java - Servlet

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

```
String paramAuteur = request.getParameter( "auteur" );
 String message = "Transmission de variables : OK ! " + paramAuteur;
 request.setAttribute( "test", message );

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

La seule ligne nécessaire pour cela est surlignée dans le code : il suffit de faire appel à la méthode `getParameter()` de l'objet requête, en lui passant comme argument le nom du paramètre que l'on souhaite récupérer. La méthode retournant directement le contenu du paramètre, je l'ai ici inséré dans une `String` que j'ai nommée **paramAuteur**.

Pour vous montrer que notre servlet récupère bien les données envoyées par le client, j'ai ajouté le contenu de cette `String` au message que je transmets ensuite à la JSP pour affichage. Si vous appelez à nouveau votre servlet depuis votre navigateur, rien ne va changer. Mais si cette fois vous l'appelez en ajoutant un paramètre nommé auteur à l'URL, par exemple :

Code : URL

```
http://localhost:8080/test/toto?auteur=Coyote
```

Alors vous observerez que le message affiché dans le navigateur contient bien la valeur du paramètre précisé dans l'URL :

Code : HTML - Contenu de la page finale

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Test</title>
    </head>
    <body>
        <p>Ceci est une page générée depuis une JSP.</p>
        <p>
            Transmission de variables : OK ! Coyote
        </p>
    </body>
</html>
```

Vous avez donc ici la preuve que votre paramètre à bien été récupéré par la servlet. Comprenez également que lorsque vous envoyez un paramètre, il reste présent dans la requête HTTP durant tout son cheminement. Par exemple, nous pouvons très bien y accéder depuis notre page JSP sans passer par la servlet, de la manière suivante :

Code : JSP - /WEB-INF/test.jsp

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Test</title>
    </head>
    <body>
        <p>Ceci est une page générée depuis une JSP.</p>
        <p>
            <%
            String attribut = (String) request.getAttribute("test");
            out.println( attribut );

String parametre = request.getParameter( "auteur" );
out.println( parametre );
            %>
        </p>
    </body>
```

```
    </html>
```

En procédant ainsi, lorsque nous appelons l'URL en y précisant un paramètre nommé auteur, nous obtenons bien le résultat escompté : notre JSP affiche une seconde fois "Coyote", cette fois en récupérant directement la valeur du paramètre depuis la requête HTTP.

Si vous ne précisez pas de paramètre auteur dans l'URL, alors la méthode `getParameter()` renverra **`null`** en lieu et place du contenu attendu.

Note : il est très imprudent de procéder à l'utilisation ou l'affichage d'un paramètre transmis par le client sans en contrôler et éventuellement sécuriser son contenu auparavant. Ici il s'agit d'un simple exemple, nous ne nous préoccupons pas encore des problèmes potentiels. Mais c'est une bonne pratique de toujours contrôler ce qu'on affiche au client. Nous y reviendrons à plusieurs reprises dans la suite du cours dans des cas plus concrets, et vous comprendrez alors mieux de quoi il retourne.

Nous allons nous arrêter là pour le moment. L'utilisation la plus courante des paramètres dans une application web est la récupération de données envoyées par le client via des formulaires, mais nous ne sommes pas encore prêts pour cela. Avant de passer à la suite, une dernière petite précision s'impose.

Quelle est la différence entre ces paramètres et les attributs que nous avons découverts en début de chapitre ?

Il ne faut pas faire de confusion ici :

- **les paramètres de requête** sont un concept appartenant au protocole HTTP. Ils sont envoyés par le client au serveur directement au sein de l'URL, et donc sous forme de chaînes de caractères. Il n'est pas possible de forger des paramètres dans l'objet `HttpServletRequest`, il est uniquement possible d'y accéder en lecture. Ce concept n'étant absolument pas spécifique à la plate-forme Java EE mais commun à toutes les technologies web, il ne peut pas être "objectifié" : c'est la raison pour laquelle la méthode `getParamater()` retourne quoi qu'il arrive un objet de type `String`, et il n'est pas possible d'ajouter une quelconque logique supplémentaire à un tel objet.

- **les attributs de requête** sont un concept appartenant au conteneur Java, et sont donc créés côté serveur : c'est au sein du code de l'application que l'on procède à leur initialisation, et qu'on les insère dans la version "objectifiée" de la requête, à savoir l'objet `HttpServletRequest`. Contrairement aux paramètres, ils ne sont pas présents directement dans la requête HTTP mais uniquement dans l'objet Java qui l'enveloppe, et peuvent contenir n'importe quel type de données. Ils sont utilisés pour permettre à une servlet de communiquer avec d'autres servlets ou pages JSP.

En résumé, les **paramètres** de requête sont propres au protocole HTTP et font partie intégrante de l'URL d'une requête, alors que les **attributs** sont des objets purement Java créés et gérés par le biais du conteneur.

Prochaine étape : l'apprentissage de la technologie JSP. Les prochains chapitres y sont entièrement consacrés, avec au programme la découverte de la syntaxe, des balises, etc. Bref, tout le nécessaire pour créer des vues dynamiques sans avoir à écrire de Java (ou presque) !

Le JavaBean

Ce court chapitre a pour unique objectif de vous présenter un type d'objet un peu particulier : le **JavaBean**. Souvent raccourci "bean", un JavaBean désigne tout simplement un composant réutilisable. Il est construit selon certains standards, définis dans les spécifications de la plate-forme et du langage Java eux-mêmes : un bean n'a donc rien de spécifique au Java EE.

Autrement dit, aucun concept nouveau n'intervient dans la création d'un bean : si vous connaissez les bases du langage Java, vous êtes déjà capables de comprendre et de créer un bean sans problème particulier. Son utilisation ne requiert aucune bibliothèque ; de même, il n'existe pas de superclasse définissant ce qu'est un bean, ni d'API.

Ainsi, **tout objet conforme à ces quelques règles peut être appelé un bean**. Découvrons pour commencer quels sont les objectifs d'un bean, puis quels sont ces standards d'écriture dont je viens de vous parler. Enfin, découvrons comment l'utiliser dans un projet !

Objectifs

Avant d'étudier sa structure, intéressons nous au pourquoi d'un bean. En réalité, un bean est un simple objet Java qui suit certaines contraintes.

Voici un récapitulatif des principaux concepts mis en jeu. Je vous donne ici des définitions plutôt abstraites, mais il faut bien passer par là :

- **les propriétés** : un bean est conçu pour être **paramétrable**. On appelle "propriétés" les champs non publics présents dans un bean. Qu'elles soient de type primitif ou objets, les propriétés permettent de paramétrer le bean, en y stockant des données.

- **la sérialisation** : un bean est conçu pour pouvoir être **persistant**. La sérialisation est un processus qui permet de sauvegarder l'état d'un bean, et donne ainsi la possibilité de le restaurer par la suite. Ce mécanisme permet une persistance des données, voire de l'application elle-même.

- **la réutilisation** : un bean est un composant conçu pour être **réutilisable**. Ne contenant que des données ou du code métier, un tel composant n'a en effet pas de lien direct avec la couche de présentation, et peut également être distant de la couche d'accès aux données (nous verrons cela avec le modèle de conception DAO). C'est cette indépendance qui lui donne ce caractère réutilisable.

- **l'introspection** : un bean est conçu pour être **paramétrable de manière dynamique**. L'introspection est un processus qui permet de connaître le contenu d'un composant (attributs, méthodes et événements) de manière dynamique, sans disposer de son code source. C'est ce processus couplé à certaines règles de normalisation qui rend possible une découverte et un paramétrage dynamique du bean !

Dans le cas d'une application Java EE, oublions les concepts liés aux événements, ceux-ci ne nous concernent pas. Tout le reste est valable, et permet de construire des applications de manière efficace : la simplicité inhérente à la conception d'un bean rend la construction d'une application basée sur des beans relativement aisée, et le caractère réutilisable d'un bean permet de **minimiser les duplications de logique dans une application**.

Structure

Un bean :

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

Voici un exemple illustrant cette structure :

Code : Java - Exemple

```
/* Cet objet est une classe publique */
public class MonBean{
 /* Cet objet ne possède aucun constructeur, Java lui assigne donc
```

```
un constructeur par défaut public et sans paramètre. */

 /* Les champs de l'objet ne sont pas publics (ce sont donc des
propriétés) */
 private String proprieteNumero1;
 private int proprieteNumero2;

 /* Les propriétés de l'objet sont accessibles via des getters et
setters publics */
 public String getProprieteNumero1() {
  return this.proprieteNumero1;
 }

 public int getProprieteNumero2() {
  return this.proprieteNumero2;
 }

 public void setProprieteNumero1( String proprieteNumero1 ) {
  this.proprieteNumero1 = proprieteNumero1;
 }

 public void setProprieteNumero2( int proprieteNumero2 ) {
  this.proprieteNumero2 = proprieteNumero2;
 }

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

Ce paragraphe se termine déjà : comme je vous le disais en introduction, un bean ne fait rien intervenir de nouveau. Voilà donc tout ce qui définit un bean, c'est tout ce que vous devez savoir et retenir. En outre, nous n'allons pas utiliser la sérialisation dans nos projets : si vous n'êtes pas familiers avec le concept, ne vous arrachez pas les cheveux et mettez cela de côté !

Plutôt que de paraphraser, passons directement à la partie qui nous intéressera dans ce cours, à savoir la mise en place de beans dans notre application web !

Mise en place

J'imagine que certains d'entre vous, ceux qui n'ont que très peu voire jamais développé d'applications Java ou Java EE, peinent à comprendre exactement à quel niveau et comment nous allons faire intervenir un objet java dans notre projet web. Voyons donc tout d'abord comment mettre en place un bean dans un projet web sous Eclipse, afin de le rendre utilisable depuis le reste de notre application.

Création de notre bean d'exemple

Définissons pour commencer un bean simple qui servira de base à nos exemples :

Code : Java - com.sdzee.beans.Coyote

```
package com.sdzee.beans;

public class Coyote {
 private String nom;
 private String prenom;
 private boolean genius;

 public String getNom() {
  return this.nom;
 }

 public String getPrenom() {
  return this.prenom;
 }

 public boolean isGenius() {
```

```
        return this.genius;
    }

    public void setNom( String nom ) {
        this.nom = nom;
    }

    public void setPrenom( String prenom ) {
        this.prenom = prenom;
    }

    public void setGenius( boolean genius ) {
        /* Wile E. Coyote fait toujours preuve d'une ingéniosité hors du
commun, c'est indéniable ! Bip bip... */
        this.genius = true;
    }
}
```

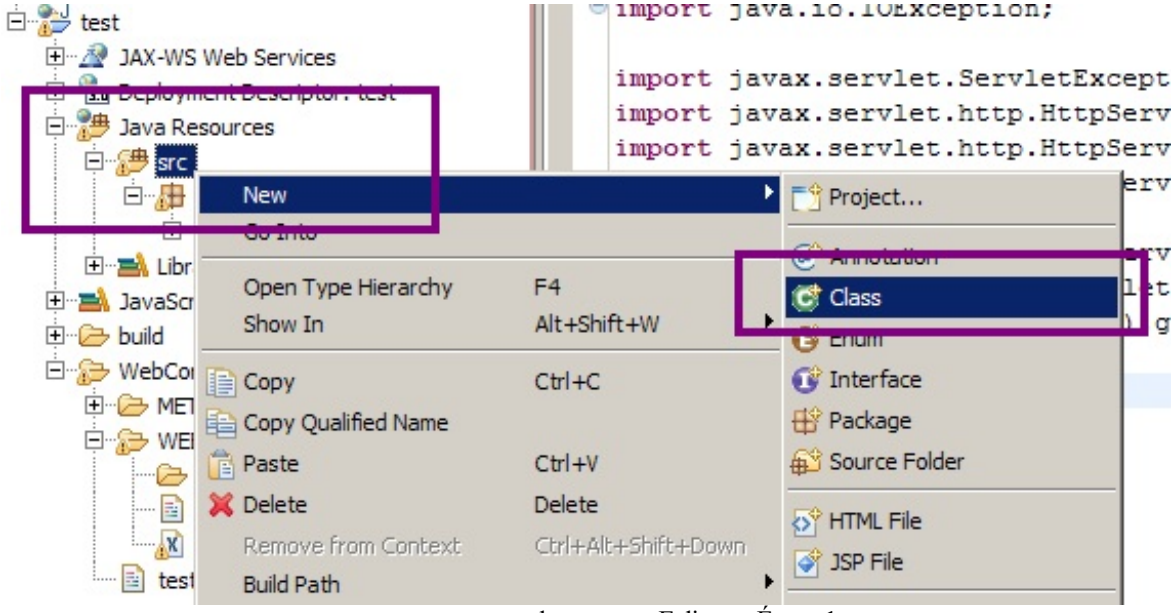
Rien de compliqué ici, c'est du pur Java sans aucune fioriture !

J'ai ici créé un bean contenant seulement trois propriétés, à savoir les trois champs non publics *nom*, *prenom* et *genius*. Inutile de s'attarder sur la nature des types utilisés ici, ceci n'est qu'un exemple totalement bidon qui ne sert à rien d'autre qu'à vous permettre de bien visualiser le concept.

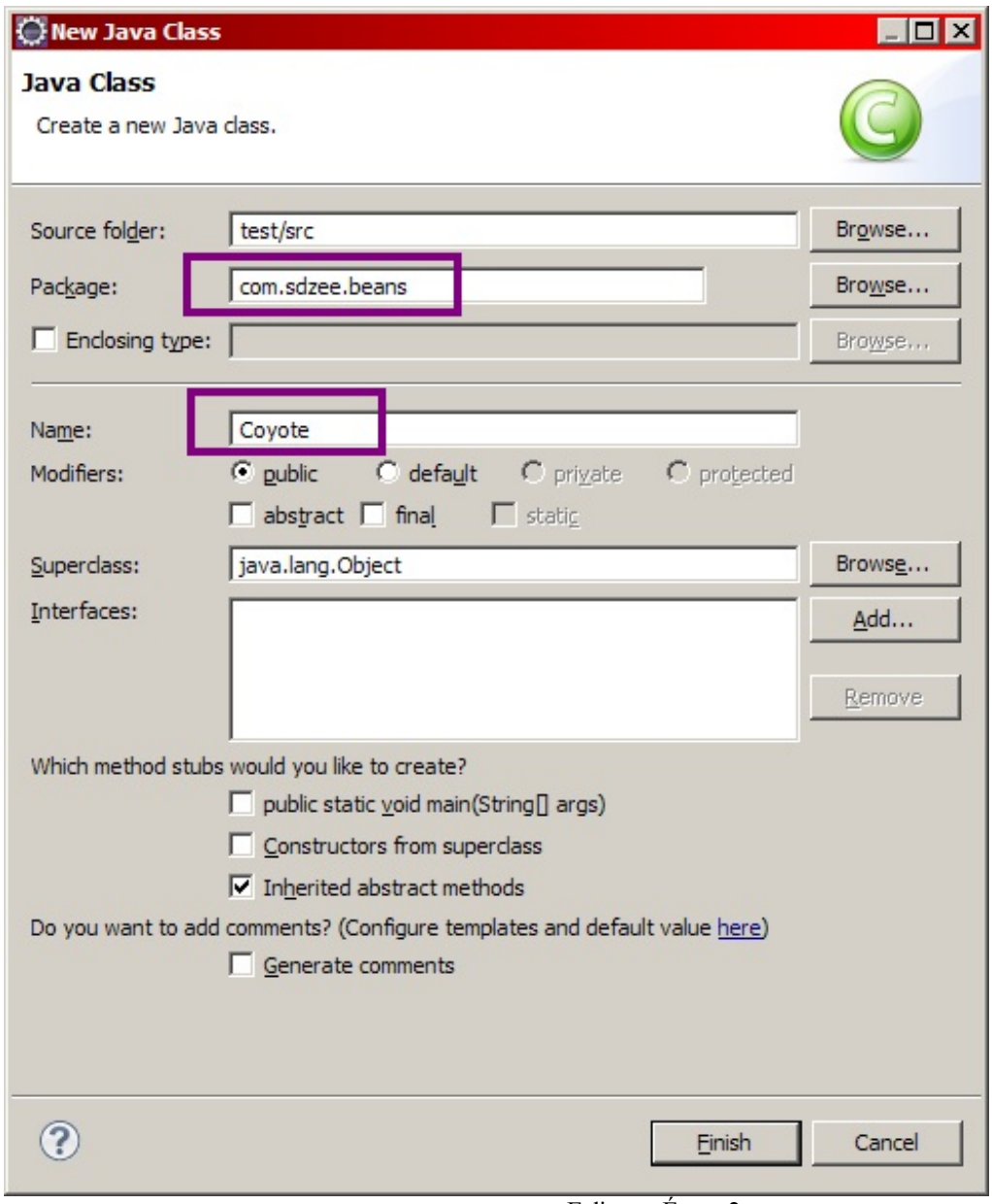
Maintenant, passons aux informations utiles. Vous pouvez remarquer que **cet objet respecte bien les règles qui régissent l'existence d'un bean** :

- un couple de getter/setter publics pour chaque champ privé ;
- aucun champ public ;
- un constructeur public sans paramètres (aucun constructeur tout court en l'occurrence).

Cet objet doit être placé dans le répertoire des sources "src" de notre projet web. J'ai ici dans notre exemple précisé le package `com.sdzee.beans`.
Jetez un œil aux copies d'écran ci-dessous pour visualiser la démarche sous Eclipse :



Mise en place d'un bean sous Eclipse - Étape 1



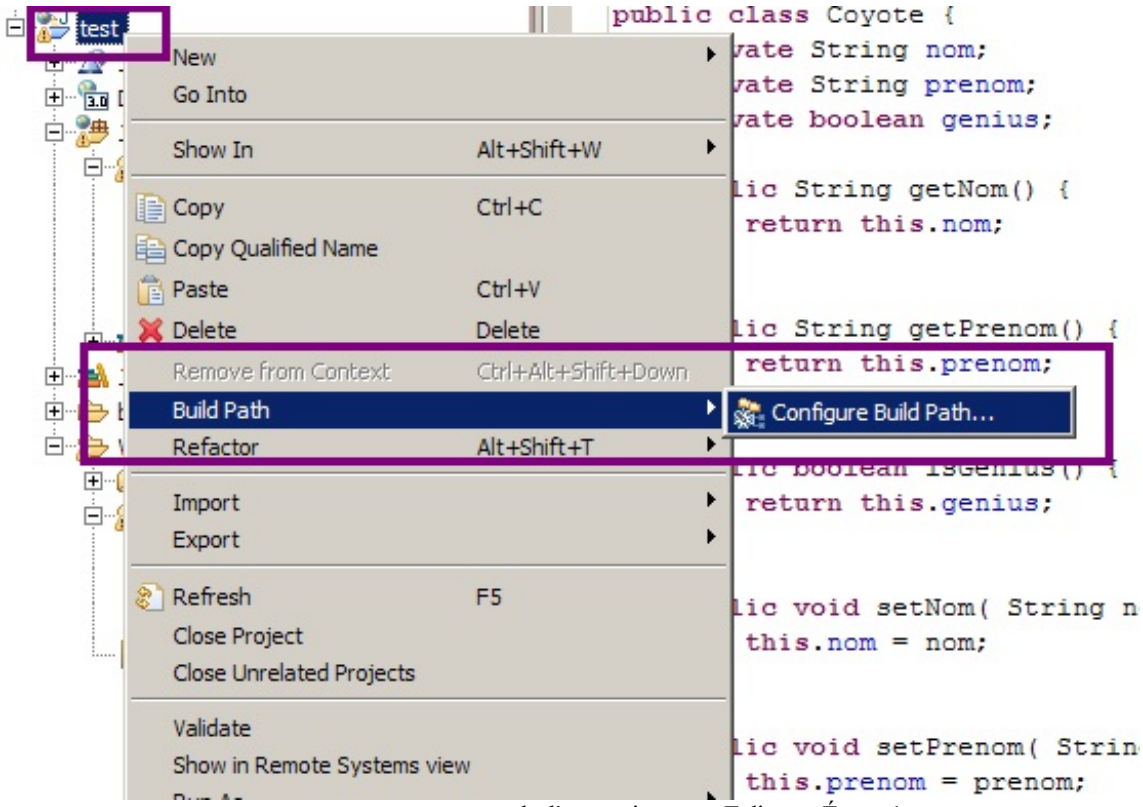
Mise en place d'un bean sous Eclipse - Étape 2

Vous savez dorénavant comment mettre en place des beans dans vos projets web. Cela dit, il reste encore une étape cruciale afin de rendre ces objets accessibles à notre application ! En effet, actuellement vous avez certes placé vos fichiers sources au bon endroit, mais vous savez très bien que votre application ne peut pas se baser sur ces fichiers sources, elle ne comprend que les classes compilées !

Configuration du projet sous Eclipse

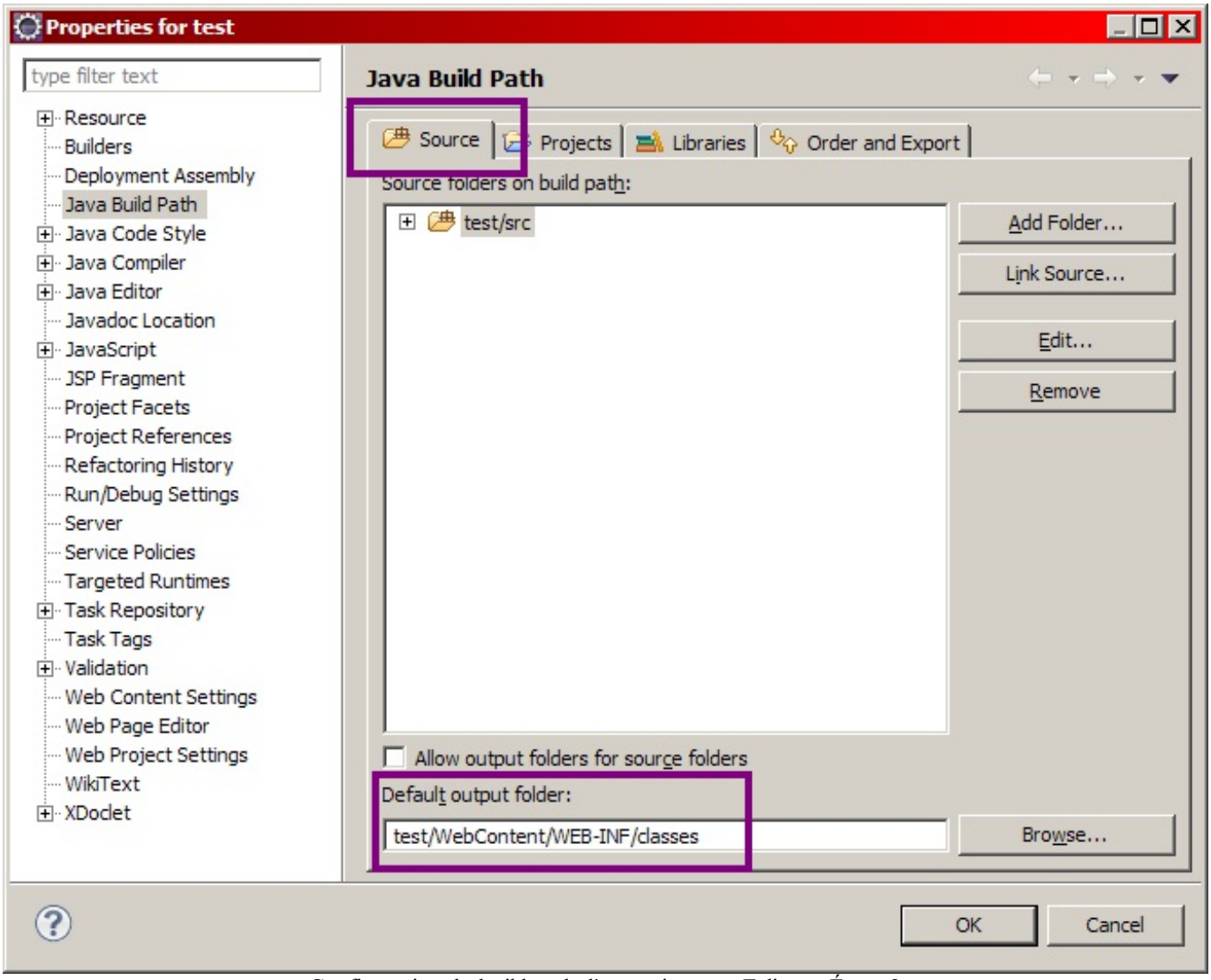
Afin de rendre vos objets accessibles à votre application, il faut que les classes compilées à partir de vos fichiers sources soient placées dans un dossier "classes", lui-même placé sous le répertoire **/WEB-INF**.
Par défaut Eclipse, toujours aussi fourbe, ne procède pas ainsi et envoie automatiquement vos classes compilées dans un dossier nommé "build".
Afin de changer de comportement, il va falloir modifier le *Build Path* de notre application.

Pour ce faire, faites un clic droit sur le dossier du projet, sélectionnez "Build Path" puis "Configure Build Path..." :



Configuration du build path d'un projet sous Eclipse - Étape 1

Sélectionnez alors l'onglet "source", puis regardez en bas le champ "Default output folder" :

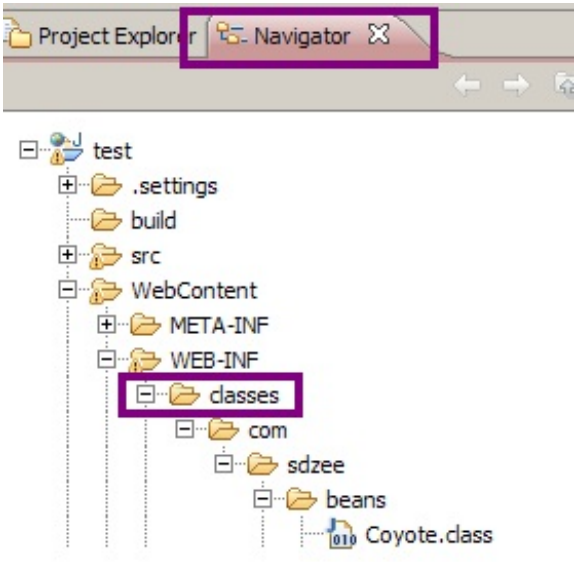


Configuration du build path d'un projet sous Eclipse - Étape 2

C'est ici qu'il faut préciser le chemin vers **WEB-INF/classes** afin que nos classes, lors de leur compilation, soient automatiquement déposées dans le dossier pris en compte par notre serveur d'applications. Le répertoire souhaité n'existant pas par défaut, Eclipse va le créer automatiquement pour nous.

Validez, et c'est terminé ! Votre application est prête, vos classes compilées seront bien déposées dans le répertoire de l'application, et vous allez ainsi pouvoir manipuler vos beans directement depuis vos servlets & JSP !

Par défaut, Eclipse ne vous montre pas ce répertoire "classes" dans l'arborescence de votre projet, simplement parce que ça n'intéresse pas le développeur de visualiser les fichiers .class, tout ce dont il a besoin depuis son IDE est de pouvoir travailler sur les fichiers sources .java ! Si toutefois vous souhaitez vérifier que le dossier est bien présent dans votre projet, il vous suffit d'ouvrir le volet "Navigator" :



Visualisation du répertoire contenant les classes sous Eclipse

Mise en service dans notre application

Notre objet étant bien inséré dans notre application, nous pouvons commencer à le manipuler. Reprenons notre servlet d'exemple précédente :

Code : Java - com.sdzee.servlets.Test

```
...

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

 /* Création et initialisation du message. */
 String paramAuteur = request.getParameter( "auteur" );
 String message = "Transmission de variables : OK ! " + paramAuteur;

 /* Création du bean */
Coyote premierBean = new Coyote();
 /* Initialisation de ses propriétés */
premierBean.setNom( "Coyote" );
premierBean.setPrenom( "Wile E." );

 /* Stockage du message et du bean dans l'objet request */
 request.setAttribute( "test", message );
request.setAttribute( "coyote", premierBean );

 /* Transmission de la paire d'objets request/response à notre JSP
*/
 this.getServletContext().getRequestDispatcher( "/WEB-INF/test.jsp"
).forward( request, response );
}
```

Et modifions ensuite notre JSP pour qu'elle réalise l'affichage des propriétés du bean. Nous n'avons pas encore découvert le langage JSP, et ne savons pas encore comment récupérer proprement un bean... Utilisons donc une nouvelle fois, faute de mieux pour le moment, du langage Java directement dans notre page JSP :

Code : JSP - /WEB-INF/test.jsp

```
<!DOCTYPE html>
<html>
    <head>
```

```
            <meta charset="utf-8" />
            <title>Test</title>
        </head>
        <body>
            <p>Ceci est une page générée depuis une JSP.</p>
            <p>
                <%
                String attribut = (String) request.getAttribute("test");
                out.println( attribut );

                String parametre = request.getParameter( "auteur" );
                out.println( parametre );
                %>
            </p>
            <p>
                Récupération du bean :
<%
com.sdzee.beans.Coyote notreBean = (com.sdzee.beans.Coyote)
request.getAttribute("coyote");
out.println( notreBean.getPrenom() );
out.println( notreBean.getNom() );
%>
            </p>
        </body>
</html>
```

Remarquez ici la nécessité de préciser le chemin complet (incluant le package) afin de pouvoir utiliser notre bean de type `Coyote`.
Retournez alors sur votre navigateur et ouvrez http://localhost:8080/test/toto. Vous observez alors :

Citation

Ceci est une page générée depuis une JSP.

Transmission de variables : OK !

Récupération du bean : Wile E. Coyote

Tout se passe comme prévu : nous retrouvons bien les valeurs que nous avions données aux propriétés nom et prenom de notre bean, lors de son initialisation dans la servlet !
L'objectif de ce chapitre est modeste : je ne vous offre ici qu'une présentation concise de ce que sont les **beans**, de leur rôles et utilisations dans une application Java EE. Encore une fois, comme pour beaucoup de concepts intervenant dans ce cours, il faudrait un tutoriel entier pour aborder toutes leurs spécificités, et couvrir en détail chacun des points importants mis en jeu.

Retenez toutefois que l'utilisation des beans n'est absolument pas limitée aux applications web : on peut en effet trouver ces composants dans de nombreux domaines, notamment dans les solutions graphiques basées sur les composants Swing et AWT (on parle alors de composant visuel).

Maintenant que nous sommes au point sur le concept, revenons à nos moutons : il est temps d'**apprendre à utiliser un bean depuis une page JSP sans utiliser de code Java** !

La technologie JSP (1/2)

Cet ensemble est consacré à l'apprentissage de la technologie JSP : nous y étudierons la syntaxe des balises, directives et actions JSP ainsi que le fonctionnement des expressions EL, et enfin nous établirons une liste de documentations utiles sur le sujet. Trop volumineux pour entrer dans un unique chapitre, j'ai dû le scinder en deux chapitres distincts.

Ce premier opus a pour objectif de vous présenter les bases de la syntaxe JSP et ses actions dites standard, toutes illustrées par de brefs exemples.

Les balises

Balises de commentaire

Tout comme dans les langages Java et HTML, il est possible d'écrire des commentaires dans le code de vos pages JSP. Ils doivent être compris entre les balises `<%-- et --%>`. Vous pouvez les placer où vous voulez dans votre code source. Ils sont uniquement destinés au(x) développeur(s), et ne sont donc pas visibles par l'utilisateur final dans la page HTML générée :

Code : JSP

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Exemple</title>
    </head>
    <body>
<%-- Ceci est un commentaire JSP, non visible dans la page HTML finale. --%>
        <!-- Ceci est un simple commentaire HTML. -->
        <p>Ceci est un simple texte.</p>
    </body>
</html>
```

Balises de déclaration

Cette balise vous permet de déclarer une variable à l'intérieur d'une JSP. Vous savez déjà et nous y reviendrons par la suite qu'il est déconseillé d'écrire du code Java dans vos JSP, mais la balise existant, je préfère vous la présenter. Si vous tombez dessus un jour, ne soyez pas déstabilisés :

Code : JSP

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

Il est possible d'effectuer plusieurs déclarations au sein d'un même bloc. Ci-dessous, les déclarations d'une variable puis d'une méthode :

Code : JSP

```
<%! String test = null;

public boolean jeSuisUnZero() {
  return true;
}
%>
```

Balises de scriptlet

Derrière ce mot étrange, un mélange atroce entre "script" et "servlet", se cache simplement du code Java. Cette balise vous la connaissez déjà, puisque nous l'avons utilisée dans le chapitre précédent. Elle sert en effet à inclure du code Java au sein de vos

pages, mais tout comme la balise précédente, elle est à proscrire dans la mesure du possible ! À titre d'information seulement donc, voici le tag en question ici au sein d'une balise HTML **<form>** :

Code : JSP

```
<form action="/tirage" method="post">
<%
for(int i = 1; i < 3; i++){
out.println("Numéro " + i + ": <select name=\"number"+i+"\">");
for(int j = 1; j <= 10; j++){
out.println("<option value=\""+j+"\">"+ j + "</option>");
}
out.println("</select><br />");
}
%>
    <br />
    <input type="submit" value="Valider" />
</form>
```

Alors oui je sais, c'est un exemple très moche, car il y a du code Java dans une JSP, code qui contient à son tour des éléments de présentation HTML... Mais c'est vraiment juste pour l'exemple ! Je vous préviens : le premier que je vois coder comme ça, je le pends à un arbre !

Balises d'expression

La balise d'expression est en quelque sorte un raccourci de la scriptlet suivante :

Code : JSP

```
<% out.println("Bip bip !"); %>
```

Elle retourne donc simplement le contenu d'une chaîne. Voici sa syntaxe :

Code : JSP

```
<%= "Bip bip !" %>
```

Notez bien l'**absence de point virgule** lors de l'utilisation de ce raccourci.

Les directives

Les directives JSP permettent :

- d'importer un package ;
- d'inclure d'autres pages JSP ;
- d'inclure des bibliothèques de balises (nous y reviendrons dans un prochain chapitre) ;
- de définir des propriétés et informations relatives à une page JSP.

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

Directive taglib

Le code ci-dessous inclut une bibliothèque personnalisée nommée *maTagLib* :

Code : JSP

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

Je ne détaille pas, nous reviendrons plus tard sur ce qu'est exactement une bibliothèque et sur cet attribut "prefix".

Directive page

La directive page définit des informations relatives à la page JSP. Voici par exemple comment importer des classes Java :

Code : JSP

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

Ici, l'import de deux classes est réalisé : `List` et `Date`. Cette fonctionnalité n'est utile que si vous mettez en place du code Java dans votre page JSP, afin de rendre disponibles les différentes classes et interfaces des API Java. En ce qui nous concerne, puisque notre objectif est de faire disparaître le Java de nos vues, nous allons très vite apprendre à nous en passer !

D'autres options sont utilisables via cet balise **page**, comme le **contentType** ou l'activation de la session. Je ne m'attarde pas sur les détails ici, vous ne vous en servirez que dans des cas très spécifiques que nous découvrirons au cas par cas dans ce cours. Voici toutefois à titre d'information l'ensemble des propriétés accessibles via cette directive :

Code : JSP

```
<%@ page
          language="..."
          extends="..."
          import="..."
          session="true | false"
          buffer="none | 8kb | sizekb"
          autoFlush="true | false"
          isThreadSafe="true | false"
          isELIgnored ="true | false"
          info="..."
          errorPage="..."
          contentType="..."
          isErrorPage="true | false"
%>
```

Directive include

Lorsque vous développez une vue, elle correspond rarement à une JSP constituée d'un seul bloc. En pratique, il est très courant de découper littéralement une page web en plusieurs fragments, qui sont ensuite rassemblés dans la page finale à destination de l'utilisateur. Cela permet notamment de pouvoir réutiliser certains blocs dans plusieurs vues différentes ! Regardez par exemple le menu des cours sur le site du zéro : c'est un bloc à part entière, qui est réutilisé dans l'ensemble des pages du site.

Pour permettre un tel découpage, la technologie JSP met à votre disposition une balise qui inclut le contenu d'un autre fichier dans le fichier courant. Via le code suivant par exemple, vous allez inclure une page interne à votre application (en l'occurrence une page JSP nommée *uneAutreJSP*, mais cela pourrait très bien être une page HTML ou autre) dans votre JSP courante :

Code : JSP

```
<%@ include file="uneAutreJSP.jsp" %>
```

La subtilité à retenir, c'est que cette directive ne doit être utilisée que pour inclure du contenu "statique" dans votre page : l'exemple le plus courant pour un site web étant par exemple le *header* ou le *footer* de la page, très souvent identiques sur l'intégralité des pages du site.

Attention, ici quand je vous parle de contenu "statique", je n'insinue pas que ce contenu est figé et ne peut pas contenir de code dynamique... Non, si je vous parle d'inclusion "statique", c'est parce qu'en utilisant cette directive pour inclure un fichier, **l'inclusion est réalisée au moment de la compilation** ; par conséquent, si le code du fichier est

changé par la suite, les répercussions sur la page l'incluant n'auront lieu qu'après une nouvelle compilation !

Pour simplifier, **cette directive peut être vue comme un simple copier/coller d'un fichier dans l'autre** : c'est comme si vous preniez l'intégralité de votre premier fichier, et que vous le colliez dans le second. Vous pouvez donc bien visualiser ici qu'il est nécessaire de procéder à cette copie avant la compilation de la page : on ne va pas copier un morceau de page JSP dans une servlet déjà compilée...

Action standard include

Une autre balise d'inclusion dite "standard" existe, et permet d'inclure du contenu de manière "dynamique". Le contenu sera ici chargé à l'exécution, et non à la compilation comme c'est le cas avec la directive précédente :

Code : JSP

```
<%-- L'inclusion dynamique d'une page fonctionne par URL relative :
--%>
<jsp:include page="page.jsp" />

<%-- Son équivalent en code Java  est : --%>
<% request.getRequestDispatcher( "page.jsp" ).include( request,
response ); %>

<%-- Et il est impossible d'inclure une page externe comme
ci-dessous :  --%>
<jsp:include page="http://www.siteduzero.com" />
```

Cela dit, ce type d'inclusion a un autre inconvénient : il ne prend pas en compte les imports et inclusions faits dans la page réceptrice. Pour simplifier, prenons un exemple : si vous utilisez un type `List` dans une première page, et que vous comptez utiliser une liste dans une seconde page que vous souhaitez inclure dans cette première page, il vous faudra importer le type `List` dans cette seconde page...

Je vous ai perdus ? Voyons tout cela au travers d'un exemple très simple. Créez une page **test_inc.jsp** contenant le code suivant, sous le répertoire **WebContent** de votre projet Eclipse, c'est-à-dire à la racine de votre application :

Code : JSP - /test_inc.jsp

```
<%
 ArrayList<Integer> liste = new ArrayList<Integer>();
 liste.add( 12 );
 out.println( liste.get( 0 ) );
%>
```

Ce code ne fait qu'ajouter un entier à une liste vide, puis l'affiche. Cependant cette page ne contient pas de directive d'import, et ne peut par conséquent pas fonctionner directement : l'import de la classe `ArrayList` doit obligatoirement être réalisé auparavant pour que nous puissions l'utiliser dans le code. Si vous tentez d'accéder directement à cette page via http://localhost:8080/test/test_inc.jsp, vous aurez le droit à une jolie exception :

Citation : Exception

```
org.apache.jasper.JasperException: Unable to compile class for JSP:

An error occurred at line: 2 in the jsp file: /test_inc.jsp
ArrayList cannot be resolved to a type
```

Créez maintenant une page **test_host.jsp**, toujours à la racine de votre application, qui va réaliser l'import de la classe `ArrayList` puis inclure la page **test_inc.jsp** :

Code : JSP - test_host.jsp

```
<%@ page import="java.util.ArrayList" %>
<%@ include file="test_inc.jsp" %>
```

Pour commencer, vous découvrez ici en première ligne une application de la directive **page**, utilisée ici pour importer la classe `ArrayList`. À la seconde ligne, comme je vous l'ai expliqué plus haut, la directive d'inclusion peut être vue comme un copier/coller : ici, le contenu de la page **test_inc.jsp** est copié dans la page **test_host.jsp**, puis la nouvelle page **test_host.jsp** contenant tout le code est compilée. Vous pouvez donc appeler la page **test_host.jsp**, et la page web finale affichera bien "12" !

Mais si maintenant nous décidons de remplacer la directive présente dans notre page **test_host.jsp** par la balise standard d'inclusion :

Code : JSP - test_host.jsp

```
<%@ page import="java.util.ArrayList" %>
<jsp:include page="test_inc.jsp" />
```

Eh bien lorsque nous allons tenter d'accéder à la page **test_host.jsp**, nous retrouverons la même erreur que lorsque nous avons tenté d'accéder directement à **test_inc.jsp** ! La raison est la suivante : les deux pages sont compilées séparément, et ensuite seulement l'inclusion se fera lors de l'exécution. Ainsi fatalement, la compilation de la page **test_inc.jsp** ne peut qu'échouer, puisque l'import nécessaire au bon fonctionnement du code n'est réalisé que dans la page hôte.

Pour faire simple, les pages incluses via la balise **<jsp:include ... />** doivent en quelque sorte être "indépendantes", elles ne peuvent pas dépendre l'une de l'autre et doivent pouvoir être compilées séparément. Ce qui n'est pas le cas des pages incluses via la directive **<%@ include ... %>**.

Pour terminer sur ces problématiques d'inclusions, je vous donne ici quelques informations et conseils supplémentaires :

- certains serveurs d'application sont capables de recompiler une page JSP incluant une autre page via la directive d'inclusion, et ainsi éclipser sa principale contrainte. Ce n'est toutefois pas toujours le cas, et reste donc à éviter si vous n'êtes pas sûrs de votre coup...
- pour inclure un même *header* et un même *footer* dans toutes les pages de votre application ou site web, il est préférable de ne pas utiliser ces techniques d'inclusion, mais de spécifier directement ces portions communes dans le fichier **web.xml** de votre projet. J'en reparlerai dans un prochain chapitre.
- très bientôt, nous allons découvrir une meilleure technique d'inclusion de pages avec la JSTL !

La portée des objets

Un concept important intervient dans la gestion des objets par la technologie JSP : **la portée des objets**. Souvent appelée visibilité, ou *scope* en anglais, elle définit tout simplement leur **durée de vie**.

Dans le chapitre traitant de la transmission de données, nous avions découvert un premier type d'attributs : les attributs de requête. Eh bien de tels objets, qui je vous le rappelle sont accessibles via l'objet `HttpServletRequest`, ne sont **visibles** que durant le traitement d'une même requête. Ils sont créés par le conteneur lors de la réception d'une requête HTTP, et disparaissent dès lors que le traitement de la requête est terminé.

Ainsi, nous avions donc sans le savoir créé des objets ayant pour portée la requête !

Il existe au total quatre portées différentes dans une application :

- **page** (JSP seulement) : les objets dans cette portée sont uniquement accessibles dans la page JSP en question ;
- **requête** : les objets dans cette portée sont uniquement accessibles durant l'existence de la requête en cours ;
- **session** : les objets dans cette portée sont accessibles durant l'existence de la session en cours ;
- **application** : les objets dans cette portée sont accessibles durant toute l'existence de l'application.

Pourquoi préciser "JSP seulement" pour la portée page ?

Eh bien c'est très simple : il est possible de créer et manipuler des objets de portées requête, session ou application depuis une page JSP ou depuis une servlet. Nous avions d'ailleurs dans le chapitre traitant de la transmission de données créé un objet de

portée requête depuis notre servlet, puis utilisé cet objet depuis notre page JSP. En revanche, il n'est possible de créer et manipuler des objets de portée page que depuis une page JSP, ce n'est pas possible via une servlet.

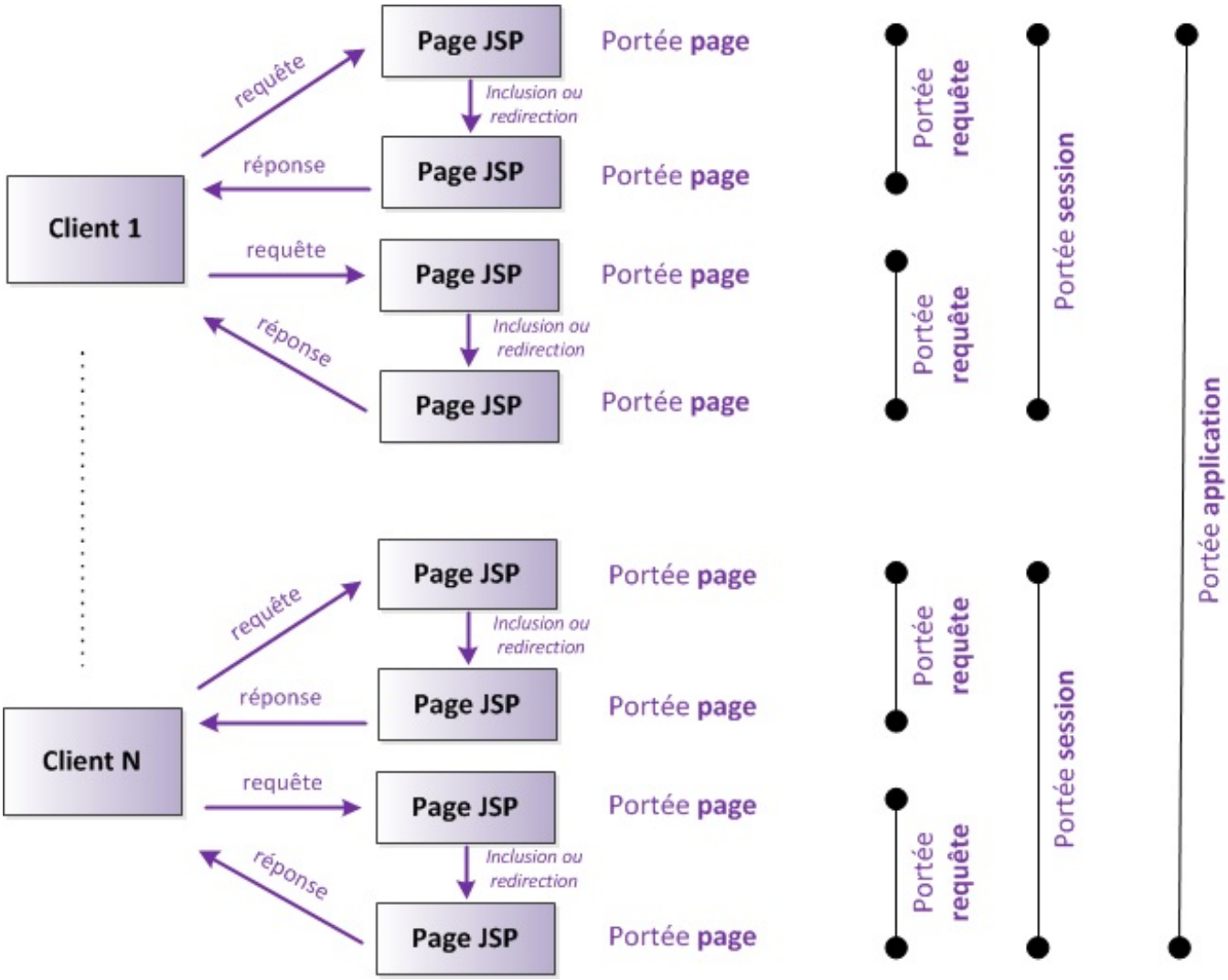
Qu'est-ce qu'une session ?

Une session est un objet associé à un utilisateur en particulier. Elle existe pour la durée pendant laquelle un visiteur va utiliser l'application, cette durée se terminant lorsque l'utilisateur ferme son navigateur, reste inactif trop longtemps, ou encore lorsqu'il se déconnecte du site.

Ainsi, il est possible de garder en mémoire des données concernant un visiteur d'une requête à l'autre, autrement dit de page en page : la session permet donc de garder une trace de la visite effectuée. Plus précisément, une session correspond en réalité à un navigateur particulier, plutôt qu'à un utilisateur : par exemple, si à un même instant vous utilisez deux navigateurs différents pour vous rendre sur le même site, le site créera deux sessions distinctes, une pour chacun des navigateurs.

Un objet session concernant un utilisateur est conservé jusqu'à ce qu'une certaine durée d'inactivité soit atteinte. Passé ce délai, le conteneur considère que ce client n'est plus en train de visiter le site, et détruit alors sa session.

Pour que vous visualisiez bien le principe, voici un schéma regroupant les différentes portées existantes :



Portées des objets

Remarquez bien les points suivants :

- un objet de **portée page** n'est accessible que sur une page JSP donnée ;
- un objet de **portée requête** n'est accessible que durant le cheminement d'une requête dans l'application, et n'existe plus dès lors qu'une réponse est renvoyée au client ;
- un objet de **portée session** est accessible durant l'intégralité de la visite d'un client donné, à condition bien sûr que le temps d'inactivité défini par le conteneur ne soit pas dépassé durant cette visite ;

- un objet de **portée application** est accessible durant toute l'existence de l'application et par tous les clients.

Vous devez bien réaliser que l'utilisation dans votre code d'objets ayant pour portée l'application est délicate. Rendez-vous compte : ces objets sont accessibles partout, tout le temps et par tout le monde ! Afin d'éviter notamment des problèmes de modifications concurrentes, si vous avez besoin de mettre en place de tels objets, il est recommandé de les initialiser dès le chargement de l'application, puis de ne plus toucher à leur contenu et d'y accéder depuis vos classes et pages **uniquement en lecture seule**. Nous étudierons ce scénario dans un prochain chapitre.

Nous reviendrons au cas par cas sur chacune de ces portées dans certains exemples des chapitres à venir.

Les actions standard

Autant vous prévenir tout de suite, le déroulement de ce chapitre peut vous perturber : je vais dans cette partie du chapitre vous présenter une certaine manière de faire pour accéder à des objets depuis une page JSP. Ensuite, je vais vous expliquer dans la partie suivante qu'il existe un autre moyen, plus simple et plus propre, et que nous n'utiliserons alors plus jamais cette première façon de faire...

Maintenant que vous connaissez les beans et les portées, vous avez presque tout en main pour constituer le modèle de votre application (le M de MVC) ! C'est lui et et uniquement lui qui va contenir les données de votre application, et les traitements à y appliquer. La seule chose qui vous manque encore, c'est la manipulation de ces beans depuis une page JSP.

Vous avez déjà fait connaissance avec l'action standard **`<jsp:include/>`**, je vais vous en présenter quatre autres : **`<jsp:useBean/>`**, **`<jsp:getProperty/>`**, **`<jsp:setProperty/>`** et enfin **`<jsp:forward/>`**.

L'action standard useBean

Voici pour commencer l'action standard permettant d'utiliser un bean, ou de le créer s'il n'existe pas, depuis une page JSP :

Code : JSP

```
<%-- L'action suivante récupère un bean de type Coyote et nommé
"coyote" dans
la portée requête s'il existe, ou en crée un sinon. --%>
<jsp:useBean id="coyote" class="com.sdzee.beans.Coyote"
scope="request" />

<%-- Elle a le même effet que le code Java suivant : --%>
<%
com.sdzee.beans.Coyote coyote = (com.sdzee.beans.Coyote)
request.getAttribute( "coyote" );
if ( coyote == null ){
    coyote = new com.sdzee.beans.Coyote();
    request.setAttribute( "coyote", coyote );
}
%>
```

Étudions les différents attributs de cette action :

- la valeur de l'attribut **id** est le nom du bean à récupérer, ou le nom que vous souhaitez donner au bean à créer.
- l'attribut **class** correspond logiquement à la classe du bean. Il doit obligatoirement être spécifié si vous souhaitez créer un bean, mais pas si vous souhaitez simplement récupérer un bean existant.
- l'attribut optionnel **scope** correspond à la portée de l'objet. Si un bean du nom spécifié en id **existe déjà** dans ce *scope*, et qu'il est du type ou de la classe précisé(e), alors il est récupéré, sinon une erreur survient. Si aucun bean de ce nom n'existe dans ce *scope*, alors un nouveau bean est créé. Enfin, **si cet attribut n'est pas renseigné, alors le *scope* par défaut sera limité à la page** en cours.
- l'attribut optionnel **type** doit indiquer le type de déclaration du bean. Il doit être une superclasse de la classe du bean, ou une interface implémentée par le bean. Cet attribut doit être spécifié si **class** ne l'est pas, et vice-versa.

En résumé, **cette action permet de stocker un bean (nouveau ou existant) dans une variable**, qui sera identifiée par la valeur saisie dans l'attribut **id**.

Il est également possible de donner un corps à cette balise, qui ne sera exécuté que si le bean est créé :

Code : JSP

```
<jsp:useBean id="coyote" class="com.sdzee.beans.Coyote">
   <%-- Ici, vous pouvez placer ce que vous voulez :
        définir des propriétés, créer d'autres objets, etc. --%>
   <p>Nouveau bean !</p>
</jsp:useBean>
```

Ici, le texte qui est présent entre les balises ne sera affiché que si un bean est bel et bien créé, autrement dit si la balise **<jsp:useBean>** est appelée avec succès. À l'inverse, si un bean du même nom existe déjà dans cette page, alors le bean sera simplement récupéré et le texte ne sera pas affiché.

L'action standard getProperty

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

Code : JSP

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

<%-- L'action suivante affiche le contenu de la propriété 'prenom'
du bean 'coyote' : --%>
<jsp:getProperty name="coyote" property="prenom" />

<%-- Elle a le même effet que le code Java suivant : --%>
<%= coyote.getPrenom() %>
```

Faites bien attention à la subtilité suivante ! Alors que **<jsp:useBean/>** récupère une instance dans une variable accessible par l'id défini, cette action standard ne récupère rien, mais réalise seulement l'affichage du contenu de la propriété ciblée. Deux attributs sont utiles ici :

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

L'action standard setProperty

Il est enfin possible de modifier une propriété du bean utilisé. Il existe pour cela quatre façons de faire via l'action standard dédiée à cette tâche :

Code : JSP - Syntaxe 1

```
<%-- L'action suivante associe une valeur à la propriété 'prenom' du
bean 'coyote' : --%>
<jsp:setProperty name="coyote" property="prenom" value="Wile E." />

<%-- Elle a le même effet que le code Java suivant : --%>
<% coyote.setPrenom("Wile E."); %>
```

Code : JSP - Syntaxe 2

```
<%-- L'action suivante associe directement la valeur récupérée
depuis le paramètre de la requête nommé ici 'prenomCoyote' à la
```

```
propriété 'prenom' : --%>
<jsp:setProperty name="coyote" property="prenom"
param="prenomCoyote"/>

<%-- Elle a le même effet que le code Java suivant : --%>
<% coyote.setPrenom( request.getParameter("prenomCoyote") ); %>
```

Code : JSP - Syntaxe 3

```
<%-- L'action suivante associe directement la valeur récupérée
depuis le paramètre de la requête nommé ici 'prenom' à la propriété
de même nom : --%>
<jsp:setProperty name="coyote" property="prenom" />

<%-- Elle a le même effet que le code Java suivant : --%>
<% coyote.setPrenom( request.getParameter("prenom") ); %>
```

Code : JSP - Syntaxe 4

```
<%-- L'action suivante associe automatiquement la valeur récupérée
depuis chaque paramètre de la requête à la propriété de même nom : -
-%>
<jsp:setProperty name="coyote" property="*" />

<%-- Elle a le même effet que le code Java suivant : --%>
<% coyote.setNom( request.getParameter("nom") ); %>
<% coyote.setPrenom( request.getParameter("prenom") ); %>
<% coyote.setGenius( (boolean) request.getParameter("genius") ); %>
```

L'action standard forward

La dernière action que nous allons découvrir permet d'effectuer une redirection vers une autre page. Comme toutes les actions standard, elle s'effectue côté serveur et pour cette raison **il est impossible via cette balise de rediriger vers une page extérieure à l'application**. L'action de *forwarding* est ainsi limitée aux pages présentes dans le contexte de la servlet ou JSP utilisée :

Code : JSP

```
<%-- Le forwarding vers une page de l'application fonctionne par URL
relative : --%>
<jsp:forward url="/page.jsp" />

<%-- Son équivalent en code Java  est : --%>
<% request.getRequestDispatcher( "/page.jsp" ).forward( request,
response ); %>

<%-- Et il est impossible de rediriger vers un site externe comme
ci-dessous :  --%>
<jsp:forward url="http://www.siteduzero.com" />
```

Une particularité du *forwarding* est qu'il n'implique pas d'aller/retour passant par le navigateur de l'utilisateur final. Autrement dit, l'utilisateur final n'est pas au courant que sa requête a été redirigée vers une ou plusieurs JSP différentes, puisque l'URL qui est affichée dans son navigateur ne change pas. Pas d'inquiétude, nous y reviendrons en détails lorsque nous étudierons un cas particulier, dans le chapitre concernant les sessions.

Sachez enfin que lorsque vous utilisez le *forwarding*, **le code présent après cette balise dans la page n'est pas exécuté.**

Je vous présente toutes ces notations afin que vous sachiez qu'elles existent, mais vous devez comprendre que la plupart de celles-ci étaient d'actualité... il y a une petite dizaine d'années maintenant ! Depuis, d'importantes évolutions ont changé la donne et tout cela n'est aujourd'hui utilisé que dans des cas bien spécifiques.

La vraie puissance de la technologie JSP, c'est dans le chapitre suivant que vous allez la découvrir !

La technologie JSP (2/2)

Nous allons dans ce chapitre terminer l'apprentissage de la technologie JSP, à travers la découverte des expression EL et des objets implicites.

Expression Language

Présentation

Dans cette seconde moitié, nous allons découvrir ensemble les bases de l'*Expression Language*, que l'on raccourcit très souvent EL.

Je répète ce dont je vous ai averti dans la partie précédente : une fois que vous aurez assimilé cette technologie, vous n'aurez plus jamais à utiliser **les actions standard d'utilisation des beans** que nous venons de découvrir ! Rassurez-vous, je ne vous les ai pas présentées juste pour le plaisir : il est important que vous connaissiez ce mode de fonctionnement, afin de ne pas être surpris si un jour vous tombez dessus. Seulement c'est une approche différente du modèle MVC, une approche qui n'est pas compatible avec ce que nous apprenons ici.

En quoi consistent les expressions EL ?

Ces expressions sont indispensables à une utilisation optimale des JSP. C'est grâce à elles que l'on peut s'affranchir définitivement de l'écriture de scriptlets (du code Java, pour ceux qui n'ont pas suivi) dans nos belles pages JSP. Pour faire simple et concis, **les expressions EL permettent via une syntaxe très épurée d'effectuer des tests basiques sur des expressions, et de manipuler simplement des objets et attributs dans une page, et ce sans nécessiter l'utilisation de code ni script Java** ! Cela facilite par conséquent grandement la maintenance de vos pages JSP, en fournissant des notations simples et surtout standard.

Avant tout, étudions la forme et la syntaxe d'une telle expression :

Code : JSP

```
${ expression }
```

Ce type de notation ne devrait pas être inconnu à ceux d'entre vous qui ont déjà programmé en Perl. Ce qu'il faut bien retenir, c'est que **ce qui est situé entre les accolades va être interprété** : lorsqu'il va analyser votre page JSP, le conteneur va repérer ces expressions entourées d'accolades et il saura ainsi qu'il doit en interpréter le contenu. Aussi, ne vous étonnez pas si dans la suite de ce chapitre j'évoque *l'intérieur d'une EL* : je parle tout simplement de ce qui est situé entre les accolades !

Les tests

Pour commencer, vous devez savoir qu'à l'intérieur d'une expression vous pouvez effectuer diverses sortes de tests. Pour réaliser ces tests, il vous est possible d'inclure certains opérateurs. Parmi ceux-ci, on retrouve les traditionnels :

- opérateurs logiques, applicables à des booléens : &&, ||, ! ;
- opérateurs arithmétiques, applicables à des nombres : +, -, *, /, % ;
- opérateurs relationnels, basés sur l'utilisation des méthodes `equals()` et `compareTo()` des objets comparés : == ou eq, != ou ne, < ou lt, > ou gt, <= ou le, >= ou ge.

Voyons concrètement ce que tout cela donne à travers quelques exemples. Créez une page nommée **test_el.jsp** à la racine de votre application, et placez-y ces quelques lignes :

Code : JSP - /test_el.jsp

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Test des expressions EL</title>
    </head>
```

```
    <body>
    <p>
        <!-- Logiques sur des booléens -->
        ${ true && true } <br /> <!-- Affiche true -->
        ${ true && false } <br /> <!-- Affiche false -->
        ${ !true || false } <br /> <!-- Affiche false -->

        <!-- Calculs arithmétiques -->
        ${ 10 / 4 } <br /> <!-- Affiche 2.5 -->
        ${ 10 mod 4 } <br /> <!-- Affiche le reste de la division
entière, soit 2 -->
        ${ 10 % 4 } <br /> <!-- Affiche le reste de la division
entière, soit 2 -->
        ${ 6 * 7 } <br /> <!-- Affiche 42 -->
        ${ 63 - 8 } <br /> <!-- Affiche 55 -->
        ${ 12 / -8 } <br /> <!-- Affiche -1.5 -->
        ${ 7 / 0 } <br /> <!-- Affiche Infinity -->

        <!-- Compare les caractères 'a' et 'b'. Le caractère 'a'
étant bien situé avant le caractère 'b' dans l'alphabet ASCII,
cette EL affiche true. -->
        ${ 'a' < 'b' } <br />

        <!-- Compare les chaînes 'hip' et 'hit'. Puisque 'p' < 't',
cette EL affiche false. -->
        ${ 'hip' gt 'hit' } <br />

        <!-- Compare les caractères 'a' et 'b', puis les chaînes
'hip' et 'hit'. Puisque le premier test renvoie true et le second
false, le résultat est false. -->
        ${ 'a' < 'b' && 'hip' gt 'hit' } <br />

        <!-- Compare le résultat d'un calcul à une valeur fixe.
Ici, 6 x 7 vaut 42 et non pas 48, le résultat est false. -->
        ${ 6 * 7 == 48 } <br />
    </p>
    </body>
</html>
```

Rendez-vous alors sur http://localhost:8080/test/test_el.jsp, et vérifiez les résultats obtenus.

Remarquez la subtilité dévoilée ici dans l'exemple de la ligne 27, au niveau des chaînes de caractères : contrairement à du code Java, dans lequel vous ne pouvez déclarer une `String` qu'en utilisant des *double quotes* (guillemets), vous pouvez utiliser également des *simple quotes* (apostrophes) dans une EL. Pour information, ceci a été rendu possible afin de simplifier l'intégration des expressions EL dans les balises JSP : celles-ci contenant déjà bien souvent leurs propres guillemets, cela évite au développeur de s'emmêler les crayons !

Attention ici aux **opérateurs relationnels** :

- si vous souhaitez vérifier l'égalité **d'objets de type non standard** via une EL, il vous faudra probablement ré-implémenter les méthodes citées dans le troisième point de la liste précédente ;
- dans le cas d'une comparaison, il vous faudra vérifier que votre objet implémente bien l'interface `Comparable`.

Pas besoin de vous casser la tête pour un objet de type `String` ou `Integer` donc, pour lesquels tout est déjà prêt nativement, mais pour des objets de types personnalisés, pensez-y !

Les opérateurs suivent comme toujours un ordre de priorité : comme on a dû vous l'apprendre en cours élémentaire, la multiplication est prioritaire sur l'addition, etc. J'omets ici volontairement certaines informations, notamment certains opérateurs que je ne juge pas utile de vous présenter ; je vous renvoie vers la documentation pour plus d'informations. Les applications que nous verrons dans ce cours ne mettront pas en jeu d'EL complexes, et vous serez par la suite assez à l'aise avec le concept pour comprendre par vous-même les subtilités de leur utilisation dans des cas plus élaborés.

En outre, deux autres types de test sont fréquemment utilisés au sein des expressions EL :

- les conditions ternaires, de la forme : `test ? sioui : sinon;`
- les vérifications si vide ou **null**, grâce à l'opérateur **empty**.

Très pratiques, ils se présentent sous cette forme :

Code : JSP

```
<!-- Vérifications si vide ou null -->
${ empty 'test' } <!-- La chaîne testée n'est pas vide, le résultat
est false -->
${ empty '' } <!-- La chaîne testée est vide, le résultat est true
-->
${ !empty '' } <!-- La chaîne testée est vide, le résultat est
false -->

<!-- Conditions ternaires -->
${ true ? 'vrai' : 'faux' } <!-- Le booléen testé vaut true, vrai
est affiché -->
${ 'a' > 'b' ? 'oui'  : 'non' } <!-- Le résultat de la comparaison
vaut false, non est affiché -->
${ empty 'test' ? 'vide' : 'non  vide'  } <!-- La chaîne testée
n'est pas vide, non vide est affiché -->
```

Les JavaBeans

Hormis la possibilité d'effectuer des tests, ce qui est réellement intéressant avec les expressions EL c'est le fait qu'elles sont basées sur les spécifications des JavaBeans. Qu'est-ce-que cela signifie concrètement ? Eh bien qu'il est possible via une expression EL de récupérer directement un attribut issu d'un bean ! Pour illustrer cette fonctionnalité, utilisons les actions standard que nous avons découvert dans le chapitre précédent pour mettre en place un exemple simple. Éditez le fichier **test_el.jsp** que nous avons mis en place pour les précédents tests, et remplacez son contenu par ce code :

Code : JSP - /test_el.jsp

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Test des expressions EL</title>
    </head>
    <body>
    <p>
        <!-- Initialisation d'un bean de type Coyote avec une
action standard, pour l'exemple : -->
        <jsp:useBean id="coyote" class="com.sdzee.beans.Coyote" />
        <!-- Initialisation de sa propriété 'prénom' : -->
        <jsp:setProperty name="coyote" property="prenom" value="Wile
E."/>
        <!-- Et affichage de sa valeur : -->
        <jsp:getProperty name="coyote" property="prenom" />
    </p>
    </body>
</html>
```

Avec les deux premières actions, la base de notre exemple est posée : nous avons créé un bean de type **Coyote** dans notre page JSP, et avons initialisé sa propriété **prenom** avec la valeur "Wile E.".

Vous retrouvez ensuite à la ligne 14 l'action qui affiche le contenu de cette propriété, et lorsque vous vous rendez sur http://localhost:8080/test/test_el.jsp, le résultat affiché par le navigateur est logiquement "Wile E.".

Maintenant, remplacez cette ligne 14 par la ligne suivante :

Code : JSP

```
${ coyote.prenom }
```

Actualisez alors la page de tests dans votre navigateur, et vous observerez que le contenu n'a pas changé, la valeur de la propriété prénom est toujours correctement affichée. Eh bien oui, c'est tout ce qu'il est nécessaire d'écrire avec la technologie EL ! Cette expression retourne le contenu de la propriété **prenom** du bean nommé **coyote**. Remarquez bien la syntaxe et la convention employées :

- **coyote** est le nom du bean, que nous avions ici défini avec l'attribut **id** de l'action **<jsp:useBean/>** ;
- **prenom** est un champ privé du bean (une propriété) accessible par sa méthode publique `getPrenom()` ;
- l'opérateur *point* permet de séparer le bean visé de sa propriété.

Pour information, mais nous y reviendrons un peu plus tard, voici ce à quoi ressemble le code Java qui est mis en œuvre dans les coulisses lors de l'interprétation de l'expression `${ coyote.prenom }` :

Code : Java

```
Coyote bean = (Coyote) pageContext.findAttribute( "coyote" );
if ( bean != null ) {
    String prenom = bean.getPrenom();
    if ( prenom != null ) {
        out.print( prenom );
    }
}
```

Peu importe que nous comparions avec une scriptlet Java ou avec une action standard, en utilisant l'expression EL, **la simplification d'écriture est radicale** ! Ci-dessous, étudiez ces exemples des erreurs et utilisations les plus courantes :

Code : JSP

```
<!-- Syntaxe conseillée pour récupérer la propriété 'nom' du bean
'coyote'. -->
${ coyote.nom }

<!-- Syntaxe correcte, il est possible d'expliciter la méthode
d'accès à l'attribut. Préférez toutefois la notation précédente. --
>
${ coyote.getNom() }

<!-- Syntaxe erronée : la première lettre de la propriété doit être
une minuscule. -->
${ coyote.Nom }

<!-- Sachez enfin que l'on peut utiliser cette syntaxe au sein de
tests : -->
${ coyote.nom == "Jean-Paul" } <!-- comparaison d'égalité avec une
chaine -->
${ empty coyote.nom } <!-- vérification si vide ou null -->
${ !empty coyote.nom ? coyote.nom : "Veuillez préciser un nom" } <!-
- condition ternaire qui affiche le nom s'il est renseigné, et un
message sinon -->
```

En outre, sachez également que **les expressions EL sont protégées contre un éventuel retour `null`** :

Code : JSP

```
<!-- La scriptlet suivante affiche "null" si la propriété "nom" n'a
pas été initialisée,
```

```
        et provoque une erreur à la compilation si l'objet "coyote" n'a pas
        été initialisé : -->
        <%= coyote.getNom() %>

        <!-- L'action suivante affiche "null" si la propriété "nom" n'a pas
        été initialisée,
        et provoque une erreur à l'exécution si l'objet "coyote" n'a pas
        été initialisé : -->
        <jsp:getProperty name="coyote" property="nom" />

        <!-- L'expression EL suivante n'affiche rien si la propriété "nom"
        n'a pas été initialisée,
        et n'affiche rien si l'objet "coyote" n'a pas été initialisé : -->
        ${ coyote.nom }
```

Pour terminer, sachez enfin que la valeur retournée par une EL positionnée dans un texte ou un contenu statique sera insérée à l'endroit même où est située l'expression :

Code : JSP

```
        <!-- La ligne suivante : -->
        <p>12 est inférieur à 8 : ${ 12 lt 8 }.</p>

        <!-- Sera rendue ainsi après interprétation de l'expression, 12
        n'étant pas inférieur à 8 : -->
        <p>12 est inférieur à 8 : false.</p>
```

Désactiver l'évaluation des expressions EL

Le format utilisé par le langage EL, à savoir `${ ... }`, n'était pas défini dans les premières versions de la technologie JSP. Ainsi, si vous travaillez sur de vieilles applications non mises à jour, il est possible que vous soyez amenés à empêcher de telles expressions d'être évaluées, afin d'assurer la rétro-compatibilité avec le code. C'est pour cela qu'il est possible de désactiver l'évaluation des expressions EL :

- au cas par cas, grâce à la directive **page** que nous avons brièvement découverte dans le chapitre précédent ;
- sur tout ou partie des pages, grâce à une section à ajouter dans le fichier web.xml.

Avec la directive page

La directive suivante désactive l'évaluation des EL dans une page JSP :

Code : JSP - Directive à placer en tête d'une page JSP

```
        <%@ page isELIgnored ="true" %>
```

Les seules valeurs acceptées par l'attribut **isELIgnored** sont **`true`** et **`false`** :

- s'il est initialisé à **`true`**, alors les expressions EL seront ignorées et apparaîtront en tant que simples chaînes de caractères ;
- s'il est initialisé à **`false`**, alors elles seront interprétées par le conteneur.

Vous pouvez d'ailleurs faire le test dans votre page **test_el.jsp**, puisqu'elle contient l'expression `${ coyote.prenom }`. Éditez le fichier et insérez-y en première ligne la directive. Actualisez-alors l'affichage de la page dans votre navigateur, et vous observerez que l'expression n'est plus évaluée : alors que le contenu de la propriété **nom** vous était affiché auparavant, cette fois l'expression vous est ré-affichée telle quelle, elle n'est plus interprétée.

Avec le fichier web.xml

Vous pouvez désactiver l'évaluation des expressions EL sur tout un ensemble de pages JSP dans une application grâce à l'option **<el-ignored>** du fichier web.xml. Elle se présente dans une section de cette forme :

Code : XML - Section à ajouter dans le fichier web.xml

```
<jsp-config>
    <jsp-property-group>
        <url-pattern>*.jsp</url-pattern>
        <el-ignored>true</el-ignored>
    </jsp-property-group>
</jsp-config>
```

Vous connaissez déjà le principe du champ **<url-pattern>**, que nous avons découvert lorsque nous avons mis en place notre première servlet. Pour rappel donc, il permet de définir sur quelles pages appliquer ce comportement. En l'occurrence, la notation *.jsp ici utilisée signifie que toutes les pages JSP de l'application seront impactées par cette configuration.

Le champ **<el-ignored>** permet logiquement de définir la valeur de l'option. Le comportement est bien entendu le même qu'avec la directive précédente : si **true** est précisé alors les EL seront ignorées, et si **false** est précisé elles seront interprétées.

Comportement par défaut

Si vous ne mettez pas de configuration spécifique en place, comme celles que nous venons à l'instant de découvrir, la valeur de l'option **isELIgnored** va dépendre de la version du conteneur utilisé par votre application :

- si la version du conteneur de servlets est supérieure ou égale à 2.4, alors les expressions EL seront **évaluées par défaut** ;
- si la version est inférieure à 2.4, alors il est possible que les expressions EL soient **ignorées par défaut**, pour assurer la rétro-compatibilité dont je vous ai déjà parlé.

Comment connaître la version du conteneur utilisé par une application ?

Cela se passe au niveau de la balise **<web-app>** du fichier **web.xml**. Dans notre projet, nous n'avons rien spécifié à ce sujet, nous nous sommes contentés d'écrire :

Code : XML - /WEB-INF/web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
...
</web-app>
```

Ceci implique donc que notre application va se baser par défaut sur la version du conteneur définie par le serveur d'application, en l'occurrence Tomcat 7. Lorsque je vous ai présenté Tomcat pour la première fois, je vous ai signalé l'existence d'un dossier **conf** placé sous son répertoire d'installation, et vous ai précisé qu'il contenait un fichier **web.xml** renfermant des paramètres de configuration communs à toutes les applications web déployées sur le serveur. Eh bien si vous allez jeter un oeil au contenu de ce fichier, vous y trouverez la ligne suivante :

Code : XML - .../apache-tomcat-7.0.XX/conf/web.xml

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
  version="3.0">
```

C'est en spécifiant ces quelques attributs dans la balise **<web-app>** que Tomcat 7 définit la version du conteneur de servlets utilisée par défaut par ses applications. En l'occurrence, nous constatons qu'il s'agit du conteneur en version 3.0, et pouvons donc en déduire que les expressions EL seront interprétées par défaut dans nos applications.

Si nous changeons de serveur d'applications, que va-t-il se passer ?

Eh oui, si jamais vous déployez votre application sur un autre serveur que Tomcat 7, il est tout à fait envisageable que la version par défaut du conteneur utilisé soit différente, et implique des changements non désirés dans le comportement de votre application. Pour rendre votre application indépendante du serveur sur laquelle elle est déployée, vous pouvez préciser la définition de la balise **<web-app>** dans son propre fichier **web.xml**. Voici par exemple comment forcer l'utilisation du conteneur en version 2.5, peu importe le serveur d'applications utilisé :

Code : XML - /WEB-INF/web.xml

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  version="2.5">
```

Cette introduction à la technologie EL se termine là. Il y en a bien assez à dire sur le sujet pour écrire un tutoriel entier, mais ce que nous avons appris ici nous suffira dans la plupart des cas. Retenez bien que ces expressions vont vous permettre de ne plus faire intervenir de Java dans vos JSP, et puisque vous savez maintenant de quoi il retourne, vous ne serez pas surpris de retrouver la syntaxe ${...} dans tous les futurs exemples de ce cours.

Les objets implicites

Il nous reste un concept important à aborder avant de passer à la pratique : **les objets implicites**. Il en existe deux types :

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

Les objets de la technologie JSP

Pour illustrer ce nouveau concept, revenons sur la première JSP que nous avions écrite dans le chapitre sur la transmission des données :

Code : JSP - /WEB-INF/test.jsp

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Test</title>
    </head>
    <body>
        <p>Ceci est une page générée depuis une JSP.</p>
        <p>
            <%
            String attribut = (String) request.getAttribute("test");
out.println( attribut );
            %>
        </p>
    </body>
</html>
```

Je vous avais alors fait remarquer que dans la ligne de code ici surlignée, nous avions directement utilisé l'objet **out** sans jamais l'avoir instancié auparavant. De même, à la ligne 10 nous accédons directement à la méthode `request.getAttribute()` sans jamais avoir instancié d'objet nommé **request**...

Comment est-ce possible ?

Pour répondre à cette question, nous devons nous intéresser une nouvelle fois au code de la servlet auto-générée par Tomcat, comme nous l'avions fait dans le second chapitre de cette partie. Retournons donc dans le répertoire **work** du serveur, qui rappelez-vous est subtilisé par Eclipse, et analysons à nouveau le code du fichier **test_jsp.java** :

Code : Java - Extrait de la servlet auto-générée par Tomcat

```
...

  public void _jspService(final
javax.servlet.http.HttpServletRequest request, final
javax.servlet.http.HttpServletResponse response)
        throws java.io.IOException, javax.servlet.ServletException {

    final javax.servlet.jsp.PageContext pageContext;
    javax.servlet.http.HttpSession session = null;
    final javax.servlet.ServletContext application;
    final javax.servlet.ServletConfig config;
javax.servlet.jsp.JspWriter out = null;
    final java.lang.Object page = this;
    javax.servlet.jsp.JspWriter _jspx_out = null;
    javax.servlet.jsp.PageContext _jspx_page_context = null;


    try {
      response.setContentType("text/html");
      pageContext = _jspxFactory.getPageContext(this, request,
response,
         null, true, 8192, true);
      _jspx_page_context = pageContext;
      application = pageContext.getServletContext();
      config = pageContext.getServletConfig();
      session = pageContext.getSession();
out = pageContext.getOut();
      _jspx_out = out;

      out.write("<!DOCTYPE html>\r\n");
      out.write("<html>\r\n");
      out.write(" <head>\r\n");
      out.write(" <meta charset=\"utf-8\" />\r\n");
      out.write(" <title>Test</title>\r\n");
      out.write(" </head>\r\n");
      out.write(" <body>\r\n");
      out.write(" <p>Ceci est une page générée depuis une
JSP.</p>\r\n");
      out.write(" <p>\r\n");
      out.write(" ");

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

      out.write("\r\n");
      out.write(" </p>\r\n");
      out.write(" </body>\r\n");
      out.write("</html>");
    }
    ...
```

Analysons ce qu'il se passe dans le cas de l'objet **out** :

- à la ligne 10, un objet nommé **out** et de type `JspWriter` est créé ;
- à la ligne 24, il est initialisé avec l'objet *writer* récupéré depuis la réponse ;
- et la ligne 39, c'est tout simplement notre ligne de code Java, basée sur l'objet **out**, qui est recopiée telle quelle de la JSP

vers la servlet auto-générée !

Pour l'objet **request**, c'est un peu différent. Comme je vous l'ai déjà expliqué dans le second chapitre, notre JSP est ici transformée en servlet. Si elle en diffère par certains aspects, sa structure globale ressemble toutefois beaucoup à celle de la servlet que nous avons créée et manipulée dans nos exemples jusqu'à présent. Regardez la ligne 3 : le traitement de la paire requête / réponse est contenu dans une méthode qui prend pour arguments les objets `HttpServletRequest` et `HttpServletResponse`, exactement comme le fait notre méthode `doGet()` ! Voilà pourquoi il est possible d'utiliser directement les objets **request** et **response** depuis une JSP.

Vous devez maintenant comprendre pourquoi vous n'avez pas besoin d'instancier ou de récupérer les objets **out** et **request** avant de les utiliser dans le code de votre JSP : dans les coulisses, **le conteneur s'en charge pour vous lorsqu'il traduit votre page en servlet** ! Et c'est pour cette raison que ces objets sont dits "implicites" : vous n'avez pas besoin de les déclarer de manière... explicite. Logique, non ?

Par ailleurs, si vous regardez attentivement le code ci-dessus, vous constaterez que les lignes 6 à 13 correspondent en réalité toutes à des initialisations d'objets : **pageContext**, **session**, **application**... En fin de compte, le conteneur met à votre disposition toute une série d'objets implicites, tous accessibles directement depuis vos pages JSP. En voici la liste :

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

De la même manière que nous avons utilisé les objets **request** et **out** dans notre exemple précédent, il est possible d'utiliser n'importe lequel de ces neuf objets à travers le code Java que nous écrivons dans nos pages JSP...

Hein ?! Encore du code Java dans nos pages JSP ?

Eh oui, tout cela est bien aimable de la part de notre cher conteneur, mais des objets sous cette forme ne vont pas nous servir à grand chose ! Souvenez-vous, nous avons pour objectif de ne plus écrire de code Java directement dans nos pages.

Les objets de la technologie EL

J'en vois déjà quelques uns au fond qui sortent les cordes... Vous avez à peine digéré les objets implicites de la technologie JSP, je vous annonce maintenant qu'il en existe d'autres rendus disponibles par les expressions EL ! Pas de panique, reprenons tout cela calmement. En réalité, et heureusement pour nous, la technologie EL va apporter une solution élégante au problème que nous venons de soulever : **nous allons grâce à elle pouvoir profiter des objets implicites sans écrire de code Java** !

Dans les coulisses, le concept est sensiblement le même que pour les objets implicites JSP : il s'agit d'objets gérés automatiquement par le conteneur lors de l'évaluation des expressions EL, et auxquels nous pouvons directement accéder depuis nos expressions sans les déclarer auparavant. Voici un tableau des différents objets implicites mis à disposition par la technologie EL :

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

La première chose à remarquer dans ce tableau, c'est que le seul objet implicite en commun entre les JSP et les expressions EL est le **pageContext**. Je ne m'attarde pas plus longtemps sur cet aspect, nous allons y revenir dans le chapitre suivant.

La seconde, c'est la différence flagrante avec les objets implicites JSP : tous les autres objets implicites de la technologie EL sont des `Map` !

D'ailleurs, qu'est-ce que c'est que toute cette histoire de `Map` et d'associations entre des noms et des valeurs ?

Ça parait compliqué, mais en réalité c'est très simple. Si vous ne vous souvenez pas bien des collections Java, sachez qu'une `Map` est un objet qui peut se représenter comme un tableau à deux colonnes :

- la première colonne contient ce que l'on nomme les **clés**, qui doivent obligatoirement être uniques ;
- la seconde contient les valeurs, qui peuvent quant à elles être associées à plusieurs clés.

Chaque ligne du tableau ne peut contenir qu'une clé et une valeur. Voici un exemple d'une `Map<String, String>` représentant une liste d'aliments et leurs types :

Aliments (Clés)	Types (Valeurs)

pomme	fruit
carotte	légume
boeuf	viande
aubergine	légume
...	...

Vous voyez bien ici qu'un même type peut être associé à différents aliments, mais qu'un même aliment ne peut exister qu'une seule fois dans la liste. Eh bien c'est ça le principe d'une `Map` : c'est un ensemble d'éléments uniques auxquels on peut associer n'importe quelle valeur.

Quel est le rapport avec la technologie EL ?

Le rapport, c'est que nos expressions EL sont capables d'accéder au contenu d'une `Map`, de la même manière qu'elles sont capables d'accéder aux propriétés d'un bean ! Continuons notre exemple avec la liste d'aliments, et créons une page **test_map.jsp**, dans laquelle nous allons implémenter rapidement la `Map` d'aliments :

Code : JSP - /test_map.jsp

```
<%@ page import="java.util.Map, java.util.HashMap" %>
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Test des Maps et EL</title>
    </head>
    <body>
    <p>
       <%
         Map<String, String> aliments = new HashMap<String,
String>();
         aliments.put( "pomme","fruit" );
         aliments.put( "carotte","légume" );
         aliments.put( "boeuf","viande" );
         aliments.put( "aubergine","légume" );
         request.setAttribute( "aliments", aliments );
       %>
${ aliments.pomme } <br /> <!-- affiche fruit -->
${ aliments.carotte } <br /> <!-- affiche légume -->
${ aliments.boeuf } <br /> <!-- affiche viande -->
${ aliments.aubergine } <br /><!-- affiche légume -->
    </p>
    </body>
</html>
```

J'utilise ici une scriptlet Java pour initialiser rapidement la `Map` et la placer dans un attribut de la requête nommé **aliments**. Ne prenez bien évidemment pas cette habitude, je ne procède ainsi que pour l'exemple et vous rappelle que nous cherchons à éliminer le code Java de nos pages JSP ! Rendez-vous alors sur http://localhost:8080/test/test_map.jsp, et observez le bon affichage des valeurs. Les expressions utilisées dans les lignes ici surlignées fonctionnent donc en apparence de la même manière qu'avec des beans.

Ok, avec des expressions EL, nous pouvons accéder au contenu d'objets de type `Map`. Mais quel est le rapport avec les objets implicites EL ?

Pour illustrer le principe, nous allons laisser tomber nos fruits et légumes et créer une page nommée **test_obj_impl.jsp**, encore et toujours à la racine de notre application, et y placer le code suivant :

Code : JSP - /test_obj_impl.jsp

```
<!DOCTYPE html>
```

```
<html>
    <head>
        <meta charset="utf-8" />
        <title>Test des objets implicites EL</title>
    </head>
    <body>
    <p>
 <%
String paramLangue = request.getParameter("langue");
 out.println( "Langue : " + paramLangue );
 %>
 <br />
 <%
String paramArticle = request.getParameter("article");
 out.println( "Article : " + paramArticle );
 %>
    </p>
    </body>
</html>
```

Vous reconnaissez aux lignes 10 et 15 la méthode `request.getParameter()` permettant de récupérer les paramètres transmis au serveur par le client à travers l'URL. Ainsi, il vous suffit de vous rendre sur http://localhost:8080/test/test_obj_im [...] r&article=782 pour que votre navigateur vous affiche :

Cherchez maintenant, dans le tableau précédent, l'objet implicite EL dédié à l'accès aux paramètres de requête... Trouvé ? Il s'agit de la `Map` nommée **param**. La technologie EL va ainsi vous mettre à disposition un objet dont le contenu peut dans le cas de notre exemple être représenté sous cette forme :

Nom du paramètre (Clé)	Valeur du paramètre (Valeur)
langue	fr
article	782

Si vous avez compris l'exemple avec les fruits et légumes, alors vous avez également compris comment accéder à nos paramètres de requêtes depuis des expressions EL, et vous êtes capables de réécrire notre précédente page d'exemple sans utiliser de code Java ! Éditez votre fichier **test_obj_impl.jsp** et remplacez le code précédent par le suivant :

Code : JSP - /test_obj_impl.jsp

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Test des objets implicites EL</title>
    </head>
    <body>
    <p>
Langue : ${ param.langue }
 <br />
Article : ${ param.article }
    </p>
    </body>
</html>
```

Actualisez la page dans votre navigateur, et observez le même affichage que dans l'exemple précédent.
Pratique et élégant, n'est-ce pas ?

D'accord, dans ce cas cela fonctionne bien : chaque paramètre a un nom unique, et est associé à une seule valeur quelconque. Mais qu'en est-il des lignes marquées avec ** dans le tableau ? Est-il possible d'associer un unique paramètre à plusieurs valeurs à la fois ?

Oui, il est tout à fait possible d'associer une clé à des valeurs multiples. C'est d'ailleurs tout à fait logique, puisque derrière les rideaux il s'agit tout simplement d'objets de type `Map` ! L'unique différence entre les objets implicites **param** et **paramValues**, ainsi qu'entre **header** et **headerValues**, se situe au niveau de la nature de l'objet utilisé dans la `Map` et des valeurs qui y sont stockées :

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

Quand pouvons-nous rencontrer plusieurs valeurs pour un seul et même paramètre ?

Prenons un exemple HTML très simple : un **<select>** à choix multiples !

Code : HTML

```
<form method="post" action="">
   <p>
       <label for="pays">Dans quel(s) pays avez-vous déjà voyagé
?</label><br />
       <select name="pays" id="pays" multiple="true">
           <option value="france">France</option>
           <option value="espagne">Espagne</option>
           <option value="italie">Italie</option>
           <option value="royaume-uni">Royaume-Uni</option>
           <option value="canada">Canada</option>
           <option value="etats-unis">Etats-Unis</option>
           <option value="chine" selected="selected">Chine</option>
           <option value="japon">Japon</option>
       </select>
   </p>
</form>
```

Alors que via un **<select>** classique, il n'est possible de choisir qu'une seule valeur dans la liste déroulante, dans cet exemple grâce à l'option **multiple**="true", il est tout à fait possible de sélectionner plusieurs valeurs pour le seul paramètre nommé **pays**. Eh bien c'est dans ce genre de cas que l'utilisation de l'objet implicite **paramValues** est nécessaire : c'est le seul moyen de récupérer la liste des valeurs associées au seul paramètre nommé **pays** !

Pour ce qui est de l'objet implicite **headerValues** par contre, sa réelle utilité est discutable. En effet, s'il est possible de définir plusieurs valeurs à un seul paramètre d'une en-tête HTTP, celles-ci sont la plupart du temps séparées par de simples point-virgules et concaténées dans une seul et même `String`, rendant l'emploi de cet objet implicite inutile. Bref, dans 99% des cas, utiliser la simple Map **header** est suffisant. Ci-dessous un exemple d'en-têtes HTTP :

Code : HTTP

```
GET / HTTP/1.1
Host: www.google.fr
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-
US; rv:1.9.2.6) Gecko/20100625 Firefox/3.6.6 ( .NET CLR 3.5.30729)
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 115
Connection: keep-alive
```

Vous remarquez bien dans cet exemple que chaque paramètre (**Host**, **User-Agent**, **Accept**, etc.) n'est défini qu'une seule fois, et que les valeurs sont simplement concaténées les unes à la suite des autres sur la même ligne.

Voilà donc la solution à notre problème : les objets implicites EL sont des raccourcis qui rendent l'accès aux différentes portées et aux différents concepts liés à HTTP extrêmement pratiques !

Nous allons nous arrêter là pour les explications sur les objets implicites. Ne vous inquiétez pas si vous ne saisissez pas encore bien leur intérêt de chacun d'entre eux, certains concepts vous sont encore inconnus. La pratique nous fera prendre de l'aisance, et j'apporterai de plus fines explications au cas par cas dans les exemples de ce cours. Avant de passer à la suite, un petit avertissement quant aux nommage de vos objets :

Faites bien attention aux noms des objets implicites listés ci-dessus. Il est fortement déconseillé de déclarer une variable portant le même nom qu'un objet implicite, par exemple **param** ou **cookie**. En effet, ces noms sont déjà utilisés pour identifier des objets implicites, et cela pourrait causer des comportements plutôt inattendus dans vos pages et expressions EL. **Bref, ne cherchez pas les ennuis : ne donnez pas à vos variables un nom déjà utilisé par un objet implicite.**

Beaucoup de nouvelles notations vous ont été présentées, prenez le temps de bien comprendre les exemples illustrant l'utilisation des balises et expressions. Lorsque vous vous sentez prêts, passez avec moi au chapitre suivant, et tentez alors de réécrire notre précédente page d'exemple JSP, en y faisant cette fois intervenir uniquement ce que nous venons d'apprendre !

Des problèmes de vue ?

Passons sur ce subtil jeu de mot, et revenons un instant sur notre premier exemple de page dynamique. Maintenant que nous connaissons la technologie JSP et les EL, nous sommes capables de remplacer le code Java que nous avions écrit en dur dans notre vue par quelque chose de propre, lisible et qui suit les recommandations MVC !

Nettoyons notre exemple

Pour rappel, voici où nous en étions après l'introduction d'un bean dans notre exemple :

Code : JSP - /WEB-INF/test.jsp

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Test</title>
    </head>
    <body>
        <p>Ceci est une page générée depuis une JSP.</p>
        <p>
            <%
            String attribut = (String) request.getAttribute("test");
            out.println( attribut );

            String parametre = request.getParameter( "auteur" );
            out.println( parametre );
            %>
        </p>
        <p>
            Récupération du bean :
            <%
     com.sdzee.beans.Coyote notreBean = (com.sdzee.beans.Coyote)
request.getAttribute("coyote");
     out.println( notreBean.getPrenom() );
            out.println( notreBean.getNom() );
            %>
        </p>
    </body>
</html>
```

Avec tout ce que nous avons appris, nous sommes maintenant capables de modifier cette page JSP pour qu'elle ne contienne plus de langage Java !
Pour bien couvrir l'ensemble des méthodes existantes, divisons le travail en deux étapes : avec des scripts et balises JSP pour commencer, puis avec des EL.

Avec des scripts et balises JSP

Code : JSP - /WEB-INF/test.jsp

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Test</title>
    </head>
    <body>
        <p>Ceci est une page générée depuis une JSP.</p>
        <p>
            <% String attribut = (String)
request.getAttribute("test"); %>
<%= attribut %>

            <% String parametre = request.getParameter( "auteur" );
%>
```

```
%>
<%= parametre %>
        </p>
        <p>
            Récupération du bean :
<jsp:useBean id="coyote" class="com.sdzee.beans.Coyote"
scope="request" />
<jsp:getProperty name="coyote" property="prenom" />
<jsp:getProperty name="coyote" property="nom" />
        </p>
    </body>
</html>
```

Vous pouvez remarquer :

- l'affichage de l'attribut et du paramètre via la balise d'expression **<%= ... %>**;
- la récupération du bean depuis la requête via la balise **<jsp:useBean>** ;
- l'affichage du contenu des propriétés via les balises **<jsp:getProperty>**.

Quel est l'objet implicite utilisé ici pour récupérer le bean "coyote" ?

Lorsque vous utilisez l'action :

Code : JSP

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

Celle-ci s'appuie derrière les rideaux sur l'objet implicite **request** (`HttpServletRequest`) : elle cherche un bean nommé "coyote" dans la requête, et si elle n'en trouve pas elle en crée un et l'y enregistre. De même, si vous aviez précisé "session" ou "application" dans l'attribut **scope** de l'action, alors elle aurait cherché respectivement dans les objets **session** (`HttpSession`) et **application** (`ServletContext`).

Enfin, lorsque vous ne précisez pas d'attribut **scope** :

Code : JSP

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

Alors l'action s'appuie par défaut sur l'objet implicite **page** (**this**) : elle cherche un bean nommé "coyote" dans la page courante, et si elle n'en trouve pas elle en crée un et l'y enregistre.

En fin de compte notre exemple est déjà bien plus propre qu'avant, mais nous avons toujours besoin de faire appel à du code Java pour récupérer et afficher nos attribut et paramètre depuis la requête...

Avec des EL

Code : JSP - /WEB-INF/test.jsp

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Test</title>
    </head>
    <body>
        <p>Ceci est une page générée depuis une JSP.</p>
        <p>
```

```
        ${test}
        ${auteur}
                </p>
                <p>
                    Récupération du bean :
        ${coyote.prenom}
        ${coyote.nom}
                </p>
            </body>
        </html>
```

Ça se passe de commentaires... Vous comprenez maintenant pourquoi je vous ai annoncé dans le chapitre précédent qu'une fois les EL découvertes, vous n'utiliserez plus jamais le reste ?
Leur simplicité d'utilisation est déconcertante, et notre page ne contient désormais plus une seule ligne de Java !

Quel sont les objets implicites utilisés ici pour récupérer l'attribut "test", le paramètre "auteur" et le bean "coyote" ?

La question mérite d'être posée, car le fonctionnement est différent de celui des actions standards : avec une EL, vous n'avez pas besoin de spécifier l'objet implicite (c'est-à-dire la portée) auquel vous souhaitez accéder ! En effet, vous voyez bien dans notre exemple que nous n'avons pas écrit `${request.test}` pour accéder à l'objet **test** présent dans l'objet implicite **request**, ni `${request.coyote.prenom}` pour accéder au bean **coyote** présent lui aussi dans l'objet implicite **request**.

D'ailleurs, si vous aviez fait ainsi... ça n'aurait pas fonctionné ! En réalité, le mécanisme des EL est un peu évolué : une EL réalise d'elle-même un parcours automatique des différentes portées accessibles à la recherche d'un objet portant le nom précisé, de la plus petite à la plus grande portée.

Comment ce mécanisme fonctionne-t-il ?

Vous vous souvenez de l'objet implicite **pageContext** ? Je vous l'avais présenté comme celui qui donne accès à toutes les portées...

Eh bien concrètement, lorsque vous écrivez par exemple l'expression `${test}` dans votre JSP, derrière les rideaux le conteneur va appeler la méthode `findAttribute()` de l'objet `PageContext` ! Et cette méthode va à son tour parcourir chacune des portées - page, puis request, puis session et enfin application - et va retourner le premier objet nommé "test" trouvé !

En clair, vous devez éviter de donner le même nom à des objets différents ! Par exemple, si vous enregistrez un objet nommé "test" en session, et un autre objet nommé "test" en requête, puisque la portée request est parcoure avant la portée session, c'est toujours l'objet enregistré dans la requête qui va être renvoyé lorsque vous écrirez `${test}`. Même si plus tard, nous allons découvrir un moyen de cibler la portée désirée, la bonne pratique en vigueur dans la conception d'une application est d'éviter les noms d'objets identiques !

Complétons notre exemple...

Tout cela se goupille plutôt bien pour le moment, oui mais voilà... il y a un "mais". Et un gros même !
Vous ne vous en êtes peut-être pas encore aperçus mais les EL, mêmes couplées à des balises JSP, ne permettent pas de mettre en place tout ce dont nous aurons couramment besoin dans une vue.

Prenons deux exemples pourtant très simples :

- nous souhaitons afficher le contenu d'une liste ou d'un tableau à l'aide d'une boucle ;
- nous souhaitons afficher un texte différent selon que le jour du mois est pair ou impair.

Eh bien ça, nous ne savons pas encore le faire sans Java !

Manipulation d'une liste

Reprenons notre exemple, en créant une liste depuis une servlet et en essayant d'afficher son contenu depuis la JSP :

Code : Java - Ajout d'une liste depuis la servlet

```
...

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

 /* Création et initialisation du message. */
 String paramAuteur = request.getParameter( "auteur" );
 String message = "Transmission de variables : OK ! " + paramAuteur;

 /* Création du bean et initialisation de ses propriétés */
 Coyote premierBean = new Coyote();
 premierBean.setNom( "Coyote" );
 premierBean.setPrenom( "Wile E." );

 /* Création de la liste et insertion de quatre éléments */
List<Integer> premiereListe = new ArrayList<Integer>();
premiereListe.add( 27 );
premiereListe.add( 12 );
premiereListe.add( 138 );
premiereListe.add( 6 );

 /* Stockage du message, du bean et de la liste dans l'objet
request */
 request.setAttribute( "test", message );
 request.setAttribute( "coyote", premierBean );
request.setAttribute( "liste", premiereListe );

 /* Transmission de la paire d'objets request/response à notre JSP
*/
 this.getServletContext().getRequestDispatcher( "/WEB-INF/test.jsp"
).forward( request, response );
}
```

Rien de problématique ici, c'est encore le même principe : nous initialisons notre objet, et le stockons dans l'objet requête pour transmission à la JSP. Regardons maintenant comment réaliser l'affichage des éléments de cette liste :

Code : JSP - Récupération et affichage du contenu de la liste depuis la JSP

```
<%@ page import="java.util.List" %>
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Test</title>
    </head>
    <body>
        <p>Ceci est une page générée depuis une JSP.</p>
        <p>
            ${test}
            ${auteur}
        </p>
        <p>
            Récupération du bean :
            ${coyote.prenom}
            ${coyote.nom}
        </p>
        <p>
            Récupération de la liste :
<%
List<Integer> liste = (List<Integer>) request.getAttribute( "liste"
);
for( Integer i : liste ){
out.println(i + " : ");
}
%>
        </p>
    </body>
```

```
</html>
```

Voilà à quoi nous en sommes réduits : réaliser un import avec la directive page, pour pouvoir utiliser ensuite le type `List` lors de la récupération de notre liste depuis l'objet requête, et afficher son contenu via une boucle for. Certes, ce code fonctionne, vous pouvez regarder le résultat obtenu depuis votre navigateur. Mais nous savons d'ores et déjà que cela va à l'encontre du modèle MVC : souvenez-vous, le Java dans une page JSP, c'est mal !

Utilisation d'une condition

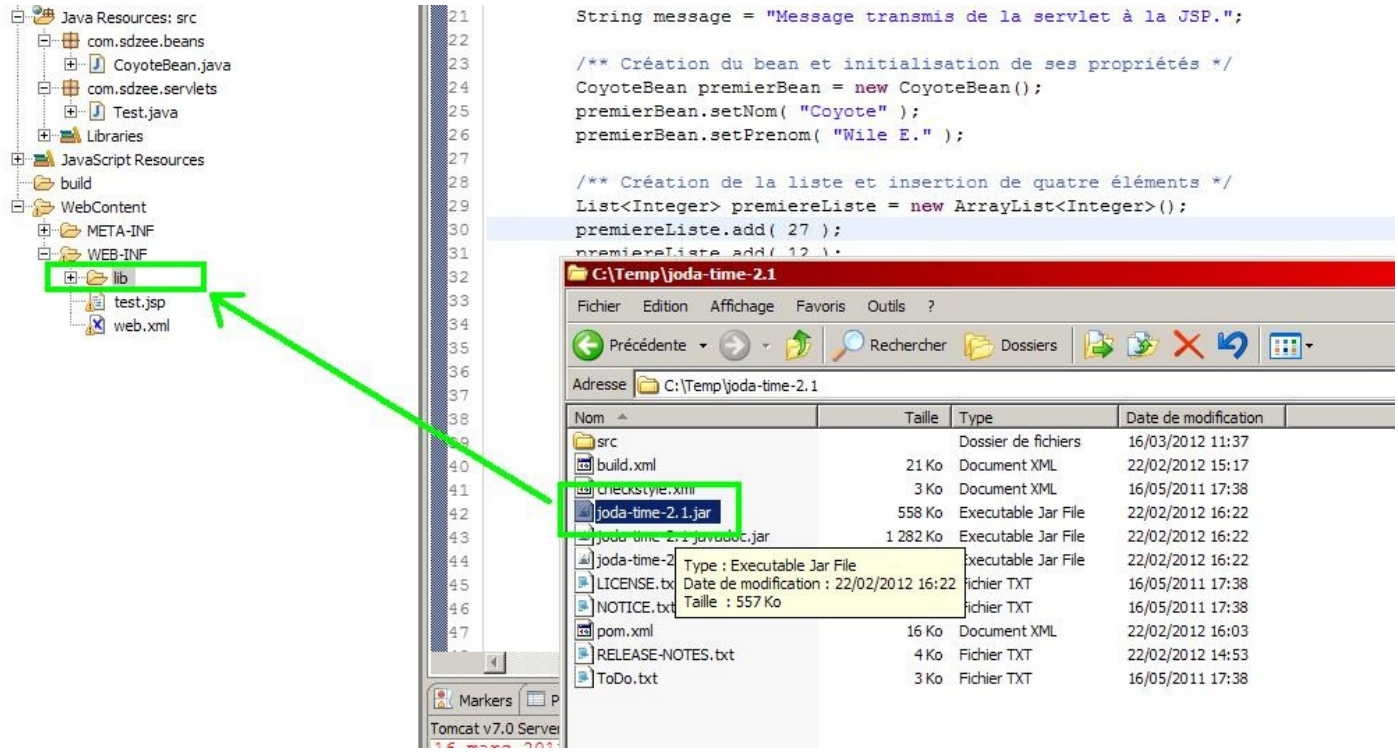
Voyons maintenant comment réaliser notre second exemple.

Puisque la mise en place d'une condition n'a vraiment rien de passionnant (vous savez tous écrire un if !), je profite de ce petit exemple pour vous faire découvrir une API très utile dès lors que votre projet fait intervenir la manipulation de dates : JodaTime.

Si vous avez déjà programmé en Java, vous avez très certainement déjà remarqué ce problème : **la manipulation de dates en Java est horriblement peu intuitive** ! Que ce soit via l'objet `Date` ou via l'objet `Calendar`, c'est très décevant et très loin de ce que l'on est en droit d'attendre d'une plate-forme évoluée comme Java !

Afin de pouvoir utiliser les méthodes et objets de cette API, il vous faut :

1. télécharger l'archive nommée **joda-time-2.1-dist** disponible sur cette page, par exemple au format zip ;
2. la décompresser et y chercher le fichier **joda-time-2.1.jar** ;
3. l'inclure à votre application, en le plaçant sous le répertoire **/WEB-INF/lib/** de votre projet (un simple glisser-déposer depuis votre fichier vers Eclipse suffit, voir copie d'écran ci-dessous).



Mise en place de l'API JodaTime dans un projet Eclipse

Une fois le fichier .jar en place, vous pouvez alors utiliser l'API depuis votre projet.

Code : Java - Récupération du jour du mois depuis la servlet

```
...

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

```
        /* Création et initialisation du message. */
        String paramAuteur = request.getParameter( "auteur" );
        String message = "Transmission de variables : OK ! " + paramAuteur;

        /* Création du bean et initialisation de ses propriétés */
        Coyote premierBean = new Coyote();
        premierBean.setNom( "Coyote" );
        premierBean.setPrenom( "Wile E." );

        /* Création de la liste et insertion de quatre éléments */
        List<Integer> premiereListe = new ArrayList<Integer>();
        premiereListe.add( 27 );
        premiereListe.add( 12 );
        premiereListe.add( 138 );
        premiereListe.add( 6 );

        /** On utilise ici la libraire Joda pour manipuler les dates, pour
    deux raisons :
    * - c'est tellement plus simple et limpide que de travailler avec
    les objets Date ou Calendar !
    * - c'est (probablement) un futur standard de l'API Java.
    */
    DateTime dt = new DateTime();
    Integer jourDuMois = dt.getDayOfMonth();

        /* Stockage du message, du bean, de la liste et du jour du mois
    dans l'objet request */
        request.setAttribute( "test", message );
        request.setAttribute( "coyote", premierBean );
        request.setAttribute( "liste", premiereListe );
    request.setAttribute( "jour", jourDuMois );

        /* Transmission de la paire d'objets request/response à notre JSP
    */
        this.getServletContext().getRequestDispatcher( "/WEB-INF/test.jsp"
    ).forward( request, response );
    }
```

Remarquez la facilité d'utilisation de l'API Joda. N'hésitez pas à parcourir par vous-mêmes les autres objets et méthodes proposées, c'est d'une simplicité impressionnante.

Code : JSP - Récupération du jour du mois et affichage d'un message dépendant de sa parité

```
<%@ page import="java.util.List" %>
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Test</title>
    </head>
    <body>
        <p>Ceci est une page générée depuis une JSP.</p>
        <p>
            ${test}
            ${auteur}
        </p>
        <p>
            Récupération du bean :
            ${coyote.prenom}
            ${coyote.nom}
        </p>
        <p>
            Récupération de la liste :
            <%
            List<Integer> liste = (List<Integer>)
request.getAttribute( "liste" );
```

```
                for( Integer i : liste ){
                    out.println(i + " : ");
                }
                %>
            </p>
            <p>
                Récupération du jour du mois :
<%
Integer jourDuMois = (Integer) request.getAttribute( "jour" );
if ( jourDuMois % 2 == 0 ){
out.println("Jour pair : " + jourDuMois);
} else {
out.println("Jour impair : " + jourDuMois);
}
%>
            </p>
        </body>
</html>
```

Encore une fois, voilà à quoi nous en sommes réduits côté JSP : écrire du code Java pour faire un simple test sur un entier...

D'autant plus que je ne vous ai ici proposé que deux exemples basiques, mais nous pourrions lister bien d'autres fonctionnalités qu'il serait intéressant de pouvoir utiliser dans nos vues, et qui ne nous sont pas accessibles à travers la technologie JSP sans utiliser de scriptlets !

Dans ce cas, comment faire ? Comment ne pas écrire de code Java dans nos pages JSP ?

Eh bien des développeurs se sont posés la même question **il y a plus de dix ans déjà**, et ont imaginé la JSTL : une bibliothèque de balises préconçues, qui permettent à la manière des balises JSP de mettre en place les fonctionnalités dont nous avons besoin couramment dans une vue, mais qui ne sont pas accessibles nativement au sein de la technologie JSP.

Le point sur ce qu'il nous manque

Avant d'attaquer l'apprentissage de cette fameuse JSTL, prenons deux minutes pour faire **le point sur les limites de ce que nous avons appris** jusqu'à présent.

La vue

Nous devons impérativement nettoyer nos lunettes ! Nous savons afficher des choses basiques, mais dès que notre vue se complexifie un minimum, nous ne savons plus faire. Vous êtes déjà au courant, c'est vers la JSTL que nous allons nous tourner : l'intégralité de la partie suivante lui est d'ailleurs consacrée.

L'interaction

Vous ne vous en êtes peut-être pas encore rendus compte, mais nous n'avons qu'effleuré la récupération de données envoyées par le client ! Nous devons mettre en place de l'interaction : une application web qui ne demande rien à l'utilisateur, c'est un site statique, et nous ce que nous souhaitons c'est une application dynamique ! Pour le moment, nous avons uniquement développé des vues basiques, couplées à des servlets qui ne faisaient presque rien. Très bientôt, nous allons découvrir que les servlets auront pour objectif de TOUT contrôler : tout ce qui arrivera dans notre application et tout ce qui en sortira passera par nos servlets.

Les données

Nous devons apprendre à gérer nos données : pour le moment, nous avons uniquement découvert ce qu'était un bean. Nous avons une vague idée de comment seront représentées nos données au sein du modèle : à chaque entité de données correspondra un bean... Toutefois, nous nous heurtons ici à de belles inconnues : d'où vont venir nos données ? Qu'allons nous mettre dans nos beans ? Comment allons-nous sauvegarder les données qu'ils contiendront ? Comment enregistrer ce que nous transmet le client ? Nous devrons pour répondre à tout cela apprendre à manipuler une base de données depuis notre application. Ainsi, nous allons découvrir que notre modèle sera en réalité constitué non pas d'une seule couche, mais de deux ! Miam !

Documentation

On n'insistera jamais assez sur l'importance, pour tout zéro souhaitant apprendre quoi que ce soit, d'avoir recours aux documentations et ressources externes en général. Le site du zéro n'est pas une bible, tout n'y est pas ; pire, des éléments sont parfois volontairement omis ou simplifiés afin de bien vous faire comprendre certains points au détriment d'autres, jugés moins

cruciaux.

Les tutoriaux d'auteurs différents vous feront profiter de nouveaux points de vue et angles d'attaque, et les documentations officielles vous permettront un accès à des informations justes et maintenues à jour (en principe).

Liens utiles

- Base de connaissances portant sur Tomcat et les serveurs d'applications, sur Tomcat's Corner
- À propos des servlets, sur stackoverflow.com
- À propos des JSP, sur stackoverflow.com
- À propos des expressions EL, sur stackoverflow.com
- Base de connaissances portant sur les servlets, sur novocode.com
- FAQ générale à propos du développement autour de Java, sur jguru.com
- Tutoriel sur les Expression Language, dans la documentation officielle J2EE 1.4 sur sun.com
- La syntaxe JSP, sur sun.com

Vous l'aurez compris, cette liste ne se veut pas exhaustive, et je vous recommande d'aller chercher par vous même l'information sur les forums et sites du web. En outre, faites bien attention aux dates de création des documents que vous lisez : **les ressources périmées font légion sur le web, notamment au sujet de la plate-forme Java EE**, en constante évolution. N'hésitez pas à demander à la communauté sur le forum Java du Site du Zéro, si vous ne parvenez pas à trouver l'information que vous cherchez.

TP Fil rouge - Étape 1

Comme je vous l'annonçais en avant-propos, vous êtes ici pour apprendre à créer un projet web, en y ajoutant de la complexité au fur et à mesure que le cours avance. Avec tout ce que vous venez de découvrir, vous voici prêts pour poser la première pierre de l'édifice ! Je vous propose, dans cette première étape de notre TP fil rouge qui vous accompagnera jusqu'au terme de votre apprentissage, de revoir et appliquer l'ensemble des notions abordées jusqu'à présent.

Objectifs
Contexte

J'ai choisi la thématique du commerce en ligne comme source d'inspiration : vous allez créer un embryon d'application qui va permettre la création et la visualisation de clients et de commandes. C'est à la fois assez global pour ne pas impliquer d'éléments qui vous sont encore inconnus, et assez spécifique pour coller avec ce que vous avez appris dans ces chapitres et êtes capables de réaliser.

L'objectif premier de cette étape, c'est de vous familiariser avec le développement web sous Eclipse. Vous allez devoir mettre en place un projet depuis zéro dans votre environnement, et y créer vos différents fichiers. Le second objectif est de vous faire prendre de l'aise avec l'utilisation de servlets, de pages JSP et de beans, et de manière générale avec le principe général d'une application Java EE.

Fonctionnalités

Création de client

À travers notre petite application, l'utilisateur devra pouvoir créer un client en saisissant des données depuis un formulaire, et visualiser la fiche client en résultant. Puisque vous n'avez pas encore découvert les formulaires, je vais vous fournir cette page qui vous servira de base. Votre travail sera de coder :

- un bean, représentant un client ;
- une servlet, chargée de récupérer les données envoyées par le formulaire, de les enregistrer dans le bean et de les transmettre à une JSP ;
- une JSP, chargée d'afficher la fiche du client créé, c'est-à-dire les données transmises par la servlet.

Création de commande

L'utilisateur devra également pouvoir créer une commande, en saisissant des données depuis un formulaire, et visualiser la fiche en résultant. De même ici, puisque vous n'avez pas encore découvert les formulaires, je vais vous fournir cette page qui vous servira de base. Votre travail sera de coder :

- un bean, représentant une commande ;
- une servlet, chargée de récupérer les données envoyées par le formulaire, de les enregistrer dans le bean et de les transmettre à une JSP ;
- une JSP, chargée d'afficher la fiche de la commande créée, c'est-à-dire les données transmises par la servlet.

Contraintes

Comme je viens de vous l'annoncer, vous devez utilisez ces deux formulaires en guise de base pour votre application. Vous les placerez directement à la racine de votre application, sous le répertoire **WebContent** d'Eclipse.

Création de client

Code : JSP - /creerClient.jsp

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Création d'un client</title>
        <link type="text/css" rel="stylesheet" href="inc/style.css"
/>
```

```
        </head>
        <body>
            <div>
                <form method="get" action="creationClient">
                    <fieldset>
                        <legend>Informations client</legend>

                        <label for="nomClient">Nom <span
class="requis">*</span></label>
                        <input type="text" id="nomClient"
name="nomClient" value="" size="20" maxlength="20" />
                        <br />

                        <label for="prenomClient">Prenom </label>
                        <input type="text" id="prenomClient"
name="prenomClient" value="" size="20" maxlength="20" />
                        <br />

                        <label for="adresseClient">Adresse de livraison
<span class="requis">*</span></label>
                        <input type="text" id="adresseClient"
name="adresseClient" value="" size="20" maxlength="20" />
                        <br />

                        <label for="telephoneClient">Numéro de téléphone
<span class="requis">*</span></label>
                        <input type="text" id="telephoneClient"
name="telephoneClient" value="" size="20" maxlength="20" />
                        <br />

                        <label for="emailClient">Adresse email</label>
                        <input type="email" id="emailClient"
name="emailClient" value="" size="20" maxlength="60" />
                        <br />
                    </fieldset>
                    <input type="submit" value="Valider"  />
                    <input type="reset" value="Remettre à zéro" /> <br
/>
                </form>
            </div>
        </body>
    </html>
```

Création de commande

Code : JSP - /creerCommande.jsp

```
    <!DOCTYPE html>
    <html>
        <head>
            <meta charset="utf-8" />
            <title>Création d'une commande</title>
            <link type="text/css" rel="stylesheet" href="inc/style.css"
/>
        </head>
        <body>
            <div>
                <form method="get" action="creationCommande">
                    <fieldset>
                        <legend>Informations client</legend>

                        <label for="nomClient">Nom <span
class="requis">*</span></label>
                        <input type="text" id="nomClient"
name="nomClient" value="" size="20" maxlength="20" />
                        <br />
```

```
                    <label for="prenomClient">Prenom </label>
                    <input type="text" id="prenomClient"
name="prenomClient" value="" size="20" maxlength="20" />
                    <br />

                    <label for="adresseClient">Adresse de livraison
<span class="requis">*</span></label>
                    <input type="text" id="adresseClient"
name="adresseClient" value="" size="20" maxlength="20" />
                    <br />

                    <label for="telephoneClient">Numéro de téléphone
<span class="requis">*</span></label>
                    <input type="text" id="telephoneClient"
name="telephoneClient" value="" size="20" maxlength="20" />
                    <br />

                    <label for="emailClient">Adresse email</label>
                    <input type="email" id="emailClient"
name="emailClient" value="" size="20" maxlength="60" />
                    <br />
                </fieldset>
                <fieldset>
                    <legend>Informations commande</legend>

                    <label for="dateCommande">Date <span
class="requis">*</span></label>
                    <input type="text" id="dateCommande"
name="dateCommande" value="" size="20" maxlength="20" disabled />
                    <br />

                    <label for="montantCommande">Montant <span
class="requis">*</span></label>
                    <input type="text" id="montantCommande"
name="montantCommande" value="" size="20" maxlength="20" />
                    <br />

                    <label for="modePaiementCommande">Mode de
paiement <span class="requis">*</span></label>
                    <input type="text" id="modePaiementCommande"
name="modePaiementCommande" value="" size="20" maxlength="20" />
                    <br />

                    <label for="statutPaiementCommande">Statut du
paiement</label>
                    <input type="text" id="statutPaiementCommande"
name="statutPaiementCommande" value="" size="20" maxlength="20" />
                    <br />

                    <label for="modeLivraisonCommande">Mode de
livraison <span class="requis">*</span></label>
                    <input type="text" id="modeLivraisonCommande"
name="modeLivraisonCommande" value="" size="20" maxlength="20" />
                    <br />

                    <label for="statutLivraisonCommande">Statut de
la livraison</label>
                    <input type="text" id="statutLivraisonCommande"
name="statutLivraisonCommande" value="" size="20" maxlength="20" />
                    <br />
                </fieldset>
                <input type="submit" value="Valider"  />
                <input type="reset" value="Remettre à zéro" /> <br
/>
            </form>
        </div>
    </body>
</html>
```

Feuille de style

Voici pour finir une feuille de style que je vous propose d'utiliser pour ce TP. Toutefois, si vous êtes motivés vous pouvez très bien créer et utiliser vos propres styles, cela n'a pas d'importance.

Code : CSS - /inc/style.css

```css
/* Général ------------------------------------------------------------
---------------------------- */

body, p, legend, label, input {
    font: normal 8pt verdana, helvetica, sans-serif;
}

/* Forms --------------------------------------------------------------
---------------------------- */
fieldset {
    padding: 10px;
    border: 1px #0568CD solid;
    margin: 10px;
}

legend {
    font-weight: bold;
    color: #0568CD;
}

form label {
    float: left;
    width: 200px;
    margin: 3px 0px 0px 0px;
}

form input {
    margin: 3px 3px 0px 0px;
    border: 1px #999 solid;
}

form input.sansLabel {
    margin-left: 200px;
}

/* Styles et couleurs -------------------------------------------------
---------------------------- */
.requis {
    color: #c00;
}

.erreur {
    color: #900;
}

.succes {
    color: #090;
}

.info {
    font-style: italic;
    color: #E8A22B;
}
```

Conseils
À propos des formulaires

Voici les seules informations que vous devez connaître pour attaquer :

- ces deux formulaires sont configurés pour envoyer les données saisies vers les adresses **/creationClient** et **/creationCommande**. Lorsque vous mettrez en place vos deux servlets, vous devrez donc effectuer un mapping respectivement sur chacune de ces deux adresses dans votre fichier **web.xml** ;
- les données seront envoyées via la méthode GET du protocole HTTP, vous devrez donc implémenter la méthode `doGet()` dans vos servlets ;
- le nom des paramètres créés lors de l'envoi des données correspond au contenu des attributs `"name"` de chaque balise **`<input/>`** des formulaires. Par exemple, le nom du client étant saisi dans la balise **`<input`** `name="nomClient"...` **`/>`** du formulaire, il sera accessible depuis votre servlet par un appel à `request.getParameter("nomClient")`. Etc.
- j'ai volontairement désactivé le champ de saisie de la date dans le formulaire de création d'une commande. Nous intégrerons bien plus tard un petit calendrier permettant le choix d'une date, mais pour le moment nous ferons sans.

Le modèle

Vous n'allez travailler que sur deux entités, à savoir un client et une commande. Ainsi, deux objets suffiront :

- un bean **Client** représentant les données récupérées depuis le formulaire **creerClient.jsp** (nom, prénom, adresse, etc.) ;
- un bean **Commande** représentant les données récupérées depuis la seconde partie du formulaire **creerCommande.jsp** (date, montant, mode de paiement, etc.).

N'hésitez pas à relire le chapitre sur les Javabeans pour vous rafraîchir la mémoire sur leur structure.

Note 1 : au sujet du type des propriétés de vos beans, je vous conseille de toutes les déclarer de type `String`, sauf le montant de la commande que vous pouvez éventuellement déclarer de type `double`.
Note 2 : lors de la création d'une commande, l'utilisateur va devoir saisir des informations relatives au client. Plutôt que de créer une propriété pour chaque champ relatif au client (nom, prénom, adresse, etc.) dans votre bean **Commande**, vous pouvez directement y inclure une propriété de type **Client**, qui à son tour contiendra les propriétés nom, prénom, etc..

Les contrôleurs

Les deux formulaires que je vous fournis sont paramétrés pour envoyer les données saisies au serveur par le biais d'une requête de type GET. Vous allez donc devoir créer deux servlets, que vous pouvez par exemple nommer **CreationClient** et **CreationCommande**, qui vont pour chaque formulaire :

- récupérer les paramètres saisis, en appelant `request.getParameter()` sur les noms des différents champs ;
- les convertir dans le type souhaité si certaines des propriétés de vos beans ne sont pas des `String`, puis les enregistrer dans le bean correspondant ;
- vérifier si l'utilisateur a oublié de saisir certains paramètres requis (ceux marqués d'une étoile sur le formulaire de saisie) :
  - si oui, alors transmettre les beans et un message d'erreur à une page JSP pour affichage ;
  - si non, alors transmettre les beans et un message de succès à une page JSP pour affichage.

Note : puisque le champ de saisie de la date est désactivé, vous allez devoir initialiser la propriété **date** du bean **Commande** avec la date courante. Autrement dit, vous allez considérer que la date d'une commande est simplement la date courante lors de la validation du formulaire. Vous allez donc devoir récupérer directement la date courante depuis votre servlet et l'enregistrer au format `String` dans le bean.

Les vues

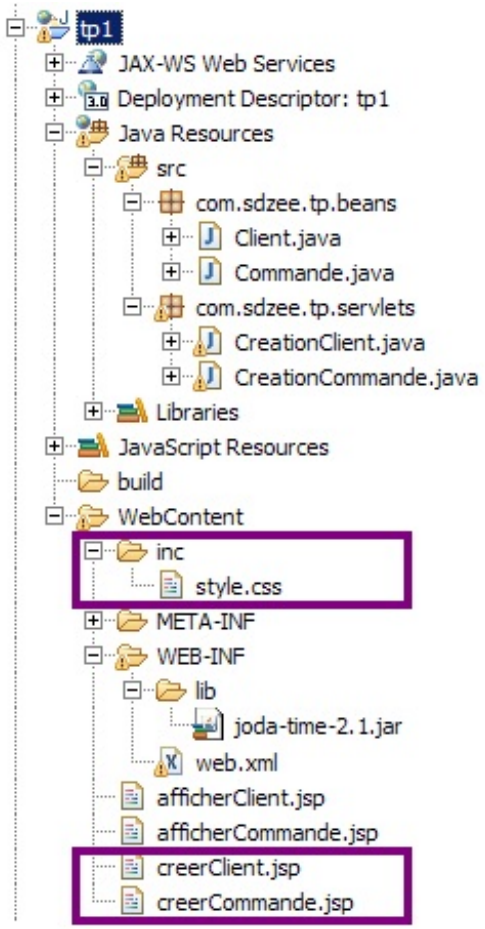
Je vous fournis les deux pages JSP contenant les formulaires de saisie des données. Les seules vues que vous avez à développer sont celles qui afficheront les données saisies, après validation du formulaire. Vous pouvez par exemple les nommer **afficherClient.jsp** et **afficherCommande.jsp**. Elles vont recevoir un ou plusieurs beans et un message depuis leur servlet respective, et devront afficher les données contenues dans ces objets. Vous l'avez probablement déjà deviné, les expressions EL sont la solution idéale ici !

Vous êtes libres au niveau de la mise en forme des données et des messages affichés, ce qui importe ce n'est pas le rendu graphique de vos pages, mais bien leur capacité à afficher correctement ce qu'on attend d'elles !

Création du projet

Avant de vous lâcher dans la nature, revenons rapidement sur la mise en place du projet. N'hésitez pas à relire le chapitre sur la configuration d'un projet si vous avez encore des incertitudes à ce sujet. Je vous conseille de créer un projet dynamique à partir de zéro dans Eclipse, que vous pouvez par exemple nommer **tp1**, basé sur le serveur Tomcat 7 que nous avons déjà mis en place dans le cadre du cours. Vous devrez alors configurer le build-path comme nous avons appris à le faire dans le chapitre sur les Javabeans. Vous allez par ailleurs devoir manipuler une date lors de la création d'une commande : je vous encourage pour cela à utiliser la bibliothèque JodaTime que nous avons découverte dans le chapitre précédent.

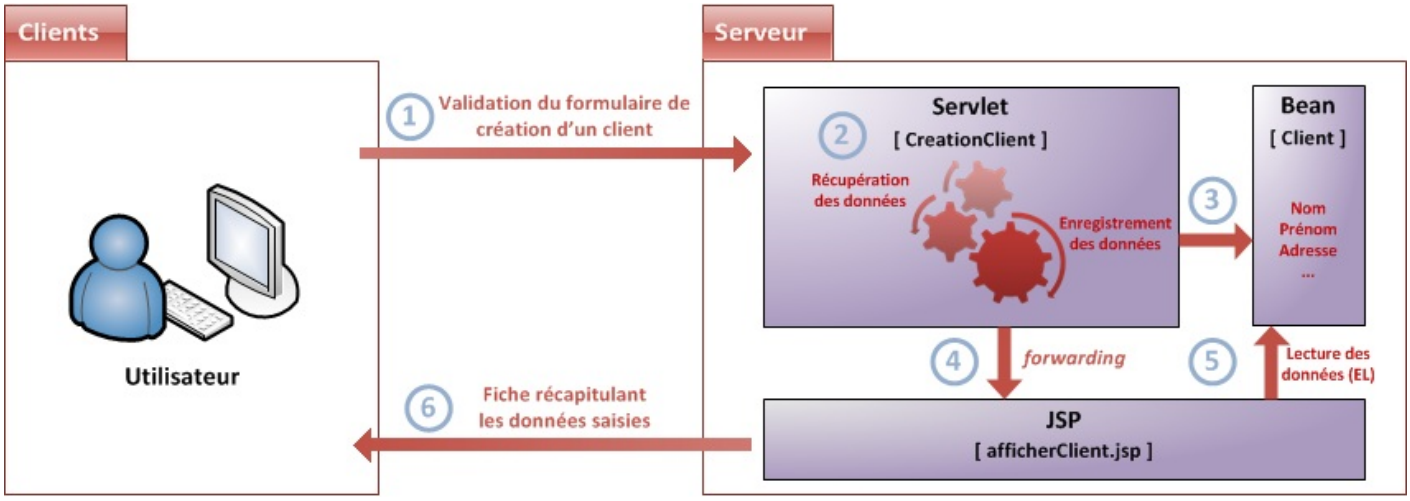
Pour conclure, voici à quoi est supposée ressembler l'architecture de votre projet fini si vous avez suivi mes conseils et exemples de nommage :



Vous pouvez observer en encadré sur cette image le positionnement des trois fichiers dont je vous ai fourni le code.

Illustration du comportement attendu

Voici sous forme d'un schéma ce que vous devez réaliser dans le cas de la création d'un client :

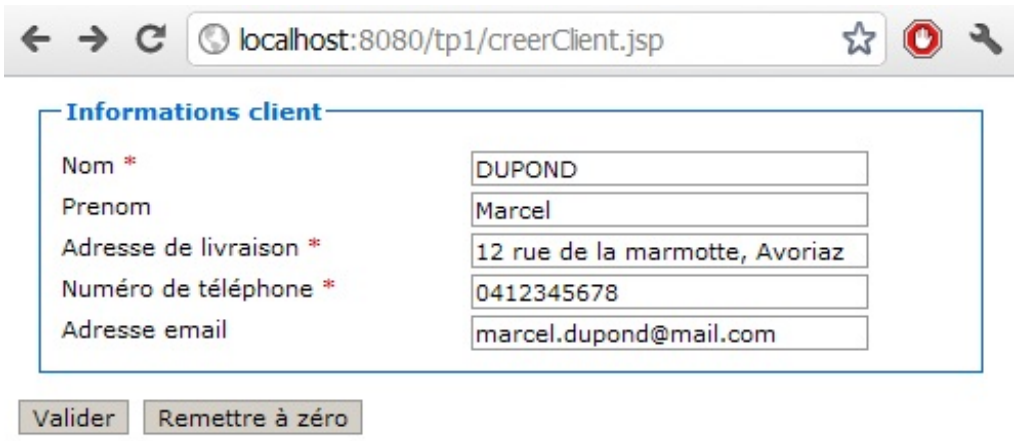


Je ne vous illustre pas la création d'une commande, le principe étant très similaire !

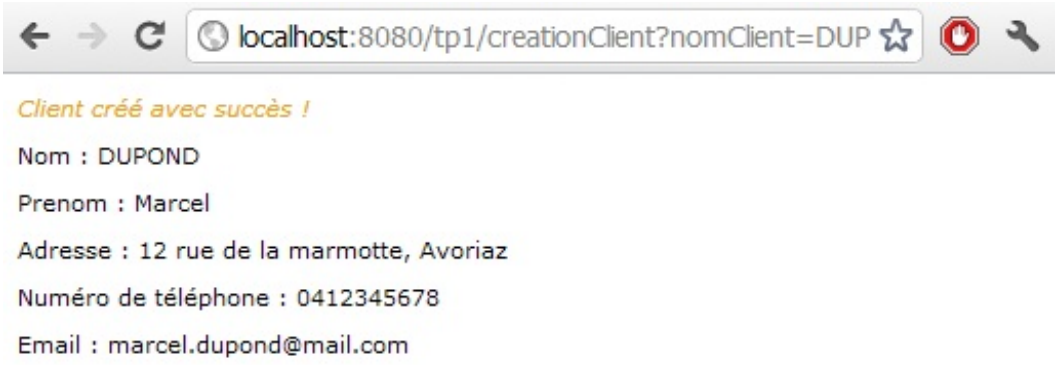
Exemples de rendu du comportement attendu

Création d'un client

Avec succès :

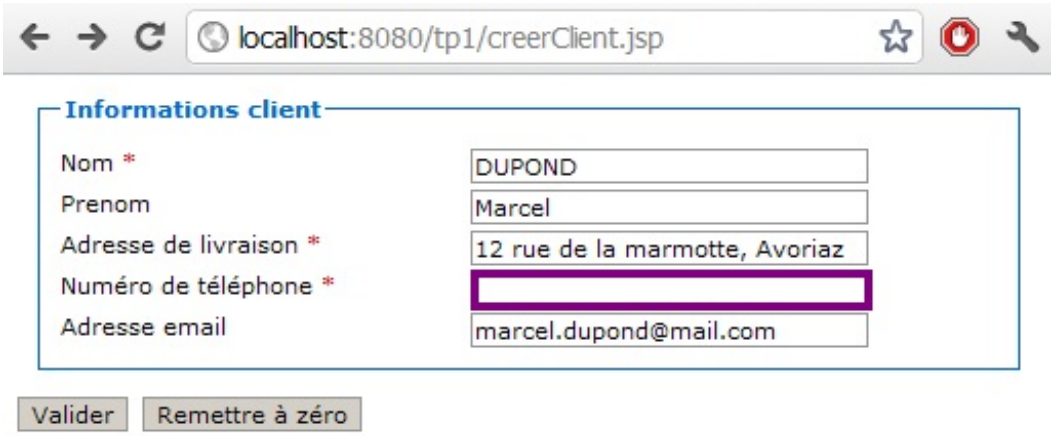


Saisie de données valides dans le formulaire



Affichage du message de succès et des données

Avec erreur :



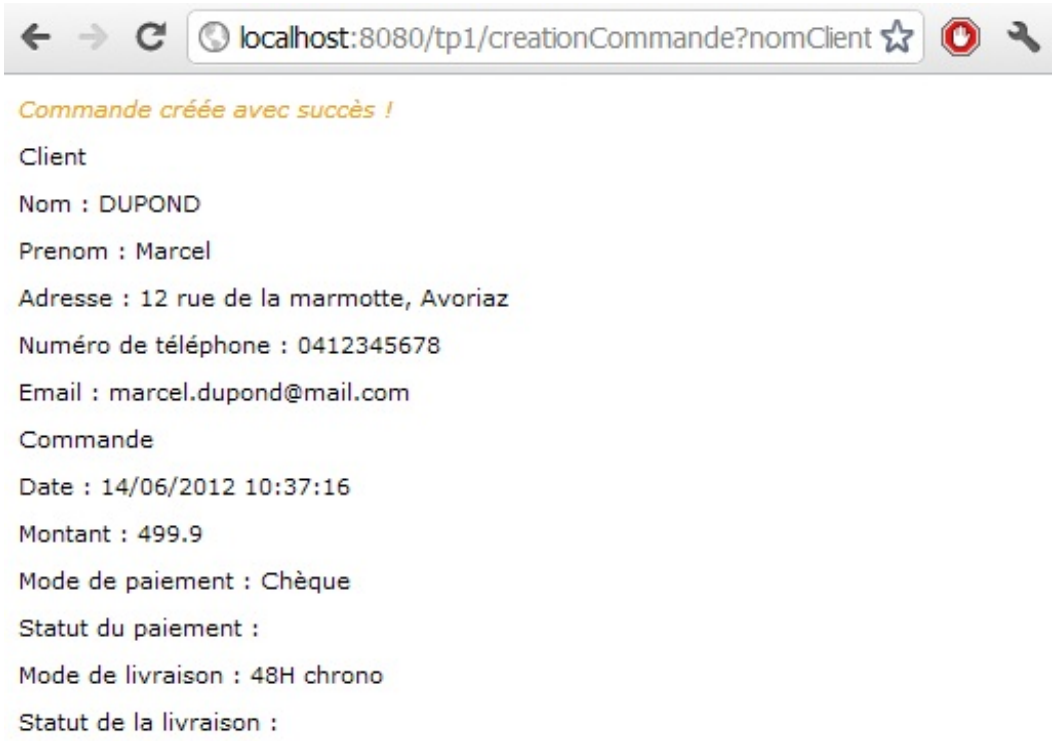
Oubli d'un champ obligatoire dans le formulaire

Affichage du message d'erreur et des données

Création d'une commande

Avec succès :

Saisie de données valides dans le formulaire



Affichage du message de succès et des données

Avec erreur :

Oubli de champs obligatoires et saisie d'un montant erroné dans le formulaire

Affichage du message d'erreur

et des données

Correction

Je vous propose cette solution en guise de correction. Ce n'est pas la seule manière de faire, ne vous inquiétez pas si vous avez procédé différemment, si vous avez nommé vos objets différemment ou si vous avez bloqué sur certains éléments. Le code est commenté et vous est parfaitement accessible : il ne contient que des instructions et expressions que nous avons déjà abordées dans les chapitres précédents.

Prenez le temps de créer votre projet depuis zéro, puis de chercher et coder par vous-mêmes. Si besoin, n'hésitez pas à relire le sujet ou à retourner lire certains chapitres. La pratique est très importante, ne vous jetez pas sur la solution avant d'avoir ~~essayé~~ réussi !

Le code des beans

Secret (cliquez pour afficher)

Code : Java - com.sdzee.tp.beans.Client

```
package com.sdzee.tp.beans;

public class Client {
    /* Propriétés du bean */
    private String nom;
    private String prenom;
    private String adresse;
    private String telephone;
    private String email;

    public void setNom( String nom ) {
        this.nom = nom;
    }

    public String getNom() {
        return nom;
    }

    public void setPrenom( String prenom ) {
        this.prenom = prenom;
    }
```

```
    public String getPrenom() {
        return prenom;
    }

    public void setAdresse( String adresse ) {
        this.adresse = adresse;
    }

    public String getAdresse() {
        return adresse;
    }

    public String getTelephone() {
        return telephone;
    }

    public void setTelephone( String telephone ) {
        this.telephone = telephone;
    }

    public void setEmail( String email ) {
        this.email = email;
    }

    public String getEmail() {
        return email;
    }
}
```

Code : Java - com.sdzee.tp.beans.Commande

```
package com.sdzee.tp.beans;


public class Commande {
    /* Propriétés du bean */
    private Client client;
    private String date;
    private Double montant;
    private String modePaiement;
    private String statutPaiement;
    private String modeLivraison;
    private String statutLivraison;

    public Client getClient() {
        return client;
    }

    public void setClient( Client client ) {
        this.client = client;
    }

    public String getDate() {
        return date;
    }

    public void setDate( String date ) {
        this.date = date;
    }

    public Double getMontant() {
        return montant;
    }
```

```
    public void setMontant( Double montant ) {
        this.montant = montant;
    }

    public String getModePaiement() {
        return modePaiement;
    }

    public void setModePaiement( String modePaiement ) {
        this.modePaiement = modePaiement;
    }

    public String getStatutPaiement() {
        return statutPaiement;
    }

    public void setStatutPaiement( String statutPaiement ) {
        this.statutPaiement = statutPaiement;
    }

    public String getModeLivraison() {
        return modeLivraison;
    }

    public void setModeLivraison( String modeLivraison ) {
        this.modeLivraison = modeLivraison;
    }

    public String getStatutLivraison() {
        return statutLivraison;
    }

    public void setStatutLivraison( String statutLivraison ) {
        this.statutLivraison = statutLivraison;
    }
}
```

Le code des servlets

Secret (cliquez pour afficher)

Configuration des servlets dans le fichier **web.xml** :

Code : XML - /WEB-INF/web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
 <servlet>
  <servlet-name>CreationClient</servlet-name>
  <servlet-class>com.sdzee.tp.servlets.CreationClient</servlet-
class>
 </servlet>
 <servlet>
  <servlet-name>CreationCommande</servlet-name>
  <servlet-class>com.sdzee.tp.servlets.CreationCommande</servlet-
class>
 </servlet>

 <servlet-mapping>
  <servlet-name>CreationClient</servlet-name>
  <url-pattern>/creationClient</url-pattern>
 </servlet-mapping>
 <servlet-mapping>
```

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

Servlet gérant le formulaire de création de client :

Code : Java - com.sdzee.tp.servlets.CreationClient

```
package com.sdzee.tp.servlets;

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.sdzee.tp.beans.Client;

public class CreationClient extends HttpServlet {

    public void doGet( HttpServletRequest request,
HttpServletResponse response ) throws ServletException,
IOException {
        /*
* Récupération des données saisies, envoyées en tant que
paramètres de
* la requête GET générée à la validation du formulaire
*/
        String nom = request.getParameter( "nomClient" );
        String prenom = request.getParameter( "prenomClient" );
        String adresse = request.getParameter( "adresseClient" );
        String telephone = request.getParameter( "telephoneClient"
);
        String email = request.getParameter( "emailClient" );

        String message;
        /*
* Initialisation du message à afficher : si un des champs
obligatoires
* du formulaire n'est pas renseigné, alors on affiche un message
* d'erreur, sinon on affiche un message de succès
*/
        if ( nom.trim().isEmpty() || adresse.trim().isEmpty() ||
telephone.trim().isEmpty() ) {
            message = "Erreur - Vous n'avez pas rempli tous les
champs obligatoires. <br> <a href=\"creerClient.jsp\">Cliquez
ici</a> pour accéder au formulaire de création d'un client.";
        } else {
            message = "Client créé avec succès !";
        }
        /*
* Création du bean Client et initialisation avec les données
récupérées
*/
        Client client = new Client();
        client.setNom( nom );
        client.setPrenom( prenom );
        client.setAdresse( adresse );
        client.setTelephone( telephone );
        client.setEmail( email );

        /* Ajout du bean et du message à l'objet requête */
        request.setAttribute( "client", client );
        request.setAttribute( "message", message );
```

```
        /* Transmission à la page JSP en charge de l'affichage
des données */
        this.getServletContext().getRequestDispatcher(
"/afficherClient.jsp" ).forward( request, response );
    }
}
```

Servlet gérant le formulaire de création de commande :

Code : Java - com.sdzee.tp.servlets.CreationCommande

```
package com.sdzee.tp.servlets;

import java.io.IOException;

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

import org.joda.time.DateTime;
import org.joda.time.format.DateTimeFormat;
import org.joda.time.format.DateTimeFormatter;

import com.sdzee.tp.beans.Client;
import com.sdzee.tp.beans.Commande;

public class CreationCommande extends HttpServlet {

    public void doGet( HttpServletRequest request,
HttpServletResponse response ) throws ServletException,
IOException {
        /*
* Récupération des données saisies, envoyées en tant que
paramètres de
* la requête GET générée à la validation du formulaire
*/
        String nom = request.getParameter( "nomClient" );
        String prenom = request.getParameter( "prenomClient" );
        String adresse = request.getParameter( "adresseClient" );
        String telephone = request.getParameter( "telephoneClient"
);
        String email = request.getParameter( "emailClient" );

        /* Récupération de la date courante */
        DateTime dt = new DateTime();
        /* Conversion de la date en String selon le format
défini */
        DateTimeFormatter formatter = DateTimeFormat.forPattern(
"dd/MM/yyyy HH:mm:ss" );
        String date = dt.toString( formatter );
        double montant;
        try {
            /* Récupération du montant */
            montant = Double.parseDouble( request.getParameter(
"montantCommande" ) );
        } catch ( NumberFormatException e ) {
            /* Initialisation à -1 si le montant n'est pas un
nombre correct */
            montant = -1;
        }
        String modePaiement = request.getParameter(
"modePaiementCommande" );
        String statutPaiement = request.getParameter(
"statutPaiementCommande" );
```

```
        String modeLivraison = request.getParameter(
"modeLivraisonCommande" );
        String statutLivraison = request.getParameter(
"statutLivraisonCommande" );

        String message;
        /*
* Initialisation du message à afficher : si un des champs
obligatoires
* du formulaire n'est pas renseigné, alors on affiche un message
* d'erreur, sinon on affiche un message de succès
*/
        if ( nom.trim().isEmpty() || adresse.trim().isEmpty() ||
telephone.trim().isEmpty() || montant == -1
                || modePaiement.isEmpty() ||
modeLivraison.isEmpty() ) {
            message = "Erreur - Vous n'avez pas rempli tous les
champs obligatoires. <br> <a href=\"creerCommande.jsp\">Cliquez
ici</a> pour accéder au formulaire de création d'une commande.";
        } else {
            message = "Commande créée avec succès !";
        }
        /*
* Création des beans Client et Commande et initialisation avec
les
* données récupérées
*/
        Client client = new Client();
        client.setNom( nom );
        client.setPrenom( prenom );
        client.setAdresse( adresse );
        client.setTelephone( telephone );
        client.setEmail( email );

        Commande commande = new Commande();
        commande.setClient( client );
        commande.setDate( date );
        commande.setMontant( montant );
        commande.setModePaiement( modePaiement );
        commande.setStatutPaiement( statutPaiement );
        commande.setModeLivraison( modeLivraison );
        commande.setStatutLivraison( statutLivraison );

        /* Ajout du bean et du message à l'objet requête */
        request.setAttribute( "commande", commande );
        request.setAttribute( "message", message );

        /* Transmission à la page JSP en charge de l'affichage
des données */
        this.getServletContext().getRequestDispatcher(
"/afficherCommande.jsp" ).forward( request, response );
    }
}
```

Le code des JSP

Secret (cliquez pour afficher)

Page d'affichage d'un client :

Code : JSP - afficherClient.jsp

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Affichage d'un client</title>
        <link type="text/css" rel="stylesheet"
href="inc/style.css" />
    </head>
    <body>
        <%-- Affichage de la chaine "message" transmise par la
servlet --%>
        <p class="info">${ message }</p>
        <%-- Puis affichage des données enregistrées dans le bean
"client" transmis par la servlet --%>
        <p>Nom : ${ client.nom }</p>
        <p>Prenom : ${ client.prenom }</p>
        <p>Adresse : ${ client.adresse }</p>
        <p>Numéro de téléphone : ${ client.telephone }</p>
        <p>Email : ${ client.email }</p>
    </body>
</html>
```

Page d'affichage d'une commande :

Code : JSP - afficherCommande.jsp

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Affichage d'une commande</title>
        <link type="text/css" rel="stylesheet"
href="inc/style.css" />
    </head>
    <body>
        <%-- Affichage de la chaine "message" transmise par la
servlet --%>
        <p class="info">${ message }</p>
        <%-- Puis affichage des données enregistrées dans le bean
"commande" transmis par la servlet --%>
        <p>Client</p>
        <%-- Les 5 expressions suivantes accèdent aux propriétés
du client, qui est lui-même une propriété du bean commande --%>
        <p>Nom : ${ commande.client.nom }</p>
        <p>Prenom : ${ commande.client.prenom }</p>
        <p>Adresse : ${ commande.client.adresse }</p>
        <p>Numéro de téléphone : ${ commande.client.telephone
}</p>
        <p>Email : ${ commande.client.email }</p>
        <p>Commande</p>
        <p>Date  : ${ commande.date }</p>
        <p>Montant  : ${ commande.montant }</p>
        <p>Mode de paiement  : ${ commande.modePaiement }</p>
        <p>Statut du paiement  : ${ commande.statutPaiement }</p>
        <p>Mode de livraison  : ${ commande.modeLivraison }</p>
        <p>Statut de la livraison  : ${ commande.statutLivraison
}</p>
    </body>
</html>
```

Encore une fois prenez votre temps, lisez bien et analysez attentivement les codes. Si jamais vous ne comprenez pas certains

points, n'hésitez pas à me préciser ce qui vous bloque par MP ou dans les commentaires du chapitre !

Partie 3 : Une bonne vue grâce à la JSTL

Après une brève introduction sur les objectifs de la JSTL, découvrez-ici sa mise en place dans un projet, les bases de sa bibliothèque principale et la manipulation de documents XML.

Objectifs et configuration

Après une brève introduction sur quelques concepts intervenant dans la suite de ce cours, et sur les versions de la JSTL, découvrez-ici les fichiers de configuration clés de votre projet ainsi que les paramètres importants à modifier pour mettre en place la bibliothèque dans votre projet web Java EE.

C'est sa raison d'être... ♪♫

La JSTL est une bibliothèque, une collection regroupant des balises implémentant des fonctionnalités à des fins générales communes aux applications web. Citons par exemple la mise en place de boucles, de tests conditionnels, le formatage des données ou encore la manipulation de données XML. Son objectif est de permettre au développeur d'éviter l'utilisation de code Java dans les pages JSP, et ainsi de respecter au mieux le découpage en couche recommandé par le modèle MVC. En apparence, ces balises ressemblent comme deux gouttes d'eau aux balises JSP que vous avez découvertes dans les chapitres précédents !

La liste d'avantages que je vous présente ci-dessous n'est probablement pas exhaustive. Je vais tenter de vous faire comprendre l'intérêt de l'utilisation de balises en vous exposant les aspects positifs qui me semblent les plus importants, et vous illustrer pourquoi l'utilisation de code Java dans vos pages JSP est déconseillée.

Lisibilité du code produit

Un des gros avantages de l'utilisation des balises JSTL, c'est sans aucun doute la lisibilité du code, et donc sa maintenabilité. Un exemple étant bien plus parlant que des mots, voici une simple boucle dans une JSP en Java d'abord (à base de scriptlet donc), et via des balises JSTL ensuite. Ne vous inquiétez pas de voir apparaître des notations qui vous sont pour le moment inconnues, les explications viendront par la suite :

Une boucle avec une scriptlet Java

Code : JSP

```
<%@ page import="java.util.List, java.util.ArrayList" %>
<%
 List<Integer> list =
(ArrayList<Integer>)request.getAttribute("tirage");
 for(int i = 0; i < list.size();i++){
  out.println(list.get(i));
 }
%>
```

Pas besoin de vous faire un dessin : c'est du Java...

La même boucle avec des tags JSTL

Code : JSP

```
<c:forEach var="item" items="${tirage}" >
 <c:out value="${item}" />
</c:forEach>
```

La boucle ainsi réalisée est nettement plus lisible, ne fait plus intervenir d'attributs et méthodes Java comme `size()`, `get()` ou encore des déclarations de variable ni de types d'objets (List, ArrayList, Date, etc.) , mais uniquement des balises à la syntaxe proche du XML qui ne gênent absolument pas la lecture du code et de la structure de la page.

Pour information, mais vous le saurez bien assez tôt, la bibliothèque de balises (on parle souvent de *tags*) ici utilisée, indiquée par le préfixe c:, est la bibliothèque *Core*, que vous découvrirez dans le chapitre suivant.

Moins de code à écrire

Un autre gros avantage de l'utilisation des balises issues des bibliothèques standard est la réduction de la quantité de code à écrire. En effet, le moins vous aurez à écrire de code, le moins vous serez susceptible d'introduire des erreurs dans vos pages. La syntaxe de nombreuses actions est simplifiée et raccourcie en utilisant la JSTL, ce qui permet d'éviter les problèmes dus à de bêtes fautes de frappe ou d'inattention dans des scripts en Java.

En outre, l'usage des scriptlets (le code Java entouré de `<% %>`) est **fortement déconseillé**, et ce depuis l'apparition des TagLibs (notamment la JSTL) et des EL, soit depuis une dizaine d'années maintenant. Les principaux maux dont souffrent les scriptlets sont :

1. **Réutilisation** : il est impossible de réutiliser une scriptlet dans une autre page, il faut la dupliquer. Cela signifie que lorsque vous avez besoin d'effectuer le même traitement dans une autre page JSP, vous n'aurez pas d'autre choix que de recopier le morceau de code dans l'autre page, et ce pour chaque page nécessitant ce bout de code. La duplication de code dans une application est bien entendu l'ennemi du bien : cela compromet énormément la maintenance de l'application.
2. **Interface** : il est impossible de rendre une scriptlet `abstract`.
3. **POO** : il est impossible dans une scriptlet de tirer parti de l'héritage ou de la composition.
4. **Debug** : si une scriptlet envoie une exception en cours d'exécution, tout s'arrête et l'utilisateur récupère une page blanche...
5. **Tests** : on ne peut pas écrire de tests unitaires pour tester les scriptlets. Lorsqu'un développeur travaille sur une application relativement large, il doit s'assurer que ses modifications n'impactent pas le code existant et utilise pour cela une batterie de tests dits "unitaires", qui ont pour objectif de vérifier le fonctionnement des différentes méthodes implémentées. Eh bien ceux-ci ne peuvent pas s'appliquer au code Java écrit dans une page JSP : là encore, cela compromet énormément la maintenance et l'évolutivité de l'application.
6. **Maintenance** : inéluctablement, il faut passer énormément plus de temps à maintenir un code mélangé, encombré, dupliqué et non testable !

À titre informatif, la maison mère Oracle elle-même recommande dans ses JSP coding conventions d'éviter l'utilisation de code Java dans une JSP autant que possible, notamment via l'utilisation de balises :

Citation : Extrait des conventions de codage JSP

> *« Where possible, avoid JSP scriptlets whenever tag libraries provide equivalent functionality. This makes pages easier to read and maintain, helps to separate business logic from presentation logic, and will make your pages easier to evolve [...] »*

Vous avez dit MVC ?

Ne plus écrire de Java directement dans vos JSP

Vous l'avez probablement remarqué dans les exemples précédents : le Java complique énormément la lecture d'une page JSP. Alors certes, ici je ne vous ai présenté qu'une gentille petite boucle, donc la différence n'est pas si flagrante. Mais imaginez que vous travailliez sur un projet de plus grande envergure, mettant en jeu des pages HTML avec un contenu autrement plus riche, voire sur un projet dans le cadre duquel vous n'êtes pas l'auteur des pages que vous avez à maintenir ou à modifier : que préféreriez-vous aborder ? Sans aucune hésitation, lorsque les balises JSTL sont utilisées, la taille des pages est fortement réduite et leurs compréhension et maintenance s'en retrouvent grandement facilitées.

Rendre à la Vue son vrai rôle

Soyez bien conscients d'une chose : je ne vous demande pas de proscrire le Java de vos pages JSP juste pour le plaisir des yeux !

Si je vous encourage à procéder ainsi, c'est pour vous faire prendre de bonnes habitudes : la vue, en l'occurrence nos JSP, ne doit se consacrer qu'à l'affichage. Ne pas avoir à déclarer de méthodes dans une JSP, ne pas modifier directement des données depuis une JSP, ne pas y insérer de traitement métier... Tout cela est recommandé, mais la frontière peut paraître bien mince si on se laisse aller à utiliser des scriptlets Java dès que l'occasion s'en présente. Avec les tags JSTL, la séparation est bien plus nette.

Un autre point positif, qui ne vous concerne pas vraiment si vous ne travaillez pas en entreprise sur des projets de grande

envergure, est que le modèle MVC permet une meilleure séparation des couches de l'application. Par exemple, imaginez une application dont le code Java est bien caché dans la couche métier (au hasard, dans des beans) : le(s) programmeur(s) UI très performant(s) en interface utilisateur peu(ven)t donc se baser sur la simple documentation du code métier pour travailler sur la couche de présentation en créant les vues, les JSP donc, et ce sans avoir à écrire ni lire de Java, langage qu'ils ne maitrisent pas aussi bien, voire pas.

À retenir

Si vous ne deviez retenir qu'une phrase de tout cela, c'est que bafouer MVC en écrivant du code Java directement dans une JSP rend la maintenance d'une application extrêmement compliquée, et par conséquent réduit fortement son évolutivité. Libre à vous par conséquent de décider de l'avenir que vous souhaitez donner à votre projet, en suivant ou non les recommandations.

Dernière couche : on écrit du code Java directement dans une JSP uniquement lorsqu'il nous est impossible de faire autrement, ou lorsque l'on désire vérifier un fonctionnement via une simple feuille de tests, et enfin pourquoi pas lorsque l'on souhaite rapidement écrire un prototype temporaire afin de se donner une idée du fonctionnement d'une application de très faible envergure. Voilà, j'espère que maintenant vous l'avez bien assimilé, ce n'est pas faute de vous l'avoir répété...

Plusieurs versions

La JSTL a fait l'objet de plusieurs versions :

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

Les différences entre ces versions se traduisent principalement par le conteneur JSP nécessaire. Le changement majeur à retenir dans le passage de la première version à la seconde version de ce conteneur, c'est la gestion des EL. Le conteneur JSP 1.2 sur lequel est basée la JSTL 1.0 ne gérait pas les EL, cette dernière proposait donc deux implémentations pour pallier ce manque : une interprétant les EL et l'autre non. Ceci se traduisait alors par l'utilisation d'adresses différentes lors de la déclaration des bibliothèques.

La version 1.1 est basée sur le conteneur JSP 2.0, qui intègre nativement un interpréteur d'EL, et ne propose par conséquent plus qu'une seule implémentation.

Ce tutoriel se base quant à lui sur la version actuelle, à savoir la JSTL 1.2, qui d'après le site officiel apporte des EL "unifiées", ainsi qu'une meilleure intégration dans le framework JSF. Ces changements par rapport à la précédente version n'ont aucun impact sur ce cours : tout ce qui suit sera valable, que vous souhaitiez utiliser la version 1.1 ou 1.2 de la JSTL.

Configuration

Configuration de la JSTL

Il y a plusieurs choses que vous devez savoir ici. Plutôt que de vous donner tout de suite les solutions aux problèmes qui vous attendent, fonçons têtes baissées, et je vous guiderai lorsque cela s'avèrera nécessaire. On apprend toujours mieux en faisant des erreurs et en apprenant à les corriger, qu'en suivant bêtement une série de manipulations.

D'erreur en erreur...

Allons-y gaiement donc, et tentons naïvement d'insérer une balise JSTL ni vu ni connu dans notre belle et vierge page JSP :

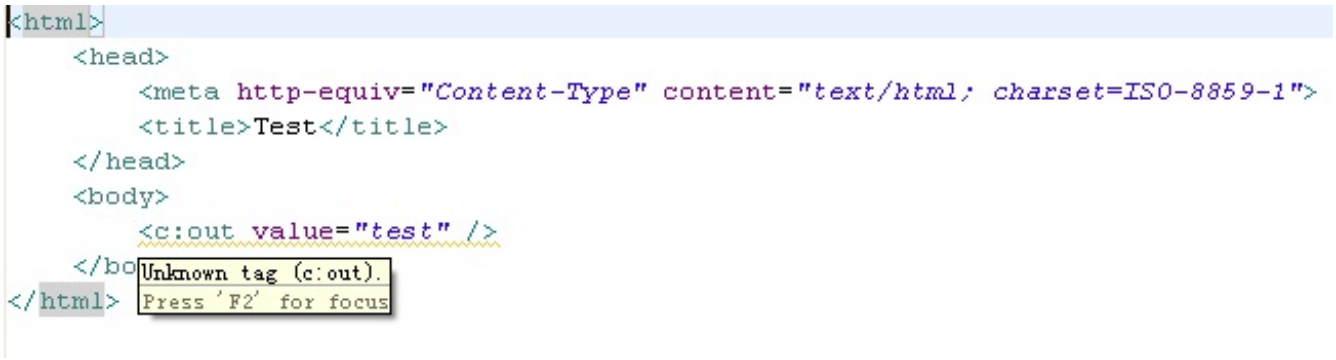
Code : JSP - Une balise JSTL dans notre page

```
<html>
 <head>
  <meta http-equiv="Content-Type" content="text/html; charset=ISO-
8859-1">
  <title>Test</title>
 </head>
 <body>
<c:out value="test" />
 </body>
```

```
</html>
```

Pour le moment, cette notation vous est inconnue, nous y reviendrons en temps voulu. Vous pouvez d'ores et déjà constater que cette balise a une syntaxe très similaire à celle des actions standard JSP. Pour votre information seulement, il s'agit ici d'un tag JSTL issu de la bibliothèque *Core*, permettant d'afficher du texte dans une page. Relativement basique donc...

Basique, sur le principe oui. Mais Eclipse vous signale alors une première erreur :



Il ne connait visiblement pas cette balise. Et pour cause : puisqu'il est issu d'une bibliothèque (la JSTL), il est nécessaire de préciser à Eclipse où ce *tag* est réellement défini ! Et si vous avez suivi la partie précédente de ce cours, vous devez vous souvenir d'une certaine directive JSP, destinée à inclure des bibliothèques... Cela vous revient ? Tout juste, c'est la directive **taglib** que nous allons utiliser ici. Voici donc notre code modifié pour inclure la bibliothèque *Core* :

Code : JSP - Ajout de la directive taglib

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
 <head>
  <meta http-equiv="Content-Type" content="text/html; charset=ISO-
8859-1">
  <title>Test</title>
 </head>
 <body>
  <c:out value="test" />
 </body>
</html>
```

Étudions cette directive :

- dans le paramètre **uri** se trouve le lien vers la définition de la bibliothèque. Remarquez bien ici l'arborescence de ce lien : **/jsp/jstl/core**. Si vous travailliez sur des codes qui ne sont pas de vous, vous serez éventuellement amenés à rencontrer dans cette balise un lien de la forme **/jstl/core** : sachez que ce type de lien est celui de la version antérieure 1.0 de la JSTL. En effet, le dossier **jsp** a été rajouté afin d'éviter toute ambiguïté avec les précédentes versions, qui comme je vous l'ai précisé en première partie ne géraient pas les EL. Faites bien attention à utiliser le bon lien selon la version de la JSTL que vous souhaitez utiliser, sous peine de vous retrouver avec des erreurs peu compréhensibles...
- dans le paramètre **prefix** se trouve l'alias qui sera utilisé dans notre page JSP pour faire appel aux balises de la bibliothèque en question. Concrètement, cela signifie que si je souhaite appeler le *tag* **if** de la bibliothèque *Core*, je dois écrire **<c:if>**. Si j'avais entré "core" dans le champ **prefix** de la directive au lieu de "c", je devrais alors écrire **<core:if>**.

Là, je suppose que vous vous apprêtez à me jeter des bûches. En effet, s'il est vrai qu'Eclipse vous signalait une alerte auparavant, vous vous retrouvez maintenant avec une nouvelle erreur en plus de la précédente !

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
        <title>Test</title>
    </head>
    <body>
        <c:out value="test" />
    </body>
</html>
```

Can not find the tag library descriptor for "http://java.sun.com/jsp/jstl/core"
Press 'F2' for focus

Effectivement, nouvelle erreur. Pourquoi ?

Eh bien cette fois, c'est Tomcat qui est en cause ! Dans ce chapitre d'introduction, lorsque je vous avais présenté Tomcat, je vous avais bien précisé qu'il n'était pas un serveur d'applications Java EE au sens complet du terme. Nous voilà devant une première illustration de cet état de fait : alors que la JSTL fait partie intégrante de la plate-forme Java EE 6, Tomcat 7 n'est par défaut pas livré avec la JSTL. Si vous utilisez par exemple le serveur Glassfish d'Oracle, qui quant à lui respecte bien les spécifications Java EE, vous ne rencontrerez pas de problème : la JSTL y est bien incluse.

La lumière étant faite sur l'origine de cette erreur, il est temps de la corriger. Maintenant que nous avons précisé la définition de notre bibliothèque, il faut définir quelque part où se situe physiquement cette bibliothèque, et donc configurer notre projet afin qu'il puisse accéder à ses fichiers sources. Si vous êtes un peu curieux et que vous vous souvenez de ce que nous avons dû faire pour utiliser l'API JodaTime dans la partie précédente, vous avez probablement déjà remarqué que dans le dossier **/WEB-INF** de votre projet, il y a un dossier nommé... **lib** !

Le chemin semble donc tout tracé, nous devons aller chercher notre bibliothèque. Où la trouver ? J'y reviendrai dans le chapitre suivant, la JSTL contient nativement plusieurs bibliothèques, et *Core* est l'une d'entre elles. Par conséquent, c'est l'archive jar de la JSTL tout entière que nous allons devoir ajouter à notre projet. Vous pouvez télécharger le jar **jstl-1.2.jar** en cliquant sur ce lien de téléchargement direct. Vous voilà donc en possession du fichier que vous allez devoir copier dans votre répertoire **lib** :

Ça commence à bien faire, nous tournons en rond ! Nous avons inclus notre bibliothèque, mais nous avons toujours nos deux erreurs !
Que s'est-il passé ?

Pas d'inquiétude, nous apercevons le bout du tunnel... Effectivement, Eclipse vous crie toujours dessus. Mais ce n'est cette fois que pure illusion !

***Note** : la raison pour laquelle Eclipse ne met pas directement ses avertissements à jour, je n'en suis pas vraiment certain. J'imagine que l'environnement a besoin d'une modification postérieure à la mise en place des bibliothèques pour prendre en compte complètement la modification. Bref, modifiez simplement votre page JSP, en y ajoutant un simple espace ou ce que vous voulez, et sauvez. Comme par magie, Eclipse cesse alors de vous crier dessus !*

Il ne vous reste plus qu'à démarrer votre Tomcat si ce n'est pas déjà fait, et vérifier que tout se passe bien, en accédant à votre JSP depuis votre navigateur : `http://localhost:8080/TestJSTL/test.jsp`. Le mot "test" devrait alors s'afficher : félicitations, vous venez de mettre en place et utiliser avec succès votre premier tag JSTL !

Si vous ne parvenez toujours pas à faire fonctionner votre page, reprenez attentivement et dans l'ordre chaque étape présentée ci-dessus, et si malgré cette relecture vous restez sur un échec, dirigez-vous vers le forum Java du site pour y exposer votre problème en détail.

Dans le prochain chapitre, nous allons aborder la bibliothèque principale de la JSTL, très justement nommée *Core*.

La bibliothèque Core

Nous voici prêts à étudier la bibliothèque *Core*, offrant des balises pour les principales actions nécessaires dans la couche présentation d'une application web. Ce chapitre va en quelque sorte faire office de documentation : je vais vous y présenter les principales balises de la bibliothèque, et expliciter leur rôle et comportement via des exemples simples.

Lorsque ces bases seront posées, nous appliquerons ce que nous aurons découvert ici dans un TP. S'il est vrai que l'on ne peut se passer de la théorie, pratiquer est également indispensable si vous souhaitez assimiler et progresser.

Les variables et expressions

Pour commencer, nous allons apprendre comment afficher le contenu d'une variable ou d'une expression, et comment gérer une variable et sa portée. Avant cela, je vous donne ici **la directive JSP nécessaire** pour permettre l'utilisation des balises de la bibliothèque Core dans vos pages :

Code : JSP

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

Cette directive devra **être présente sur chacune des pages de votre projet** utilisant les balises JSTL que je vous présente dans ce chapitre. Dans un prochain chapitre, nous verrons comment il est possible de ne plus avoir à se soucier de cette commande. En attendant, ne l'oubliez pas !

Affichage d'une expression

La balise utilisée pour l'affichage est `<c:out value="" />`. Le seul attribut obligatoirement requis pour ce *tag* est **value**. Cet attribut peut contenir une chaîne de caractères simple, ou une expression EL. Voici quelques exemples :

Code : JSP

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

A celui-ci s'ajoutent deux attributs optionnels :

- **default** : permet de définir une valeur affichée par défaut si le contenu de l'expression évaluée est vide.
- **escapeXml** : permet de remplacer les caractères de scripts <, >, ", ' et & par leurs équivalents en code html `<`, `>`, `"`, `'`, `&`. Cette option est activée par défaut, et vous devez expliciter `<c:out ... escapeXml="false" />` pour la désactiver.

Pourquoi utiliser une balise pour afficher simplement du texte ou une expression ?

C'est une question légitime. Après tout c'est vrai, pourquoi ne pas directement écrire le texte ou l'expression dans notre page JSP ? Pourquoi s'embêter à inclure le texte ou l'expression dans cette balise ? Eh bien la réponse se trouve dans l'explication de l'attribut optionnel **escapeXml** : celui-ci est **activé par défaut** ! Cela signifie que l'utilisation de la balise `<c:out />` permet d'échapper automatiquement les caractères spéciaux de nos textes et rendus d'expressions, et c'est là une excellente raison d'utilisation (voir ci-dessous l'avertissement concernant les failles XSS).

Voici des exemples d'utilisation de l'attribut **default** :

Code : JSP

```
<%-- Cette balise affichera le mot 'test' si le bean n'existe pas :
--%>
<c:out value="${bean}">
    test
</c:out>

<%-- Elle peut également s'écrire sous cette forme : --%>
<c:out value="${bean}" default="test" />
```

```
<%-- Et il est interdit d'écrire : --%>
<c:out value="${bean}" default="test">
    une autre chaine
</c:out>
```

Pour le dernier cas, l'explication est simple : l'attribut **default** jouant déjà le rôle de valeur par défaut, le corps du *tag* ne peut exister que lorsqu'aucune valeur par défaut n'est définie. Pour information, Eclipse vous signalera une erreur si vous tentez d'écrire la balise sous cette forme.

Pour en finir avec cette balise, voici un exemple d'utilisation de l'attribut **escapeXml** :

Code : JSP

```
<%-- Sans préciser d'attribut escapeXml : --%>
<c:out value="<p>Je suis un 'paragraphe'.</p>" />

<%-- La balise affichera : --%>
&lt;p&gt;Je suis un &#039;paragraphe&#039;.&lt;/p&gt;

<%-- Et en précisant l'attribut à false :--%>
<c:out value="<p>Je suis un 'paragraphe'.</p>" escapeXml="false" />

<%-- La balise affichera : --%>
<p>Je suis un 'paragraphe'.</p>
```

Vous pouvez constater dans cet exemple l'importance de l'activation par défaut de l'option **escapeXml** : elle empêche l'interprétation de ce qui est affiché par le navigateur, en modifiant les éléments de code HTML présents dans le contenu traité (en l'occurrence les caractères <, > et ').

Vous devez prendre l'habitude d'utiliser ce tag JSTL lorsque vous affichez des variables, notamment lorsqu'elles sont récupérées depuis le navigateur, c'est-à-dire lorsqu'elle sont saisies par l'utilisateur. Prenons l'exemple d'un formulaire :

Code : JSP

```
<%-- Mauvais exemple --%>
<input type="text" name="donnee"
value="${donneeSaisieParUnUtilisateur}" />

<%-- Bon exemple --%>
<input type="text" name="donnee" value="<c:out
value="${donneeSaisieParUnUtilisateur}"/>" />
```

Les données récupérées depuis un formulaire sont potentiellement dangereuses, puisqu'elles permettent des attaques de type XSS ou d'injection de code. L'utilisation du tag **<c:out/>** permet d'échapper les caractères spéciaux responsables de cette faille, et ainsi de prévenir tout risque à ce niveau. Nous reviendrons en détails sur cette faille dans le chapitre sur les formulaires.

Gestion d'une variable

Avant de parler variable, revenons sur leur **portée** ! La portée (ou visibilité) d'une variable correspond concrètement à l'endroit dans lequel elle est stockée, et par corolaire aux endroits depuis lesquels elle est accessible. Selon la portée affectée à votre variable, elle sera par exemple accessible depuis toute votre application, ou seulement depuis une page particulière, etc. Il y a quatre portées différentes (ou *scopes* en anglais), que vous connaissez déjà et redécouvrirez au fur et à mesure des exemples de ce chapitre :

- **la page** : les objets créés avec la portée **page** ne sont accessibles que depuis cette même page, et une fois la réponse retournée au navigateur ces données ne sont plus accessibles.
- **la requête** : les objets créés avec la portée **request** ne sont accessibles que depuis les pages qui traitent cette même requête. Si la requête est transmise à une autre page, les données sont conservées, mais sont perdues en cas de

redirection.

- **la session** : les objets créés avec la portée **session** ne sont accessibles que depuis les pages traitant les requêtes créées dans cette même session. Concrètement, une session correspond à la durée pendant laquelle un visiteur va utiliser l'application, cette durée se terminant lorsque l'utilisateur ferme son navigateur ou encore lorsque l'application le décide (le développeur, pour être exact), par exemple via un lien de déconnexion ou encore un temps maximum de validité imposé après lequel la session sera automatiquement détruite. Les données ainsi créées ne sont plus accessibles une fois que le visiteur quitte le site.
- **l'application** : les objets créés avec la portée **application** sont accessibles depuis toutes les pages JSP de l'application web ! C'est en quelque sorte une variable globale, accessible partout.

Création

La balise utilisée pour la création d'une variable est **<c:set />**. Abordons pour commencer la mise en place d'un attribut dans la requête. En JSP/servlets, vous savez tous faire ça, mais qu'en est-il avec la JSTL ? Il suffit d'utiliser les trois attributs suivants : **var**, **value** et **scope**.

Code : JSP

```
<%-- Cette balise met l'expression "Salut les zéros !" dans
l'attribut "welcomeMessage" de la requête : --%>
<c:set var="welcomeMessage" value="Salut les zéros !"
scope="request" />

<%-- Et est l'équivalent du scriplet Java suivant : --%>
<% request.setAttribute( "welcomeMessage", "Salut les zéros !" ); %>
```

L'attribut **var** contient le nom de la variable que l'on veut stocker, **value** sa valeur, et **scope** la portée de cette variable. Simple, rapide et efficace ! Voyons maintenant comment récupérer cette valeur pour l'afficher à l'utilisateur, par exemple :

Code : JSP

```
<%-- Affiche l'expression contenue dans la variable "welcomeMessage"
de la requête --%>
<c:out value="${requestScope.welcomeMessage}" />
```

Vous remarquerez que nous utilisons ici dans l'EL l'objet implicite **requestScope**, qui permet de rechercher un objet dans la portée requête uniquement. Les plus avertis d'entre vous ont peut-être tenté d'accéder à la valeur fraîchement créée via un simple **<c:out** `value="${ welcomeMessage }"` **/>**. Et effectivement, dans ce cas cela fonctionne également. Pourquoi ?

Nous retrouvons ici une illustration du mécanisme dont je vous ai parlé lorsque nous avons appliqué les EL dans notre code d'exemple : par défaut, si le terme de l'expression n'est ni un type primitif (`int`, `char`, `boolean`, etc.) ni un objet implicite de la technologie EL, l'expression va d'elle-même chercher un attribut correspondant à ce terme dans les différentes portées de votre application : **page**, puis **request**, puis **session** et enfin **application**.

Souvenez-vous, je vous avais expliqué que c'est grâce à l'objet implicite **pageContext** que le mécanisme parcourt toutes les portées, et qu'il renvoie alors automatiquement le premier objet trouvé lors de son parcours. Voilà donc pourquoi cela fonctionne avec la seconde écriture : puisque nous ne précisons pas de portée, l'expression EL les parcourt automatiquement une par une jusqu'à ce qu'elle trouve un objet nommé **welcomeMessage**, et nous le renvoie !

N'oubliez pas : la bonne pratique veut que vous ne donniez pas le même nom à deux variables différentes, présentes dans des portées différentes. Toutefois, afin d'éviter toute confusion si jamais des variables aux noms identiques venaient à coexister, il est également conseillé de n'utiliser la seconde écriture que lorsque que vous souhaitez faire référence à des attributs de portée **page**, et d'utiliser la première écriture que je vous ai présentée pour le reste (**session**, **request** et **application**).

Modification

La modification d'une variable s'effectue de la même manière que sa création. Ainsi, le code suivant créera une variable nommée "maVariable" si elle n'existe pas déjà, et initialisera son contenu à "12" :

Code : JSP

```
<%-- L'attribut scope n'est pas obligatoire. Rappelez-vous, le scope
par défaut est dans ce cas la page,
puisque c'est le premier dans la liste des scopes parcourus --%>
<c:set var="maVariable" value="12" />
```

Pour information, il est également possible d'initialiser une variable en utilisant le corps de la balise, plutôt qu'en utilisant l'attribut **value** :

Code : JSP

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

À ce sujet, sachez d'ailleurs qu'il est possible d'imbriquer d'autres balises dans le corps de cette balise, et pas seulement d'utiliser de simples chaînes de caractères ou expressions. Voici par exemple comment vous pourriez initialiser la valeur d'une variable de session depuis une valeur lue dans un paramètre de l'URL :

Code : JSP

```
<c:set var="locale" scope="session">
   <c:out value="${param.lang}" default="FR"/>
</c:set>
```

Plusieurs points importants ici :

- vous constatez bien ici l'utilisation de la balise **<c:out/>** à l'intérieur du corps de la balise **<c:set/>**.
- vous pouvez remarquer l'utilisation de l'objet implicite **param**, pour récupérer la valeur du paramètre de la requête nommé **lang** ;
- si le paramètre **lang** n'existe pas ou s'il est vide, c'est la valeur par défaut "FR" spécifiée dans notre balise **<c:out/>** qui sera utilisée pour initialiser notre variable en session.

Modification des propriétés d'un objet

Certains d'entre vous se demandent probablement comment il est possible de définir ou modifier une valeur particulière lorsqu'on travaille sur certains types d'objets... Et ils ont bien raison ! En effet, avec ce que je vous ai présenté pour le moment, vous êtes capables de définir une variable de n'importe quel type, type qui est défini par l'expression que vous écrivez dans l'attribut **value** du tag **<c:set/>** :

Code : JSP

```
<%-- Crée un objet de type String --%>
<c:set scope="session" var="description" value="Je suis une loutre."
/>

<%-- Crée un objet du type du bean ici spécifié dans l'attribut
'value'--%>
<c:set scope="session" var="tonBean" value="${monBean}" />
```

Et c'est ici que vous devez vous poser la question suivante : comment modifier les propriétés du bean créé dans cet exemple ? En effet, il vous manque deux attributs pour y parvenir ! Regardons donc de plus près quels sont ces attributs, et comment ils fonctionnent :

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

Code : JSP

```
<!-- Définir ou modifier la propriété 'prenom' du bean 'coyote' -->
<c:set target="${coyote}" property="prenom" value="Wile E."/>

<!-- Définir ou modifier la propriété 'prenom' du bean 'coyote' via
le corps de la balise -->
<c:set target="${coyote}" property="prenom">
   Wile E.
</c:set>

<!-- Passer à null la valeur de la propriété 'prenom' du bean
'coyote' -->
<c:set target="${coyote}" property="prenom" value="${null}" />
```

Remarquez dans le dernier exemple qu'il suffit d'utiliser une EL avec pour mot-clé **null** dans l'attribut **value** pour faire passer la valeur d'une propriété à **null**. Pour information, lorsque l'objet traité n'est pas un bean mais une simple `Map`, cette action a pour effet de directement supprimer l'entrée de la `Map` concernée : le comportement est alors identique avec la balise présentée dans le paragraphe suivant.

Suppression

Dernière étape : supprimer une variable. Une balise est dédiée à cette tâche, avec pour seul attribut requis **var**. Par défaut toujours, c'est le *scope* page qui sera parcouru si l'attribut **scope** n'est pas explicité :

Code : JSP

```
<%-- Supprime la variable "maVariable" de la session --%>
<c:remove var="maVariable" scope="session" />
```

Voilà déjà un bon morceau de fait ! Ne soyez pas abattus si vous n'avez pas tout compris lorsque nous avons utilisée des objets implicites. Nous y reviendrons de toute manière au fur et à mesure que nous en aurons besoin dans nos exemples, et vous comprendrez alors avec la pratique.

Les conditions

Une condition simple

La JSTL fournit deux moyens d'effectuer des tests conditionnels. Le premier, simple et direct, permet de tester une seule expression, et correspond au bloc `if()` du langage Java. Le seul attribut obligatoire est **test**.

Code : JSP

```
<c:if test="${ 12 > 7 }" var="maVariable" scope="session">
    Ce test est vrai.
</c:if>
```

Ici, le corps de la balise est une simple chaîne de caractères. Elle ne sera affichée dans la page finale que si la condition est vraie, à savoir si l'expression contenue dans l'attribut **test** renvoie **true**. Ici, c'est bien entendu le cas, 12 est bien supérieur à 7.

Les attributs optionnels **var** et **scope** ont ici sensiblement le même rôle que dans la balise **<c:set/>**. Le résultat du test conditionnel sera stocké dans la variable et dans le *scope* défini, et sinon dans le *scope* page par défaut. L'intérêt de cette utilisation réside principalement dans le stockage des résultats de tests coûteux, un peu à la manière d'un cache, afin de pouvoir les réutiliser en accédant simplement à des variables de *scope*.

Des conditions multiples

La seconde méthode fournie par la JSTL est utile pour traiter les conditions mutuellement exclusives, équivalentes en Java à une suite de `if() / else if()` ou au bloc `switch()`. Elle est en réalité constituée de plusieurs balises :

Code : JSP

```
<c:choose>
    <c:when test="${expression}">Action ou texte.</c:when>
    ...
    <c:otherwise>Autre action ou texte.</c:otherwise>
</c:choose>
```

La balise **<c:choose/>** ne peut contenir aucun attribut, et son corps ne peut contenir qu'une ou plusieurs balise(s) **<c:when/>** et une ou zéro balise **<c:otherwise/>**.

La balise **<c:when/>** ne peut exister qu'à l'intérieur d'une balise **<c:choose/>**. Elle est l'équivalent du mot-clé **case** en Java, dans un bloc `switch()`. Tout comme la balise **<c:if/>**, elle doit obligatoirement se voir définir un attribut **test** contenant la condition. À l'intérieur d'un même bloc **<c:choose/>**, un seul **<c:when/>** verra son corps évalué, les conditions étant mutuellement exclusives.

La balise **<c:otherwise/>** ne peut également exister qu'à l'intérieur d'une balise **<c:choose/>**, et après la dernière balise **<c:when/>**. Elle est l'équivalent du mot-clé **default** en Java, dans un bloc `switch()`. Elle ne peut contenir aucun attribut, et son corps ne sera évalué que si aucune des conditions la précédant dans le bloc n'est vérifiée.

Voilà pour les conditions avec la JSTL. Je ne pense pas qu'il soit nécessaire de prendre plus de temps ici, la principale différence avec les conditions en Java étant la syntaxe utilisée.

Les boucles

Abordons à présent la question des boucles. Dans la plupart des langages, les boucles ont une syntaxe similaire : **for**, **while**, **do/while**... Avec la JSTL, deux choix vous sont offerts, en fonction du type d'élément que vous souhaitez parcourir avec votre boucle : **<c:forEach/>** pour parcourir une collection, et **<c:forTokens/>** pour parcourir une chaîne de caractères.

Boucle "classique"

Prenons pour commencer une simple boucle **for** en scriptlet Java, affichant un résultat formaté dans un tableau HTML par exemple :

Code : JSP - Une boucle sans la JSTL

```
<%-- Boucle calculant le cube des entiers de 0 à 7 et les affichant
dans un tableau HTML --%>
<table>
  <tr>
    <th>Valeur</th>
    <th>Cube</th>
  </tr>

<%
int[] cube= new int[8];
/* Boucle calculant et affichant le cube des entiers de 0 à 7 */
for(int i = 0 ; i < 8 ; i++)
{
  cube[i] = i * i * i;
  out.println("<tr><td>" + i  + "</td> <td>" + cube[i] +
"</td></tr>");
}
%>

</table>
```

Avec la JSTL, si l'on souhaite réaliser quelque chose d'équivalent, il faudrait utiliser la syntaxe suivante :

Code : JSP - Une boucle avec la JSTL

```
<%-- Boucle calculant le cube des entiers de 0 à 7 et les affichant
dans un tableau HTML --%>
<table>
  <tr>
    <th>Valeur</th>
    <th>Cube</th>
  </tr>

<c:forEach var="i" begin="0" end="7" step="1">
  <tr>
    <td><c:out value="${i}"/></td>
    <td><c:out value="${i * i * i}"/></td>
  </tr>
</c:forEach>

</table>
```

Avant tout, on peut déjà remarquer la clarté du second code en comparaison avec le premier : les balises JSTL s'intègrent très bien au formatage HTML englobant les résultats. On devine rapidement ce que produira cette boucle, ce qui était bien moins évident avec le code en Java, pourtant tout aussi basique. Étudions donc les attributs de cette fameuse boucle :

- **begin** : la valeur de début de notre compteur (la valeur de **i** dans la boucle en Java, initialisé à zéro en l'occurrence).
- **end** : la valeur de fin de notre compteur. Vous remarquez ici que la valeur de fin est 7 et non pas 8, comme c'est le cas dans la boucle Java. La raison est simple : dans la boucle Java en exemple j'ai utilisé une comparaison stricte (i strictement inférieur à 8), alors que la boucle JSTL ne procède pas par comparaison stricte (i inférieur ou égal à 7). J'aurais certes pu écrire `i <= 7` dans ma boucle Java, mais je n'ai par contre pas le choix dans ma boucle JSTL, c'est uniquement ainsi. Pensez-y, c'est une erreur bête mais facile à commettre si l'on oublie ce comportement.
- **step** : c'est le pas d'incrémentation de la boucle. Concrètement, si vous changez cette valeur de 1 à 3 par exemple, alors le compteur de la boucle ira de 3 en 3 et non plus de 1 en 1. Par défaut, si vous ne spécifiez pas l'attribut **step**, la valeur 1 sera utilisée.
- **var** : cette attribut est, contrairement à ce qu'on pourrait croire *a priori*, non obligatoire. Si vous ne le spécifiez pas, vous ne pourrez simplement pas accéder à la valeur du compteur en cours (via la variable **i** dans notre exemple). Vous pouvez choisir de ne pas préciser cet attribut si vous n'avez pas besoin de la valeur du compteur à l'intérieur de votre boucle. Par ailleurs, tout comme en Java lorsqu'on utilise une syntaxe équivalente à l'exemple précédent (déclaration de l'entier **i** à l'intérieur du **for**), la variable n'est accessible qu'à l'intérieur de la boucle, autrement dit dans le corps de la balise **<c:forEach/>**.

Vous remarquerez bien évidemment que l'utilisation de tags JSTL dans le corps de la balise est autorisé : nous utilisons ici dans cet exemple l'affichage via des balises **<c:out/>**.

Voici, mais cela doit vous paraître évident, le code HTML produit par cette page JSP :

Code : HTML

```
<table>
  <tr>
    <th>Valeur</th>
    <th>Cube</th>
  </tr>
  <tr>
    <td>0</td>
    <td>0</td>
  </tr>
  <tr>
    <td>1</td>
    <td>1</td>
  </tr>
  <tr>
    <td>2</td>
```

```
      <td>8</td>
    </tr>
    <tr>
      <td>3</td>
      <td>27</td>
    </tr>
    <tr>
      <td>4</td>
      <td>64</td>
    </tr>
    <tr>
      <td>5</td>
      <td>125</td>
    </tr>
    <tr>
      <td>6</td>
      <td>216</td>
    </tr>
    <tr>
      <td>7</td>
      <td>343</td>
    </tr>
  </table>
```

Itération sur une collection

Passons maintenant à quelque chose de plus intéressant et utilisé dans la création de pages web : les itérations sur les **collections**. Si ce terme ne vous parle pas, c'est que vous avez besoin d'un bonne piqûre de rappel en Java ! Et ce n'est pas moi qui vous la donnerai, si vous en sentez le besoin, allez faire un tour sur ce chapitre du tuto de Java.

La syntaxe utilisée pour parcourir une collection est similaire à celle d'une boucle simple, sauf que cette fois, un attribut **items** est requis. Et pour cause, c'est lui qui indiquera la collection à parcourir. Imaginons ici que nous souhaitions réaliser l'affichage de news sur une page web. Imaginons pour cela que nous ayons à disposition un `ArrayList` ici nommé *maListe*, contenant simplement des HashMap. Chaque HashMap ici contiendra le titre d'une news et son contenu. Nous souhaitons alors parcourir cette liste afin d'afficher ces informations dans une page web :

Code : JSP

```
<%@ page import="java.util.*" %>
<%
  /* Création de la liste et des données */
  List<HashMap<String, String>> maListe = new
ArrayList<HashMap<String, String>>();
  HashMap<String, String> news = new HashMap<String, String>();
  news.put("titre", "Titre de ma première news");
  news.put("contenu", "corps de ma première news");
  maListe.add(news);
  news = new HashMap<String, String>();
  news.put("titre", "Titre de ma seconde news");
  news.put("contenu", "corps de ma seconde news");
  maListe.add(news);
  pageContext.setAttribute("maListe", maListe);
%>

<c:forEach items="${maListe}" var="news">
<div class="news">
  <div class="titreNews">
<c:out value="${news['titre']}" />
  </div>
  <div class="corpsNews">
<c:out value="${news['contenu']}" />
  </div>
</div>
</c:forEach>
```

Je sens que certains vont m'attendre au tournant... Eh oui, j'ai ici utilisé du code Java ! Et du code sale en plus ! Mais attention à ne pas vous y méprendre : je n'ai recours à du code Java ici que pour l'exemple, afin de vous procurer un moyen simple et rapide pour initialiser des données de test, et afin de vérifier le bon fonctionnement de notre boucle.

Il va de soi que dans une vraie application web, ces données seront initialisées correctement, et non pas salement comme je l'ai fait ici. Qu'elles soient récupérées depuis une base de données, depuis un fichier, voire codées en dur dans la couche métier de votre application, ces données ne doivent jamais et en aucun cas, je répète elles ne doivent jamais et en aucun cas, être initialisées directement depuis vos JSP. Le rôle d'une page JSP, je le rappelle, c'est de présenter l'information, un point c'est tout. Ce n'est pas pour rien que la couche dans laquelle se trouve les JSP s'appelle la couche de présentation.

Revenons à notre boucle : je n'ai ici pas encombré la syntaxe, en utilisant les seuls attributs **items** et **var**. Le premier indique la collection sur laquelle la boucle porte, en l'occurrence notre `List` nommé *maListe*, et le second indique quant à lui le nom de la variable qui sera liée à l'élément courant de la collection parcourue par la boucle, que j'ai ici de manière très originale nommée "news", nos HashMap contenant... des news. Ainsi, pour accéder respectivement aux titre et contenu de nos news, il suffit via des crochets de préciser les éléments visés dans notre map : `${news['titre']}` et `${news['contenu']}`. Les lignes contenant des éléments importants sont surlignées dans le code précédent.

Voici le rendu HTML de cet exemple :

Code : HTML

```
<div class="news">
  <div class="titreNews">
      Titre de ma première news
  </div>
  <div class="corpsNews">
      corps de ma première news
  </div>
</div>

<div class="news">
  <div class="titreNews">
      Titre de ma seconde news
  </div>
  <div class="corpsNews">
      corps de ma seconde news
  </div>
</div>
```

Les attributs présentés précédemment lors de l'étude d'une boucle simple sont là aussi valables : si vous souhaitez par exemple n'afficher que les dix premières news sur votre page, vous pouvez limiter le parcours de votre liste aux dix premiers éléments ainsi :

Code : JSP

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

Si les attributs **begin** et **end** spécifiés dépassent le contenu réel de la collection, par exemple si vous voulez afficher les dix premiers éléments d'une liste mais qu'elle n'en contient que trois, la boucle s'arrêtera automatiquement lorsque le parcours de la liste sera terminé, peu importe l'indice **end** spécifié.

Simple, n'est-ce pas ?

À titre d'information, voici enfin les différentes collections sur lesquelles il est possible d'itérer avec la boucle **<c:forEach/>** de la bibliothèque *Core* :

- `java.util.Collection` ;
- `java.util.Map` ;
- `java.util.Iterator` ;
- `java.util.Enumeration` ;
- Array d'objets ou de types primitifs ;
- ~~Chaînes de caractères séparées par des séparateurs définis.~~

Si j'ai barré le dernier élément, c'est parce qu'il est déconseillé d'utiliser cette boucle pour parcourir une chaîne de caractères dont les éléments sont séparés par des caractères séparateurs définis. Voyez le paragraphe suivant pour en savoir plus à ce sujet.

Enfin, sachez qu'il est également possible d'itérer directement sur le résultat d'une requête SQL. Je n'aborderai cependant volontairement pas ce cas, pour deux raisons :

- je ne vous ai pas encore présenté la bibliothèque sql de la JSTL, permettant d'effectuer des requêtes SQL depuis vos JSP ;
- je vous présenterai pas la bibliothèque sql de la JSTL, ne souhaitant pas vous voir effectuer des requêtes SQL depuis vos JSP !

L'attribut varStatus

Il reste un attribut que je n'ai pas encore évoqué, et qui est comme les autres utilisables pour tout type d'itérations, que ce soit sur des entiers ou sur des collections : l'attribut **varStatus**. Tout comme l'attribut **var**, il est utilisé pour créer une variable de *scope*, mais présente une différence majeure : alors que **var** permet de stocker la valeur de l'index courant ou l'élément courant de la collection parcourue, le **varStatus** permet de stocker un objet `LoopTagStatus`, qui définit un ensemble de propriétés définissant l'état courant d'une itération :

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

Reprenons l'exemple utilisé précédemment, mais cette fois-ci mettant en jeu l'attribut **varStatus** :

Code : JSP

```
<c:forEach items="${maListe}" var="news" varStatus="status">
<div class="news">
News n° <c:out value="${status.count}"/> :
  <div class="titreNews">
      <c:out value="${news['titre']}" />
  </div>
  <div class="corpsNews">
      <c:out value="${news['contenu']}" />
  </div>
</div>
</c:forEach>
```

J'ai ici utilisé la propriété **count** de l'attribut **varStatus**, et l'affiche ici simplement en tant que numéro de news. Cet exemple est simple, mais suffit à vous faire comprendre comment utiliser cet attribut : il suffit d'appeler directement une propriété de l'objet **varStatus**, que j'ai ici de manière très originale nommé... **status**.

Pour terminer, sachez enfin que l'objet créé par cet attribut **varStatus** n'est visible que dans le corps de la boucle, tout comme l'attribut **var**.

Itération sur une chaîne de caractères

Une variante de la boucle **<c:forEach/>** existe, spécialement dédiée aux chaînes de caractères. La syntaxe est presque identique :

Code : JSP

```
<p>
<%-- Affiche les différentes sous-chaînes séparées par une virgule
ou un point-virgule --%>
<c:forTokens var="sousChaine" items="salut; je suis un,gros;zéro+!"
delims=";,+">
  ${sousChaine}<br/>
</c:forTokens>
</p>
```

Un seul attribut apparaît : **delims**. C'est ici que l'on doit spécifier quels sont les caractères qui serviront de séparateurs dans la chaîne que la boucle parcourra. Il suffit de les spécifier les uns à la suite des autres, comme c'est le cas ici dans notre exemple. Tous les autres attributs vus précédemment peuvent également s'appliquer ici (**begin**, **end**, **step**...).

Le rendu HTML de ce dernier exemple est donc :

Code : HTML

```
<p>
  salut<br/>
   je suis un<br/>
  gros<br/>
  zero<br/>
  !<br/>
</p>
```

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

Il est possible en Java d'utiliser les commandes **break** et **continue** pour sortir d'une boucle en cours de parcours. Eh bien sachez que ces fonctionnalités ne sont pas implémentées dans la JSTL. Il est par conséquent la plupart du temps impossible de sortir d'une boucle en cours d'itération.

Il existe dans certains cas des moyens plus ou moins efficaces de sortir d'une boucle, via l'utilisation de conditions **<c:if />** notamment. Quant aux cas d'itérations sur des collections, la meilleure solution si le besoin de sortir en cours de boucle se fait ressentir est de déporter le travail de la boucle dans une classe Java. Pour résumer, ce genre de situations se résout au cas par cas, selon vos besoins. Mais n'oubliez pas, votre vue doit se consacrer à l'affichage uniquement : si vous sentez que vous avez besoin de fonctionnalités qui n'existent pas dans la JSTL, il y a de grandes chances pour que vous soyez en train de trop en demander à votre vue, et éventuellement de bafouer MVC !

Les liens

Liens

La balise **<c:url/>** a pour objectif de générer des URL. En lisant ceci, j'imagine que vous vous demandez ce qu'il peut bien y avoir de particulier à gérer dans la création d'une URL ! Dans une page HTML simple, lorsque l'on crée un lien on se contente en

effet d'écrire directement l'adresse au sein de la balise **<a>** :

Code : HTML

```
<a href="url">lien</a>
```

Dans ce cas, qu'est-ce qui peut motiver le développeur à utiliser la balise **<c:url/>** ?

Eh bien vous devez savoir qu'en réalité, un adresse n'est pas qu'une simple chaîne de caractères, elle est soumise à plusieurs contraintes.
Voici les trois fonctionnalités associées à la balise :

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

L'attribut **value** contient logiquement l'adresse, et l'attribut **var** permet comme pour les tags vus auparavant de stocker le résultat dans une variable. Voici un premier jeu d'exemples :

Code : JSP

```
<%-- Génère une url simple, positionnée dans un lien HTML --%>
<a href="<c:url value="test.jsp" />">lien</a>

<%-- Génère une url et la stocke dans la variable lien --%>
<c:url value="test.jsp" var="lien" />
```

Reprenons maintenant les trois propriétés en détail, et analysons leur fonctionnement.

1. Ajout du contexte

Lorsqu'une URL est absolue, c'est-à-dire lorsqu'elle fait référence à la racine de l'application et commence par le caractère / , le contexte de l'application sera par défaut ajouté en début d'adresse. Ceci est principalement dû au fait que lors du développement d'une application, le nom du contexte importe peu et on y écrit souvent un nom par défaut, faute de mieux. Il n'est généralement choisi définitivement que lors du déploiement de l'application, qui intervient en fin de cycle.

Lors de l'utilisation d'adresses relatives, pas de soucis puisqu'elles ne font pas référence au contexte, et pointeront quoi qu'il arrive vers le répertoire courant. Mais pour les adresses absolues, pointant à la racine, sans cette fonctionnalité il serait nécessaire d'écrire en dur le contexte de l'application dans les URL lors du développement, et de toutes les modifier si le contexte est changé par la suite lors du déploiement. Vous comprenez donc mieux l'intérêt d'un tel système.

Code : JSP

```
<%-- L'url absolue ainsi générée --%>
<c:url value="/test.jsp" />

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

<%-- Et une url relative ainsi générée --%>
<c:url value="test.jsp" />

<%-- Ne sera pas modifiée lors du rendu --%>
test.jsp
```

2. Gestion des sessions

Si le conteneur JSP détecte un cookie stockant l'identifiant de session dans le navigateur de l'utilisateur, alors aucune modification ne sera apportée à l'URL. Par contre, si ce cookie est absent, les URL générées via la balise **<c:url/>** seront réécrites pour intégrer l'identifiant de session en question. Regardez ces exemples, afin de bien visualiser la forme de ce paramètre :

Code : JSP

```
<%-- L'url ainsi générée --%>
<c:url value="test.jsp" />

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

<%-- Et sera rendue sous cette forme si le cookie est absent --%>
test.jsp;jsessionid=A6B57CE08012FB431D
```

Ainsi, via ce système une application Java EE ne dépendra pas de l'activation des cookies du côté utilisateur. Ne vous inquiétez pas si vous ne saisissez pas le principe ici, nous reviendrons sur cette histoire de cookies et de sessions plus tard. Pour le moment, essayez simplement de retenir que la balise **<c:url/>** est équipée pour leur gestion automatique !

3. Encodage

En utilisant la balise **<c:url/>**, les paramètres que vous souhaitez passer à cette URL seront encodés : les caractères spéciaux qu'ils contiennent éventuellement vont être transformés en leurs codes HTML respectifs. Toutefois, il ne faut pas faire de confusion ici : **ce sont seulement les paramètres (leur nom et contenu) qui seront encodés, le reste de l'URL ne sera pas modifié**. La raison de ce comportement est de pouvoir assurer la compatibilité avec l'action standard d'inclusion **<jsp:include/>**, qui ne sait pas gérer une URL encodée.

D'accord, mais comment faire pour passer des paramètres ?

Pour transmettre proprement des paramètres à une URL, une balise particulière existe : **<c:param/>**. Elle ne peut exister que dans le corps des balises **<c:url/>**, **<c:import/>** ou **<c:redirect/>**. Elle se présente sous cette forme, et est assez intuitive :

Code : JSP

```
<c:url value="/monSiteWeb/countZeros.jsp">
  <c:param name="nbZeros" value="${countZerosBean.nbZeros}"/>
  <c:param name="date" value="22/06/2010"/>
</c:url>
```

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

Code : JSP

```
<%-- Une URL générée de cette manière --%>
<a href="<c:url value="/monSiteWeb/test.jsp">
  <c:param name="date" value="22/06/2010"/>
  <c:param name="donnees" value="des données contenant des c@r#ct%res bi&a**es..
</c:url>">Lien HTML</a>

<%-- Sera rendue ainsi dans la page web finale --%>
<a href="/test/monSiteWeb/test.jsp?
date=22%2f06%2f2010&donnees=des+donn%e9es+contenant+des+c%40r%23ct%25res+bi%26a**
```

```
    HTML</a>
```

Vous voyez bien dans cet exemple que :

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

Si vous travaillez sur une page XML ou une page XHTML stricte, alors vous devez savoir qu'afin de respecter les normes qui régissent ces technologies, il est impératif d'encoder proprement l'URL. Cela dit, je viens de vous expliquer que la balise **<c:url/>** n'effectue pas cette opération, elle ne s'occupe que des paramètres... Par conséquent, vous devrez transformer vous-mêmes les caractères spéciaux, comme le & séparant les paramètres d'une URL, en leur code HTML équivalent (en l'occurrence, & doit devenir `&` pour que la syntaxe soit valide). Si vous avez bien suivi, vous savez qu'il est possible d'effectuer ces transformations à l'aide de la balise **<c:out/>** !

Pour résumer

Récapitulons tout cela avec un exemple assez complet :

Code : JSP

```
<%-- L'url avec paramètres ainsi générée --%>
<c:url value="/monSiteWeb/countZeros.jsp">
  <c:param name="nbZeros" value="123"/>
  <c:param name="date" value="22/06/2010"/>
</c:url>

<%-- Sera rendue ainsi dans la page web finale,
si le cookie est présent et le contexte est Test --%>
/Test/monSiteWeb/countZeros.jsp?nbZeros=123&date=22%2f06%2f2010

<%-- Et sera rendue sous cette forme si le cookie est absent --%>
/Test/monSiteWeb/countZeros.jsp;jsessionid=A6B57CE08012FB431D?
nbZeros=123&date=22%2f06%2f2010
```

Vous pouvez ici observer :

- la mise en place de paramètres via **<c:param/>** ;
- l'ajout automatique du contexte en début de l'URL absolue ;
- l'encodage automatique des paramètres (ici les caractères / dans la date sont transformés en %2f) ;
- le non-encodage de l'URL (le caractère & séparant les paramètres n'est pas transformé) ;
- l'ajout automatique de l'identifiant de session. Remarquez d'ailleurs ici sa présence avant les paramètres de la requête, et non après.

Redirection

La balise **<c:redirect/>** est utilisée pour envoyer un message de redirection HTTP au navigateur de l'utilisateur. Si elle ressemble à l'action **<jsp:forward/>**, il existe toutefois une grosse différence, qui réside dans le fait qu'elle va entraîner un changement de l'URL dans le navigateur de l'utilisateur final, contrairement à **<jsp:forward/>** qui est quant à elle transparente du point de vue de l'utilisateur (l'URL dans la barre de navigation du navigateur n'est pas modifiée).

La raison de cette différence de comportement est simple : le *forwarding* se fait côté serveur, contrairement à la redirection qui elle est effectuée par le navigateur. Cela limite par conséquent la portée de l'action de *forwarding*, qui puisqu'exécutée côté serveur est limitée aux pages présentes dans le contexte de la servlet utilisée. La redirection étant elle exécutée côté client, rien ne vous empêche de rediriger l'utilisateur vers n'importe quelle page web.

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

Bref, je vous conseille pour débuter d'utiliser la redirection via la balise **<c:redirect/>** plutôt que l'action standard JSP, cela vous évitera bien des ennuis. Voyons pour terminer quelques exemples d'utilisation :

Code : JSP

```
<%-- Forwarding avec l'action standard JSP --%>
<jsp:forward url="/monSiteWeb/erreur.jsp">

<%-- Redirection avec la balise redirect --%>
<c:redirect url="http://www.siteduzero.com"/>


<%-- Les attributs valables pour <c:url/> le sont aussi pour la
redirection.
Ici par exemple, l'utilisation de paramètres --%>
<c:redirect url="http://www.siteduzero.com">
  <c:param name="mascotte" value="zozor"/>
  <c:param name="langue" value="fr"/>
</c:redirect>

<%-- Redirigera vers --%>
http://www.siteduzero.com?mascotte=zozor&langue=fr
```

Imports

La balise **<c:import/>** est en quelque sorte un équivalent à **<jsp:include/>**, mais qui propose plus d'options, et pallie ainsi les manques de l'inclusion standard.

L'attribut **url** est le seul paramètre obligatoire lors de l'utilisation de cette balise, et il désigne logiquement le fichier à importer :

Code : JSP

```
<%-- Copie le contenu du fichier ciblé dans la page actuelle --%>
<c:import url="exemple.html"/>
```

Un des avantages majeurs de la fonction d'import est qu'elle permet d'utiliser une variable pour stocker le flux récupéré, et ne propose pas simplement de l'inclure dans votre JSP comme c'est le cas avec **<jsp:include/>**. Cette fonctionnalité est importante, puisqu'elle permet d'effectuer des traitements sur les pages importées. L'attribut utilisé pour ce faire est nommé **varReader**. Nous reverrons cela en détails lorsque nous découvrirons la bibliothèque xml de la JSTL, où ce système prend toute son importance lorsqu'il s'agit de lire et de parser des fichiers XML :

Code : JSP

```
<%-- Copie le contenu d'un fichier xml dans une variable
(fileReader),
puis parse le flux récupéré dans une autre variable (doc). --%>
<c:import url="test.xml" varReader="fileReader">
  <x:parse var="doc" doc="${fileReader}" />
</c:import>
```

Ne vous inquiétez pas si la ligne **<x:parse** var="doc" doc="${fileReader}" **/>** vous est inconnue, c'est une balise de la bibliothèque xml de la JSTL, que je vous présenterai dans le chapitre suivant. Vous pouvez cependant retenir l'utilisation du **<c:import/>** pour récupérer un flux xml.

Note : le contenu du **varReader** utilisé, autrement dit la variable dans laquelle est stockée le contenu de votre fichier, n'est accessible qu'à l'intérieur du corps de la balise d'import, entre les tags **<c:import>** et **</c:import>**. Il n'est par conséquent pas possible de s'en servir en dehors. Dans le chapitre portant sur la bibliothèque xml, nous découvrirons un autre moyen, permettant de pouvoir travailler sur le contenu du fichier en dehors de l'import, au travers d'une variable de *scope*.

De la même manière que la redirection en comparaison avec le *forwarding*, **<c:import/>** permet d'inclure des pages extérieures au contexte de votre servlet ou à votre serveur, contrairement à l'action standard JSP. Voyons une nouvelle fois quelques exemples d'utilisation :

Code : JSP

```
<%-- Inclusion d'une page avec l'action standard JSP. --%>
<jsp:include page="index.html" />

<%-- Importer une page distante dans une variable
Le scope par défaut est ici page si non précisé. --%>
<c:import url="http://www.siteduzero.com/zozor/biographie.html"
var="bio" scope="page"/>

<%-- Les attributs valables pour <c:url/> le sont aussi pour la
redirection.
Ici par exemple, l'utilisation de paramètres --%>
<c:import url="footer.jsp">
  <c:param name="design" value="bleu"/>
</c:import>
```

Au terme de ce chapitre, vous devez être capable de transformer des scriptlets Java contenant variables, boucles et conditions en une jolie page JSP basée sur des tags JSTL.
Testez maintenant vos connaissances dans le TP qui suit ce chapitre !

Sachez avant de continuer que d'autres bibliothèques de base existent, la JSTL ne contenant en réalité pas une bibliothèque mais cinq ! Voici une brève description des quatre autres :

- **fmt** : destinée au formatage et au parsage des données. Permet notamment la localisation de l'affichage.
- **fn** : destinée au traitement de chaînes de caractères.
- **sql** : destinée à l'interaction avec une base de données. Celle-ci ne trouve pour moi son sens que dans une petite application *standalone*, ou une feuille de tests. En effet, le code ayant trait au stockage des données dans une application web Java EE suivant le modèle MVC doit être masqué de la vue, et être encapsulé dans le modèle, éventuellement dans une couche dédiée (voir le modèle de conception DAO pour plus d'information à ce sujet). Bref, je vous laisse parcourir par vous-mêmes les liens de documentation si vous souhaitez en faire usage dans vos projets. En ce qui nous concerne, nous suivons MVC à la lettre et je ne souhaite clairement pas vous voir toucher aux données de la base directement depuis vos pages JSP... Une fois n'est pas coutume, le premier que je vois coder ainsi, je le pends à un arbre !
- **xml** : destinée au traitement de fichiers et données XML. À l'instar de la bibliothèque **sql**, celle-ci trouve difficilement sa place dans une application MVC, ces traitements ayant bien souvent leur place dans des objets du modèle, et pas dans la vue. Cela dit, dans certains cas elle peut s'avérer très utile, ce format étant très répandu dans les applications et communications web : c'est pour cela que j'ai décidé d'en faire l'objet du prochain chapitre !

JSTL core : exercice d'application

La bibliothèque *Core* n'a maintenant plus de secrets pour vous. Mais si vous souhaitez vous familiariser avec toutes ces nouvelles balises et être à l'aise lors du développement de pages JSP, vous devez vous entrainer !

Je vous propose ici un petit exercice d'application qui met en jeu des concepts réalisables à l'aide des balises que vous venez de découvrir. Suivez le guide...

Les bases de l'exercice

Pour mener à bien ce petit exercice, commencez par créer un nouveau projet web nommé **jstl_exo1**. Référez-vous au premier chapitre de cette partie si vous avez encore des hésitations sur la démarche nécessaire. Configurez bien entendu ce projet en y intégrant la JSTL, afin de pouvoir utiliser nos chères balises dans nos pages JSP !

Une fois ceci fait, créez une première page JSP à la racine de votre application, sous le répertoire **WebContent**. Je vous en donne ici le contenu complet :

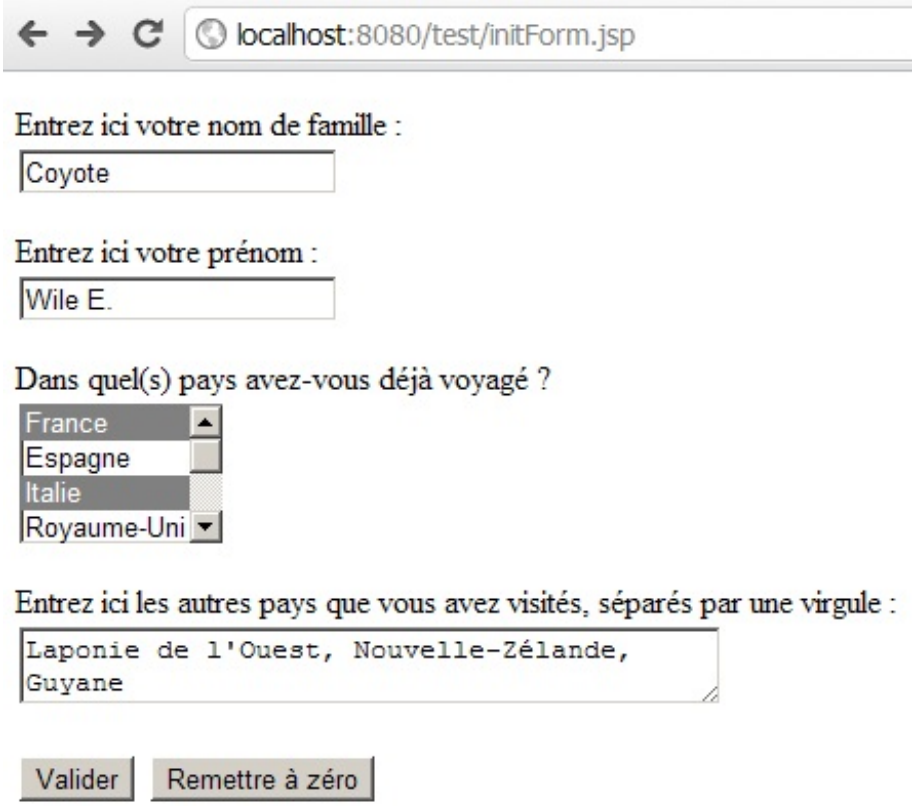
Code : JSP - initForm.jsp

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>
      Initialisation des données
    </title>
  </head>
  <body>
    <form method="post" action="initProcess.jsp">
      <p>
        <label for="nom">Entrez ici votre nom de famille
:</label><br />
        <input type="text" name="nom" id="nom" tabindex="10" />
      </p>
      <p>
        <label for="prenom">Entrez ici votre prénom :</label><br />
        <input type="text" name="prenom" id="prenom" tabindex="20"
/>
      </p>
      <p>
        <label for="pays">Dans quel(s) pays avez-vous déjà voyagé
?</label><br />
        <select name="pays" id="pays" multiple="multiple"
tabindex="30">
          <option value="France">France</option>
          <option value="Espagne">Espagne</option>
          <option value="Italie">Italie</option>
          <option value="Royaume-uni">Royaume-Uni</option>
          <option value="Canada">Canada</option>
          <option value="Etats-unis">Etats-Unis</option>
          <option value="Chine">Chine</option>
          <option value="Japon">Japon</option>
        </select>
      </p>
      <p>
        <label for="autre">Entrez ici les autres pays que vous avez
visités, séparés par une virgule :</label><br />
        <textarea id="autre" name="autre" rows="2" cols="40"
tabindex="40" palceholder="Ex: Norvège, Chili, Nouvelle-
Zélande"></textarea>
      </p>
      <p>
        <input type="submit" value="Valider" /> <input type="reset"
value="Remettre à zéro" />
      </p>
    </form>
  </body>
</html>
```

Récapitulons rapidement la fonction de cette page :

- permettre à l'utilisateur de saisir son nom ;
- permettre à l'utilisateur de saisir son prénom ;
- permettre à l'utilisateur de choisir les pays qu'il a visités parmi une liste de choix par défaut ;
- permettre à l'utilisateur de saisir d'autres pays qu'il a visité, en les séparant par une virgule.

Voici le rendu, rempli avec des données de test :



Votre mission maintenant, c'est d'écrire la page **initProcess.jsp** qui va se charger de traiter les données saisies dans la page contenant le formulaire.

Nous n'avons pas encore étudié le traitement des formulaires en Java EE, mais ne paniquez pas. Ce que je vous demande de créer ici a pour unique objectif de vous faire manipuler la JSTL. En effet, vous savez que les données envoyées par le client sont accessibles à travers les paramètres de requêtes, avec la JSTL et les EL vous avez tout en main pour mettre en place ce petit exercice !

Ne vous inquiétez pas, nous apprendrons dans la partie suivante de ce cours comment gérer proprement les formulaires dans une application Java EE.

Le sujet est ici volontairement simple, et son utilité nulle. L'objectif est purement didactique ici, l'intérêt est de vous familiariser avec le développement de pages et la manipulations de données en utilisant la JSTL. Ne vous intéressez par conséquent pas aux détails de cette première page **initForm.jsp**, elle n'est là que pour servir de base à notre exercice.

Pour en revenir à l'exercice, je ne vous demande ici rien de bien compliqué. La page devra simplement afficher :

1. une liste récapitulant le nom de chaque champ du formulaire et les informations qui y ont été saisies ;
2. les nom et prénom saisis par l'utilisateur ;
3. une liste des pays visités par l'utilisateur.

Il y a plusieurs manières de réaliser ces tâches basiques, choisissez celle qui vous semble la plus simple et logique.

Prenez le temps de chercher et de réfléchir, et on se retrouve ensuite pour la correction !

Correction

Ne vous jetez pas sur la correction sans chercher par vous-mêmes : cet exercice n'aurait alors plus aucun intérêt. Pour ceux d'entre vous qui peinent à voir par où partir, ou comment agencer tout cela, voilà en exemple le squelette de la page que j'ai réalisée, contenant seulement les commentaires expliquant les traitements à effectuer :

Secret (cliquez pour afficher)

Code : JSP

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

<!DOCTYPE html>
<html>
  <head>
     <meta charset="utf-8" />
     <title>Traitement des données</title>
  </head>
  <body>
      <p>
         <b>Vous avez renseigné les informations suivantes :</b>
      </p>

      <%-- Parcourt chaque paramètre de la requête --%>
            <%-- Affiche le nom de chaque paramètre. --%>

            <%-- Parcourt la liste des valeurs de chaque
paramètre. --%>
               <%-- Affiche chacune des valeurs --%>

   <p>
     <b>Vous vous nommez :</b>
   </p>
   <p>
     <%-- Affiche les valeurs des paramètres nom et prenom --%>
   </p>

   <p>
     <b>Vous avez visité les pays suivants :</b>
   </p>
   <p>
     <%-- Teste l'existence du parametre pays. S'il existe on le
traite,
     si non on affiche un message par défaut.--%>

     <%-- Teste l'existence du parametre autre. Si des données
existent on les traite,
     si non on affiche un message par défaut.--%>
      </p>
   </body>
</html>
```

Si vous étiez perdus, avec cette ébauche vous devriez avoir une meilleure idée de ce que j'attends de vous. Prenez votre temps, et n'hésitez pas à relire les chapitres précédents pour vérifier les points qui vous semblent flous !

Voici finalement la page que j'ai écrite. Comme je vous l'ai signalé plus tôt, ce n'est pas LA solution, c'est simplement une des manières de réaliser ce simple traitement :

Secret (cliquez pour afficher)

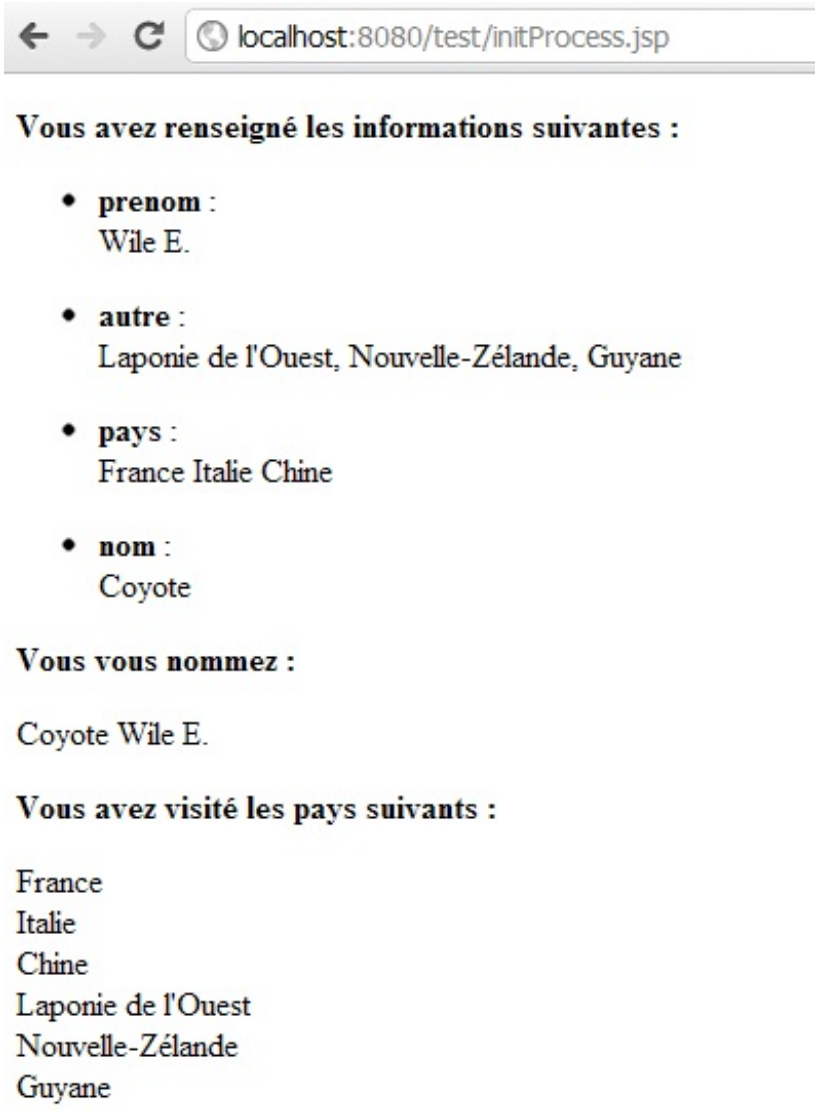
Code : JSP - initProcess.jsp

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

```
<!DOCTYPE html>
<html>
  <head>
     <meta charset="utf-8" />
     <title>Traitement des données</title>
  </head>
  <body>
      <p>
         <b>Vous avez renseigné les informations suivantes :</b>
      </p>

      <%-- Parcourt l'objet implicite paramValues, qui
souvenez-vous est une Map,
      pour traiter chaque paramètre de la requête --%>
      <c:forEach var="parametre" items="${ paramValues }">
         <ul>
            <%-- Affiche la clé de la Map paramValues,
            qui correspond concrètement au nom du paramètre. --%>
            <li><b><c:out value="${ parametre.key }"/></b> :</li>

            <%-- Parcourt le tableau de String[] associé à la clé
traitée,
            qui correspond à la liste de ses valeurs. --%>
            <c:forEach var="value" items="${ parametre.value }">
               <%-- Affiche chacune des valeurs pour la clé donnée
--%>
               <c:out value="${ value }"/>
            </c:forEach>
         </ul>
      </c:forEach>

   <p>
     <b>Vous vous nommez :</b>
   </p>
   <p>
     <%-- Affiche les valeurs des paramètres nom et prenom en
  y accédant directement via l'objet implicite (une Map) param.
  On sait en effet qu'il n'y a qu'une valeur associée à chacun de
ces 2 paramètres,
  pas besoin d'utiliser paramValues ! --%>
       <c:out value="${ param.nom }"/> <c:out value="${
param.prenom }"/>
   </p>

   <p>
     <b>Vous avez visité les pays suivants :</b>
   </p>
   <p>
     <%-- Teste l'existence du parametre pays. S'il existe on le
traite,
     si non on affiche un message par défaut.--%>
  <c:choose>
   <c:when test="${ !empty paramValues.pays }">
   <%-- Parcourt le tableau de valeurs associées au paramètre pays
de la requête,
      en utilisant l'objet implicite paramValues. En effet, c'est
nécessaire ici puisque
      le select permet de renvoyer plusieurs valeurs pour le seul
paramètre nommé pays. --%>
   <c:forEach var="pays" items="${ paramValues.pays }">
    <c:out value="${ pays }"/><br/>
   </c:forEach>
   </c:when>
   <c:otherwise>
    Vous n' avez pas visité de pays parmi la liste proposée.<br/>
   </c:otherwise>
  </c:choose>
  <%-- Teste l'existence du parametre autre. Si des données
existent on les traite,
     si non on affiche un message par défaut.--%>
```

```
        <c:choose>
         <c:when test="${ !empty param.autre }">
         <%-- Parcourt les valeurs associées au paramètre autre de la
requête,
              en utilisant l'objet implicite param. En effet, toutes les
valeurs sont ici concaténées
              et transmises dans une seule chaîne de caractères, qu'on
parcourt via la boucle forTokens ! --%>
         <c:forTokens var="pays" items="${ param.autre }" delims=",">
          <c:out value="${ pays }"/><br/>
         </c:forTokens>
         </c:when>
         <c:otherwise>
          Vous n' avez pas visité d'autre pays.<br/>
         </c:otherwise>
        </c:choose>
            </p>
         </body>
      </html>
```

Et voici le rendu avec les données de test :



J'ai utilisé ici des tests conditionnels et différentes boucles afin de vous faire pratiquer, et mis en jeu différents objets implicites. J'aurais par exemple très bien pu mettre en jeu des variables de *scope* pour stocker les informations récupérées depuis la requête. Si vous n'êtes pas parvenus à réaliser ce simple traitement de formulaire, vous devez identifier les points qui vous ont posé problème et revoir le cours plus attentivement !

Si vous avez du mal avec l'utilisation des Map et objets implicites, et peinez à comprendre cette correction, n'hésitez pas à m'envoyer un MP expliquant ce que vous n'avez pas bien assimilé. Je tâcherai alors de compléter ce chapitre pour que tout le monde puisse être à l'aise et autonome avec ce genre de manipulations.

Je n'irai pour le moment pas plus loin dans la pratique. De nombreuses balises ne sont pas intervenues dans cet exercice : ne vous inquiétez pas, vous aurez bien assez tôt l'occasion d'appliquer de manière plus exhaustive ce que vous avez découvert, dans les prochaines parties du cours. Soyez patients !

En attendant, n'hésitez pas à travailler davantage, à tenter de développer d'autres fonctionnalités de votre choix. Vous serez alors prêts pour étudier la bibliothèque xml de la JSTL, que je vous présente dans le chapitre suivant !

La bibliothèque xml

Nous allons ici parcourir les fonctions principales de la bibliothèque **xml**. Les flux ou fichiers XML sont très souvent utilisés dans les applications web, et la JSTL offrant ici un outil très simple d'utilisation pour effectuer quelques actions de base sur ce type de format, il serait bête de s'en priver. Toutefois, n'oubliez pas mon avertissement dans la conclusion du chapitre sur la bibliothèque **Core** : seuls certains cas particuliers justifient l'utilisation de la bibliothèque **xml**, dans la plupart des applications MVC ces actions ont leur place dans le modèle, et pas dans la vue !

Petite remarque avant de commencer : le fonctionnement de certaines balises étant très similaire à celui de balises que nous avons déjà abordées dans le chapitre précédent sur la bibliothèque **Core**, ce chapitre sera par moment un peu plus expéditif.

La syntaxe XPath

Pour vous permettre de comprendre simplement les notations que j'utiliserai dans les exemples de ce chapitre, je dois d'abord vous présenter le langage **XML Path Language**, ou **XPath**. Autant vous prévenir tout de suite, de la même manière que pour les EL lors du chapitre présentant la syntaxe JSP, je ne vous présenterai ici que succinctement les bases dont j'ai besoin. Un tuto à part entière serait nécessaire afin de faire le tour complet des possibilités offertes par ce langage, et ce n'est pas notre objectif ici. Encore une fois, si vous êtes curieux, les documentations et ressources ne manquent pas sur le web à ce sujet !

Le langage XPath

Le langage XPath permet d'identifier les nœuds dans un document XML. Il fournit une syntaxe permettant de cibler directement un fragment du document traité, comme un ensemble de nœuds ou encore un attribut d'un nœud en particulier, de manière relativement simple. Comme son nom le suggère, "path" signifiant chemin en anglais, la syntaxe de ce langage ressemble aux chemins d'accès aux fichiers dans un système : les éléments d'une expression XPath sont en effet séparés par des slashes '/'.

Je n'introduis ici que les notions qui me seront nécessaires dans la suite de ce cours. Pour des notions exhaustives, dirigez-vous vers la page du w3c, ou vers un tuto dédié à ce sujet.

Structure XML

Voici la structure de notre fichier XML de test, que j'utiliserai dans les quelques exemples illustrant ce paragraphe :

Code : XML - monDocument.xml

```xml
<news>
  <article id="1">
    <auteur>Pierre</auteur>
    <titre>Foo...</titre>
    <contenu>...bar !</contenu>
  </article>
  <article id="27">
    <auteur>Paul</auteur>
    <titre>Bientôt un LdZ J2EE !</titre>
    <contenu>Woot ?</contenu>
  </article>
  <article id="102">
    <auteur>Jacques</auteur>
    <titre>Coyote court toujours</titre>
    <contenu>Bip bip !</contenu>
  </article>
</news>
```

Plutôt que de paraphraser, voyons directement comment sélectionner diverses portions de ce document via des exemples commentés :

Code : XML

```
<!-- Sélection du nœud racine -->
/

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

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

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

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

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

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

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

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

<!-- Sélection des nœuds 'article' enfants de 'news' dont l'attribut
id vaut '12' -->
/news/article[@id='12']
```

Je m'arrêterai là pour les présentations. Sachez qu'il existe des commandes plus poussées que ces quelques éléments, et je vous laisse le loisir de vous plonger dans les ressources que je vous ai communiquées pour plus d'information. Mon objectif ici est simplement de vous donner un premier aperçu de ce qu'est la syntaxe XPath, afin que vous compreniez sa logique et ne soyez pas perturbés lorsque vous me verrez utiliser cette syntaxe dans les attributs de certaines balises de la bibliothèque xml.

J'imagine que cela reste assez flou dans votre esprit, et que vous vous demandez probablement comment diable ces expressions vont-elles bien pouvoir nous servir, et surtout où nous allons pouvoir les utiliser. Pas d'inquiétude, les explications vous seront fournies au fur et à mesure que vous découvrirez les balises mettant en jeu ce type d'expressions.

Pour ceux d'entre vous qui veulent tester les expressions XPath précédentes, ou qui veulent pratiquer en manipulant d'autres fichiers XML et/ou en mettant en jeu d'autres expressions XPath, voici un site web qui vous permettra de tester en direct le résultat de vos expressions sur le document XML de votre choix : XPath Expression Testbed. Amusez-vous et vérifiez ainsi votre bonne compréhension du langage !

Les actions de base

Avant de vous présenter les différentes balises disponibles, je vous donne ici la directive JSP nécessaire pour permettre l'utilisation des balises de la bibliothèque xml dans vos pages :

Code : JSP

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

Retenez bien que cette directive devra être présente sur chacune des pages de votre projet utilisant les balises JSTL que je vous présente dans ce chapitre. Dans un prochain chapitre concernant la création d'une bibliothèque personnalisée, nous verrons comment il est possible de ne plus avoir à se soucier de cette commande. En attendant, ne l'oubliez pas !

Récupérer et analyser un document

Je vais dans ce paragraphe vous montrer comment procéder pour récupérer et analyser simplement un fichier XML depuis votre page JSP. Je reprends le fichier XML que j'ai utilisé précédemment lorsque je vous ai présenté la syntaxe du langage XPath, et je

le nomme `monDocument.xml`.

Commençons par aborder la récupération du fichier XML. Cette étape correspond simplement à un import, réalisé avec ce tag de la bibliothèque *Core* que vous devez déjà connaître :

Code : JSP

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

Remarquez l'utilisation de l'attribut **varReader**. C'est en quelque sorte un buffer, une variable qui sera utilisée pour une utilisation postérieure du contenu du fichier importé. Notez que lorsque vous utiliserez cet attribut, il vous sera impossible d'utiliser conjointement l'attribut **var**.

Rappelez-vous, lorsque l'on utilise cet attribut **varReader**, le contenu du fichier importé n'est pas inclus littéralement dans votre page JSP comme c'est le cas lors d'un import simple : il est copié dans la variable nommée dans l'attribut **varReader**.

Le document XML étant récupéré et stocké dans la variable *monReader*, nous souhaitons maintenant l'analyser. Nous allons pour cela faire intervenir une nouvelle balise, issue de la librairie xml cette fois :

Code : JSP

```
<c:import url="monDocument.xml" varReader="monReader">
   <%-- Parse le contenu du fichier XML monDocument.xml dans une
variable nommée 'doc' --%>
<x:parse var="doc" doc="${monReader}" />
    ...
</c:import>
```

Deux attributs sont ici utilisés :

- **var** : contient le nom de la variable de *scope* qui contiendra les données qui représentent notre document XML parsé. Comme d'habitude, si l'attribut **scope** n'est pas explicité, la portée par défaut de cette variable sera la page.
- **doc** : permet de préciser que l'on souhaite parser le contenu de notre **varReader** défini précédemment lors de l'import. Souvenez-vous, le **varReader** ici nommé *monReader* est une variable, il nous faut donc utiliser une EL pour y faire référence, en l'occurrence **${monReader}** !

Dans certains codes vieillissant, vous trouverez parfois dans l'utilisation de la balise **<x:parse/>** un attribut nommé **xml**. Sachez qu'il joue le même rôle que l'attribut **doc**, et qu'il est déprécié : concrètement, il a été remplacé par **doc**, et il ne faut donc plus l'utiliser.

Note : l'import qui stocke le fichier dans le **varReader** doit rester ouvert pour pouvoir appliquer un **<x:parse/>** sur le contenu de ce **varReader** ! La portée du **varReader** défini est en effet uniquement l'intérieur du corps du **<c:import/>**. Afin de pouvoir accéder à ce **varReader**, il ne faut donc pas fermer directement la balise d'import comme c'est le cas ci-dessous :

Code : JSP

```
<%-- Mauvaise utilisation du varReader --%>
<c:import url="monDocument.xml" varReader="monReader" />
```

Toutefois, il est possible de ne pas utiliser le **varReader**, et de simplement utiliser une variable de *scope*. Vous pourrez ainsi faire votre import, puis traiter le contenu du fichier par la suite, sans devoir travailler dans le corps de la balise d'import :

Code : JSP

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

Cela dit, je vous conseille de travailler avec le **varReader**, puisque c'est l'objectif premier de cet attribut.

Plusieurs remarques sont d'ores et déjà nécessaires :

- Comprenez bien ici la différence entre le **varReader** de la balise **<c:import/>** et le **var** de la balise **<x:parse/>**: le premier contient le contenu brut du fichier XML, alors que le second contient le résultat du parsing du fichier XML. Pour faire simple, la JSTL utilise une structure de données qui représente notre document XML parsé, et c'est cette structure qui est stockée dans la variable définie par **var**.

- Le type de la variable définie via cet attribut **var** dépendra de l'implémentation choisie par le développeur. Pour information, il est ici possible de remplacer l'attribut **var** par l'attribut nommé **varDom**, qui permet de fixer l'implémentation utilisée : la variable ainsi définie sera de type `org.w3c.dom.Document`. De même, **scope** sera remplacé par **scopeDom**. Ceci impose donc que votre fichier XML respecte l'interface `Document` citée précédemment. Tout cela étant vraiment spécifique, je ne m'étalerai pas davantage sur le sujet et je vous renvoie à la documentation pour plus d'infos.

- Importer un fichier n'est pas nécessaire. Il est en effet possible de traiter directement un flux XML depuis la page JSP, en le plaçant dans le corps de la balise **<x:parse/>** :

Code : JSP

```
<%-- Parse le flux XML contenu dans le corps de la balise --%>
<x:parse var="doc">
 <news>
   <article id="1">
  <auteur>Pierre</auteur>
  <titre>Foo...</titre>
  <contenu>...bar !</contenu>
   </article>
   <article id="27">
  <auteur>Paul</auteur>
  <titre>Bientôt un LdZ J2EE !</titre>
  <contenu>Woot ?</contenu>
   </article>
   <article id="102">
  <auteur>Jacques</auteur>
  <titre>Coyote court toujours</titre>
  <contenu>Bip bip !</contenu>
   </article>
 </news>
</x:parse>
```

Il reste seulement deux attributs que je n'ai pas encore abordés :

- **filter** : permet de limiter le contenu traité par l'action de parsing **<x:parse/>** à une portion d'un flux XML seulement. Cet attribut peut s'avérer utile lors de l'analyse de documents XML lourds, afin de ne pas détériorer les performances à l'exécution de votre page. Pour plus d'information sur ces filtres de type `XMLFilter`, essayez la documentation.
- **systemId** : cet attribut ne vous sera utile que si votre fichier XML contient des références vers des entités externes. Vous devez y saisir l'adresse URI qui permettra de résoudre les liens relatifs contenus dans votre fichier XML. *Bref rappel : une référence à une entité externe dans un fichier XML est utilisée pour y inclure un fichier externe, principalement lorsque des données ou textes sont trop longs et qu'il est plus simple de les garder dans un fichier à part.* Le processus accèdera à ces fichiers externes lors du parsage du document XML spécifié.

Je n'ai pour ces derniers pas d'exemple trivial à vous proposer. Je fais donc volontairement l'impasse ici, je pense que ceux parmi

vous qui connaissent et ont déjà manipulé les filtres XML et les entités externes comprendront aisément de quoi il retourne.

Afficher une expression

Les noms des balises que nous allons maintenant aborder devraient vous être familiers : ils trouvent leur équivalent dans la bibliothèque *Core* que vous avez découverte dans le chapitre précédent. Alors que ces dernières accédaient à des données de l'application en utilisant des EL, les balises de la bibliothèque xml vont quant à elles accéder à des données issues de documents XML, via des expressions XPath.

Pour afficher un élément, nous allons utiliser la balise **<x:out/>**, pour laquelle seul l'attribut **select** est nécessaire :

Code : JSP

```
<c:import url="monDocument.xml" varReader="monReader">
   <%-- Parse le contenu du fichier XML monDocument.xml dans une
variable nommée 'doc' --%>
   <x:parse var="doc" doc="${monReader}" />
<x:out select="$doc/news/article/auteur" />
</c:import>
```

Le rendu HTML du code ci-dessus est alors le suivant :

Code : HTML

```
Pierre
```

En suivant le paragraphe introduisant XPath, j'avais compris qu'une telle expression renvoyait tous les nœuds "auteur" du document !
Où sont passés Paul et Jacques ? Où est l'erreur ?

Héhé... Eh bien, à vrai dire il n'y a aucune erreur ! En effet, l'expression XPath renvoie bel et bien un ensemble de nœuds, en l'occurrence les nœuds "auteur" ; cet ensemble de noeuds est stocké dans une structure de type `NodeSet`, un type propre à XPath qui implémente le type Java standard `NodeList`.

Le comportement ici observé provient du fait que la balise d'affichage **<x:out/>** ne gère pas réellement un ensemble de nœuds, et n'affiche que le premier nœud contenu dans cet ensemble de type `NodeSet`. Toutefois, le contenu de l'attribut **select** peut très bien contenir un `NodeSet` ou une opération sur un `NodeSet`. Vérifions par exemple que `NodeSet` contient bien 3 nœuds, puisque nous avons 3 auteurs dans notre document XML :

Code : JSP

```
<c:import url="monDocument.xml" varReader="monReader">
   <%-- Parse le contenu du fichier XML monDocument.xml dans une
variable nommée 'doc' --%>
   <x:parse var="doc" doc="${monReader}" />
<x:out select="count($doc/news/article/auteur)" />
</c:import>
```

J'utilise ici la fonction **count()**, qui renvoie le nombre d'éléments que l'expression XPath a sélectionnés et stockés dans le `NodeSet`. Et le rendu HTML de cet exemple est bien "3" ; notre ensemble de nœuds contient donc bien trois auteurs, Paul et Jacques ne sont pas perdus en cours de route.

L'attribut **select** de la balise **<x:out/>** est l'équivalent de l'attribut **value** de la balise **<c:out/>**, sauf qu'il attend ici une expression XPath et non plus une EL ! Rappelez-vous que le rôle des expressions XPath est de sélectionner des portions de document XML. Expliquons rapidement l'expression **<x:out** `select="$doc/news/article/auteur"` **/>** : elle va sélectionner tous les nœuds "auteur" qui sont enfants d'un nœud "article" lui-même enfant du nœud racine "news" présent dans

le document $doc. En l'occurrence, $doc se réfère ici au conteu parsé de notre variable **varReader**.

Dans une expression XPath, pour faire référence à une variable nommée *nomVar* on n'utilise pas `${nomVar}` comme c'est le cas dans une EL, mais `$nomVar`. Essayez de retenir cette syntaxe, cela vous évitera bien des erreurs ou comportements inattendus !

À ce sujet, sachez enfin qu'outre une variable simple, il est possible de faire intervenir les objets implicites dans une expression XPath, de cette manière :

Code : JSP

```
<%-- Récupère le document nommé 'doc' enregistré auparavant en
session, via l'objet implicite sessionScope  --%>
<x:out select="$sessionScope:doc/news/article" />

<%-- Sélectionne le nœud 'article' dont l'attribut 'id' a pour
valeur le contenu de la variable
 nommée 'idArticle' qui a été passée en paramètre de la requête, via
l'objet implicite param  --%>
<x:out select="$doc/news/article[@id=$param:idArticle]"/>
```

Ce qu'on peut retenir de cette balise d'affichage, c'est qu'elle fournit, grâce à un fonctionnement basé sur des expressions XPath, une alternative aux feuilles de style XSL pour la transformation de contenus XML, en particulier lorsque le format d'affichage final est une page web html.

Créer une variable

Nous passerons très rapidement sur cette balise. Sa syntaxe est **<x:set/>**, et comme vous vous en doutez elle est l'équivalent de la balise **<c:set/>** de la bibliothèque *Core*, avec de petites différences :

- l'attribut **select** remplace l'attribut **value**, ce qui a la même conséquence que pour la balise d'affichage : une expression XPath est attendue, et non pas une EL ;
- l'attribut **var** est obligatoire, ce qui n'était pas le cas pour la balise **<c:set/>**.

Ci-dessous un bref exemple de son utilisation :

Code : JSP

```
<%-- Enregistre le résultat de l'expression XPath, spécifiée dans
l'attribut select,
dans une variable de session nommée 'auteur' --%>
<x:set var="auteur" scope="session" select="$doc//auteur" />

<%-- Affiche le contenu de la variable nommée 'auteur' enregistrée
en session --%>
<x:out select="$sessionScope:auteur" />
```

Le rôle de la balise est donc sensiblement le même que son homologue de la bibliothèque *Core* : enregistrer le résultat d'une expression dans une variable de *scope*. La seule différence réside dans la nature de l'expression évaluée, qui est ici une expression XPath et non plus une EL.

Les conditions

Les conditions

Les balises permettant la mise en place de conditions sont là aussi sensiblement identiques à leurs homologues de la bibliothèque *Core* : la seule et unique différence réside dans le changement de l'attribut **test** pour l'attribut **select**. Par conséquent, comme vous le savez maintenant c'est ici une expression XPath qui est attendue, et non plus une EL !

Plutôt que de paraphraser le précédent chapitre, je ne vous donne ici que de simples exemples commentés, qui vous permettront

de repérer les quelques différences de syntaxe.

Une condition simple

Code : JSP

```
<%-- Afficher le titre de la news postée par 'Paul' --%>
<x:if select="$doc/news/article[auteur='Paul']">
   Paul a déjà posté une news dont voici le titre :
   <x:out select="$doc/news/article[auteur='Paul']/titre" />
</x:if>
```

Le rendu HTML correspondant :

Code : JSP

```
Paul a déjà posté une news dont voici le titre : Bientôt un LdZ J2EE
!
```

De même que pour la balise **<c:if/>**, il est possible de stocker le résultat du test conditionnel en spécifiant un attribut **var**.

Des conditions multiples

Code : JSP

```
<%-- Affiche le titre de la news postée par 'Nicolas' si elle
existe, et un simple message sinon --%>
<x:choose>
   <x:when select="$doc/news/article[auteur='Nicolas']">
      Nicolas a déjà posté une news dont voici le titre :
      <x:out select="$doc/news/article[auteur='Nicolas']/titre" />
   </x:when>
   <x:otherwise>
      Nicolas n'a pas posté de news.
   </x:otherwise>
</x:choose>
```

Le rendu HTML correspondant :

Code : JSP

```
Nicolas n'a pas posté de news.
```

Les contraintes d'utilisation de ces balises sont les mêmes que celles de la bibliothèques *Core*. Je vous renvoie au chapitre précédent si vous ne vous en souvenez plus.

Voilà tout pour les tests conditionnels de la bibliothèque xml : leur utilisation est semblable à celle des conditions de la bibliothèque *Core*, seule la cible change : on traite ici un flux XML, via des expressions XPath.

Les boucles

Les boucles

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

Code : JSP

```
<!-- Affiche les auteurs et titres de tous les articles -->
<p>
<x:forEach var="element" select="$doc/news/article">
   <strong><x:out select="$element/auteur" /></strong> :
   <x:out select="$element/titre" />.<br/>
</x:forEach>
</p>
```

Le rendu HTML correspondant :

Code : JSP

```
<p>
   <strong>Pierre</strong> : Foo....<br/>
   <strong>Paul</strong> : Bientôt un LdZ J2EE !.<br/>
   <strong>Jacques</strong> : Coyote court toujours.<br/>
</p>
```

De même que pour la balise **<c:forEach/>**, il est possible de faire intervenir un pas de parcours via l'attribut **step**, de définir les index de début et de fin via les attributs **begin** et **end**, ou encore d'utiliser l'attribut **varStatus** pour accéder à l'état de chaque itération.

Les tranformations
Transformations

Le bibliothèque xml de la JSTL permet d'appliquer des transformations à un flux XML via une feuille de style XSL. Je ne reviendrai pas ici sur le langage et les méthodes à employer, si vous n'êtes pas familiers avec ce concept, je vous conseille de lire cette introduction à la mise en forme de documents XML avec XSLT.

La balise dédiée à cette tâche est **<x:tranform/>**. Commençons par un petit exemple, afin de comprendre comment elle fonctionne. J'utiliserai ici le même fichier XML que pour les exemples précédents, ainsi que la feuille de style XSL suivante :

Code : XML - test.xsl

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <html xmlns="http://www.w3.org/1999/xhtml">
      <head>
        <meta http-equiv="Content-Type" content="text/html;
charset=utf-8" />
        <title>Mise en forme avec XSLT</title>
      </head>
      <body>
        <table width="1000" border="1" cellspacing="0"
cellpadding="0">
          <tr>
            <th scope="col">Id</th>
            <th scope="col">Auteur</th>
            <th scope="col">Titre</th>
            <th scope="col">Contenu</th>
          </tr>
          <xsl:for-each select="/news/article">
            <tr>
              <td>
                <xsl:value-of select="@id" />
              </td>
              <td>
                <xsl:value-of select="auteur" />
              </td>
              <td>
                <xsl:value-of select="titre" />
```

```
                    </td>
                    <td>
                      <xsl:value-of select="contenu" />
                    </td>
                  </tr>
                </xsl:for-each>
              </table>
            </body>
          </html>
        </xsl:template>
      </xsl:stylesheet>
```

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

Code : JSP - testTransformXsl.jsp

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

<c:import varReader="xslFile" url="test.xsl">
<c:import varReader="xmlFile" url="monDocument.xml">
<x:transform doc="${xmlFile}" xslt="${xslFile}"/>
</c:import>
</c:import>
```

On importe ici simplement nos deux fichiers, puis on appelle la balise **<x:transform/>**. Deux attributs sont ici utilisés :

- **doc** : contient la référence au document XML sur lequel la transformation doit être appliquée. Attention, ici on parle bien du document XML d'origine, et pas d'un document analysé via **<x:parse/>**. On travaille bien directement sur le contenu XML. Il est d'ailleurs possible ici de ne pas utiliser d'import, en définissant directement le flux XML à traiter dans une variable de *scope*, voire directement dans le corps de la balise comme dans l'exemple suivant :

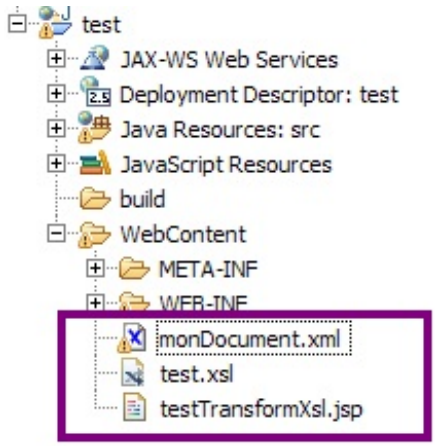
Code : JSP

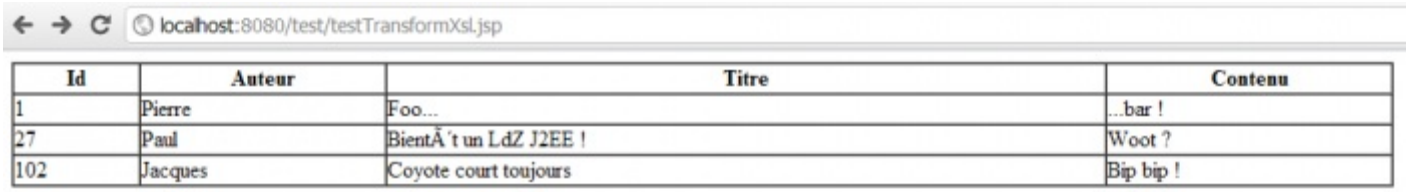
```
<x:transform xslt="${xslFile}">
   <news>
     <article id="1">
 <auteur>Pierre</auteur>
 <titre>Foo...</titre>
 <contenu>...bar !</contenu>
     </article>
     <article id="27">
 <auteur>Paul</auteur>
 <titre>Bientôt un LdZ J2EE !</titre>
 <contenu>Woot ?</contenu>
     </article>
     <article id="102">
 <auteur>Jacques</auteur>
 <titre>Coyote court toujours</titre>
 <contenu>Bip bip !</contenu>
     </article>
   </news>
</x:transform>
```

- **xslt** : contient logiquement la feuille de style XSL. Il est également possible de ne pas utiliser d'import ici, et de simplement définir une feuille de style dans une variable de *scope*.

En l'absence d'attribut **var**, le contenu transformé sera automatiquement généré dans la page HTML finale. Et lorsque vous accédez à cette page JSP depuis votre navigateur, vous apercevez un tableau contenant les données de votre fichier XML : la transformation a bien été appliquée ! Ceci est particulièrement intéressant lorsque vous souhaitez formater un contenu XML en

HTML, par exemple lors de la lecture de flux RSS. En images :





Id	Auteur	Titre	Contenu
1	Pierre	Foo...	...bar !
27	Paul	BientÃ´t un LdZ J2EE !	Woot ?
102	Jacques	Coyote court toujours	Bip bip !

Si par contre vous précisez un attribut **var**, le résultat de cette transformation sera alors stocké dans la variable de *scope* ainsi créée, de type `Document`. Sachez qu'il existe également un attribut **result** qui, en l'absence des attributs **var** et **scope**, stocke l'objet créé par la transformation.

Il est pour terminer possible de passer des paramètres à une transformation XSLT, en utilisant la balise **<x:param/>**. Cette dernière ne peut exister que dans le corps d'une balise **<x:transform/>**, et s'emploie de la même manière que son homologue de la bibliothèque *Core* :

Code : JSP - testTransformXsl.jsp

```
<c:import var="xslFile" url="test.xsl"/>
<c:import var="xmlFile" url="monDocument.xml"/>
<x:transform doc="${xmlFile}" xslt="${xslFile}">
<x:param name="couleur" value="orange" />
</x:transform>
```

Le comportement et l'utilisation sont identiques à ceux de **<c:param/>** : deux attributs **name** et **value** contiennent simplement le nom et la valeur du paramètre à transmettre. Ici dans cet exemple, ma feuille de style ne traite pas de paramètre, et donc ne fait rien de ce paramètre nommé **couleur** que je lui passe. Si vous souhaitez en savoir plus sur l'utilisation de paramètres dans une feuille XSL, vous savez où chercher !

Il reste deux attributs que je n'ai pas explicités : **docSystemId** and **xsltSystemId**. Ils ont tous deux la même utilité que l'attribut **systemId** de la balise **<x:parse/>**, et s'utilisent de la même façon : il suffit d'y renseigner l'URI destinée à résoudre les liens relatifs contenus respectivement dans le document XML et dans la feuille de style XSL.
Je n'ai pour le moment pas prévu de vous présenter les autres bibliothèques de la JSTL : je pense que vous êtes maintenant assez familiers avec la compréhension du fonctionnement des tags JSTL pour voler de vos propres ailes.

Mais ne partez pas si vite, prenez le temps de faire tous les tests que vous jugez nécessaires. Il n'y a que comme ça que ça rentrera, et que vous prendrez suffisamment de recul pour comprendre parfaitement ce que vous faites. Dans le chapitre suivant je vous propose un exercice d'application de ce que vous venez de découvrir, et ensuite on reprendra le code d'exemple de la partie précédente en y intégrant la JSTL !

JSTL xml : exercice d'application

La bibliothèque xml n'a maintenant plus de secrets pour vous. Mais si vous souhaitez vous familiariser avec toutes ces nouvelles balises et être à l'aise lors du développement de pages JSP, vous devez vous entrainer !

Je vous propose ici un petit exercice d'application qui met en jeu des concepts réalisables à l'aide des balises que vous venez de découvrir. Suivez le guide...

Les bases de l'exercice

On prend les mêmes et on recommence...

Pour mener à bien ce petit exercice, commencez par créer un nouveau projet nommé **jstl_exo2**. Configurez bien entendu ce projet en y intégrant la JSTL, afin de pouvoir utiliser nos chères balises dans nos pages JSP !

Une fois ceci fait, créez pour commencer un document XML. Je vous en donne ici le contenu complet :

Code : XML - inventaire.xml

```
<?xml version="1.0" encoding="utf-8"?>
<inventaire>
  <livre>
    <auteur>Pierre</auteur>
    <titre>D&#233;veloppez vos applications web avec JRuby !</titre>
    <date>Janvier 2012</date>
    <prix>22</prix>
    <stock>127</stock>
    <minimum>10</minimum>
  </livre>
  <livre>
    <auteur>Paul</auteur>
    <titre>D&#233;couvrez la puissance du langage Perl</titre>
    <date>Avril 2017</date>
    <prix>26</prix>
    <stock>74</stock>
    <minimum>10</minimum>
  </livre>
  <livre>
    <auteur>Matthieu</auteur>
    <titre>Apprenez &#224; programmer en C</titre>
    <date>Novembre 2009</date>
    <prix>25</prix>
    <stock>19</stock>
    <minimum>20</minimum>
  </livre>
  <livre>
    <auteur>Matthieu</auteur>
    <titre>Concevez votre site web avec PHP et MySQL</titre>
    <date>Mars 2010</date>
    <prix>30</prix>
    <stock>7</stock>
    <minimum>20</minimum>
  </livre>
  <livre>
    <auteur>Cysboy</auteur>
    <titre>La programmation en Java</titre>
    <date>Septembre 2010</date>
    <prix>29</prix>
    <stock>2000</stock>
    <minimum>20</minimum>
  </livre>
</inventaire>
```

Ne prêtez pas grande attention aux données modélisées par ce document. Nous avons simplement besoin d'une base simple, contenant de quoi nous amuser un peu avec les balises que nous venons de découvrir !

Note : toute ressemblance avec des personnages existant ou ayant existé serait fortuite et indépendante de la volonté de l'auteur...

Votre mission maintenant, c'est d'écrire la page **rapportInventaire.jsp** se chargeant d'analyser ce document XML, et de générer un rapport qui :

- listera chacun des livres présents ;
- affichera un message d'alerte pour chaque livre dont le stock est en dessous de la quantité minimum spécifiée ;
- listera enfin chacun des livres présents, regroupés par stocks triés du plus grand au plus faible.

Il y a plusieurs manières de réaliser ces tâches basiques, choisissez celle qui vous semble la plus simple et logique.
Prenez le temps de chercher et de réfléchir, et on se retrouve ensuite pour la correction !

Correction

Ne vous jetez pas sur la correction sans chercher par vous-mêmes : cet exercice n'aurait alors plus aucun intérêt. Je ne vous donne ici pas d'aide supplémentaire, si vous avez suivi le cours jusqu'ici vous devez être capables de comprendre comment faire, les balises nécessaires pour cet exercice ressemblant fortement à celles utilisées dans celui concernant la bibliothèque *Core* !

Voici donc une correction possible :

Secret (cliquez pour afficher)

Code : JSP

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

<!DOCTYPE html>
<html>
  <head>
     <meta charset="utf-8" />
     <title>Rapport d'inventaire</title>
  </head>
  <body>

<%-- Récupération du document XML. --%>
<c:import url="inventaire.xml" var="documentXML" />
<%-- Analyse du document XML récupéré. --%>
<x:parse var="doc" doc="${documentXML}" />

<p><b>Liste de tous les livres :</b></p>
<div>
<ul>
<%-- Parcours du document parsé pour y récupérer chaque noeud
"livre". --%>
<x:forEach var="livre" select="$doc/livre">
 <%-- Affichage du titre du livre récupéré. --%>
    <li><x:out select="$livre/titre" /></li>
</x:forEach>
</ul>
</div>

<p><b>Liste des livres qu'il faut réapprovisionner :</b></p>
<div>
<ul>
<%-- Parcours du document parsé pour y récupérer chaque noeud
"livre"
 dont le contenu du noeud "stock" est inférieur au contenu du
noeud
 "minimum". --%>
<x:forEach var="livre" select="$doc/livre[stock < minimum]">
 <%-- Affichage des titres, stocks et minimaux du livre récupéré.
--%>
    <li><x:out select="$livre/titre" /> : <x:out
select="$livre/stock" /> livres en stock (limite avant alerte :
<x:out select="$livre/minimum" />)</li>
</x:forEach>
```

```
        </ul>
        </div>

        <p><b>Liste des livres classés par stock :</b></p>
        <%-- Il faut réfléchir... un peu ! --%>
        <pre>
        Le tri d'une liste, d'un tableau, d'une collection... bref de
        manière générale le tri de données,
        ne doit pas se faire depuis votre page JSP ! Que ce soit en
        utilisant les API relatives aux collections,
        ou via un bean de votre couche métier, ou que sais-je encore, il
        est toujours préférable que votre tri
        soit effectué avant d'arriver à votre JSP. La JSP ne doit en
        principe que récupérer cette collection déjà triée,
        formater les données pour une mise en page particulière si
        nécessaire, et seulement les afficher.

        C'était un simple piège ici, j'espère que vous avez réfléchi avant
        de tenter d'implémenter un tri avec
        la JSTL, et que vous comprenez pourquoi cela ne doit pas
        intervenir à ce niveau ;)
        </pre>

          </body>
        </html>
```

Je n'ai fait intervenir ici que des traitements faciles, n'utilisant que des boucles et des expressions XPath. J'aurais pu vous imposer l'utilisation de tests conditionnels, ou encore de variables de *scope*, mais l'objectif est ici uniquement de vous permettre d'être à l'aise avec l'analyse de document XML. Si vous n'êtes pas parvenus à réaliser ce simple traitement de document, vous devez identifier les points qui vous ont posé problème et revoir le cours plus attentivement !
N'hésitez pas à travailler davantage, à tenter de développer d'autres fonctionnalités de votre choix. Vous pouvez par exemple faire intervenir des transformations XSL sur le document XML.

Faisons le point !

Il est temps de mettre en pratique ce que nous avons appris. Nous avons en effet abordé toutes les balises et tous les concepts nécessaires et sommes maintenant capables de réécrire proprement nos premiers exemples en utilisant des tags JSTL ! Je vous propose ensuite, pour vous détendre un peu, quelques conseils autour de l'écriture de code Java en général.

Reprenons notre exemple

Dans la partie précédente, la mise en place de boucles et conditions était un obstacle que nous étions incapables de franchir sans écrire de code Java. Maintenant que nous avons découvert les balises de la bibliothèque *Core* de la JSTL, nous avons tout ce qu'il nous faut pour réussir.

Pour rappel, voici où nous en étions :

Code : JSP - État final de notre vue d'exemple en fin de partie précédente

```
<%@ page import="java.util.List" %>
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Test</title>
    </head>
    <body>
        <p>Ceci est une page générée depuis une JSP.</p>
        <p>
            ${test}
            ${auteur}
        </p>
        <p>
            Récupération du bean :
            ${coyote.prenom}
            ${coyote.nom}
        </p>
        <p>
            Récupération de la liste :
<%
List<Integer> liste = (List<Integer>) request.getAttribute( "liste"
);
for( Integer i : liste ){
out.println(i + " : ");
}
%>
        </p>
        <p>
            Récupération du jour du mois :
<%
Integer jourDuMois = (Integer) request.getAttribute( "jour" );
if ( jourDuMois % 2 == 0 ){
out.println("Jour pair : " + jourDuMois);
} else {
out.println("Jour impair : " + jourDuMois);
}
%>
        </p>
    </body>
</html>
```

Et voici les nouvelles balises qui vont nous permettre de faire disparaître le code Java de notre JSP d'exemple :

- **`<c:choose>`** pour la mise en place de conditions ;
- **`<c:foreach>`** pour la mise en place de boucles.

Reprise de la boucle

En utilisant la syntaxe JSTL, notre boucle devient simplement :

Code : JSP

```
<p>
    Récupération de la liste :
    <%-- Boucle sur l'attribut de la requête nommé 'liste' --%>
    <c:forEach items="${liste}" var="element">
        <c:out value="${element}" /> :
    </c:forEach>
</p>
```

Comme prévu, plus besoin de récupérer explicitement la variable contenant la liste depuis la requête, et plus besoin d'écrire du code Java en dur pour mettre en place la boucle sur la liste.

Reprise de la condition

En utilisant la syntaxe JSTL, notre condition devient simplement :

Code : JSP

```
<p>
    Récupération du jour du mois :
    <c:choose>
        <%-- Test de parité sur l'attribut de la requête nommé
'jour' --%>
        <c:when test="${ jour % 2 == 0 }">Jour pair :
${jour}</c:when>
        <c:otherwise>Jour impair : ${jour}</c:otherwise>
    </c:choose>
</p>
```

Comme prévu, plus besoin de récupérer explicitement la variable contenant le jour du mois depuis la requête, et plus besoin d'écrire du code Java en dur pour mettre en place le test de parité.

Ainsi, notre page finale est bien plus claire et compréhensible :

Code : JSP - Page d'exemple sans code Java

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Titre</title>
    </head>
    <body>
        <p>Ceci est une page générée depuis une JSP.</p>
        <p>
            ${test}
            ${auteur}
        </p>
        <p>
            Récupération du bean :
            ${coyote.prenom}
            ${coyote.nom}
        </p>
        <p>
            Récupération de la liste :
<c:forEach items="${liste}" var="element">
```

```
${element} :
</c:forEach>
 </p>
 <p>
            Récupération du jour du mois :
<c:choose>
<c:when test="${ jour % 2 == 0 }">Jour pair : ${jour}</c:when>
<c:otherwise>Jour impair : ${jour}</c:otherwise>
</c:choose>
 </p>
    </body>
</html>
```

Quelques conseils

Avant d'attaquer la suite du cours, détendez-vous un instant et découvrez ces quelques astuces pour mieux organiser votre code et le rendre plus lisible.

Utilisation de constantes

Afin de faciliter la lecture et la modification du code d'une classe, il est recommandé de ne pas écrire le contenu des attributs de types primitifs en dur au sein de votre code, et de le regrouper sous forme de constantes en début de classe afin d'y centraliser les données.

Reprenons par exemple notre servlet d'exemple, dans laquelle j'ai surligné les lignes contenant des `String` initialisés directement dans le code :

Code : Java - com.sdzee.servlets.Test

```
package com.sdzee.servlets;

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

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

import org.joda.time.DateTime;

import com.sdzee.beans.CoyoteBean;

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

  /** Création et initialisation du message. */
  String message = "Message transmis de la servlet à la JSP.";

  /** Création du bean et initialisation de ses propriétés */
  CoyoteBean premierBean = new CoyoteBean();
  premierBean.setNom( "Coyote" );
  premierBean.setPrenom( "Wile E." );

  /** Création de la liste et insertion de quatre éléments */
  List<Integer> premiereListe = new ArrayList<Integer>();
  premiereListe.add( 27 );
  premiereListe.add( 12 );
  premiereListe.add( 138 );
  premiereListe.add( 6 );

  /** On utilise ici la libraire Joda pour manipuler les dates,
pour deux raisons :
* - c'est tellement plus simple et limpide que de travailler avec
les objets Date ou Calendar !
* - c'est (probablement) un futur standard de l'API Java.
```

```
        */
          DateTime dt = new DateTime();
          Integer jourDuMois = dt.getDayOfMonth();

          /** Stockage du message, du bean et de la liste dans l'objet
        request */
        request.setAttribute( "test", message );
        request.setAttribute( "coyote", premierBean );
        request.setAttribute( "liste", premiereListe );
        request.setAttribute( "jour", jourDuMois );

          /** Transmission de la paire d'objets request/response à notre
        JSP */
        this.getServletContext().getRequestDispatcher( "/WEB-INF/test.jsp"
        ).forward( request, response );
         }
        }
```

Je n'ai volontairement pas surligné les lignes 20, 24 à 25 et 29 à 32, car bien qu'elles contiennent des `String` et `int` en dur, il s'agit ici simplement de l'initialisation des données d'exemple que nous transmettons à notre JSP : ce sont des données "externes". Dans le cas d'une réelle application, ces données seront issues de la base de données, du modèle ou encore d'une saisie utilisateur, mais bien évidemment jamais directement issues de la servlet comme c'est le cas ici dans l'exemple.

En ce qui concerne les String initialisées en dur dans les lignes surlignées par contre, vous devez remarquer qu'elles ne contiennent que des données "internes" : en l'occurrence, un nom de page JSP et quatre noms d'attributs. Il s'agit bien ici de données propres au fonctionnement de l'application et non pas de données destinées à être transmises à la vue pour affichage.

Eh bien comme je vous l'ai annoncé, une bonne pratique est de remplacer ces initialisations directes par des constantes, regroupées en tête de classe. Voici donc le code de notre servlet après modification :

Code : Java - com.sdzee.servlets.Test

```
        package com.sdzee.servlets;

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

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

        import org.joda.time.DateTime;

        import com.sdzee.beans.CoyoteBean;

        public class Test extends HttpServlet {
        public static final String ATT_MESSAGE = "test";
        public static final String ATT_BEAN = "coyote";
        public static final String ATT_LISTE = "liste";
        public static final String ATT_JOUR = "jour";
        public static final String VUE = "/WEB-INF/test.jsp";

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

          /** Création et initialisation du message. */
          String message = "Message transmis de la servlet à la JSP.";

          /** Création du bean et initialisation de ses propriétés */
          CoyoteBean premierBean = new CoyoteBean();
          premierBean.setNom( "Coyote" );
          premierBean.setPrenom( "Wile E." );

          /** Création de la liste et insertion de quatre éléments */
          List<Integer> premiereListe = new ArrayList<Integer>();
```

```
        premiereListe.add( 27 );
        premiereListe.add( 12 );
        premiereListe.add( 138 );
        premiereListe.add( 6 );

        /** On utilise ici la libraire Joda pour manipuler les dates,
      pour deux raisons :
      * - c'est tellement plus simple et limpide que de travailler avec
      les objets Date ou Calendar !
      * - c'est (probablement) un futur standard de l'API Java.
      */
        DateTime dt = new DateTime();
        Integer jourDuMois = dt.getDayOfMonth();

        /** Stockage du message, du bean et de la liste dans l'objet
      request */
      request.setAttribute( ATT_MESSAGE, message );
      request.setAttribute( ATT_BEAN, premierBean );
      request.setAttribute( ATT_LISTE, premiereListe );
      request.setAttribute( ATT_JOUR, jourDuMois );

        /** Transmission de la paire d'objets request/response à notre
      JSP */
      this.getServletContext().getRequestDispatcher( VUE ).forward(
      request, response );
       }
      }
```

Vous visualisez bien ici l'intérêt d'une telle pratique : en début de code sont accessibles en un coup d'œil toutes les données utilisées en dur au sein de la classe. Si vous nommez intelligemment vos constantes, vous pouvez alors sans avoir à parcourir le code savoir quelle constantes correspond à quelle donnée. Ici par exemple, j'ai préfixé les noms des attributs de requête par "ATT_" et nommé "VUE" la constante contenant le chemin vers notre page JSP. Ainsi, si vous procédez plus tard à une modification sur une de ces données, il vous suffira de modifier la valeur de la constante correspondante et vous n'aurez pas besoin de parcourir votre code. C'est d'autant plus utile que votre classe est volumineuse : plus long est votre code, plus pénible il sera d'y chercher les données initialisées en dur.

Dorénavant, dans tous les exemples de code à venir dans la suite du cours, je mettrai en place de telles constantes.

Inclure automatiquement la JSTL *Core* à toutes vos JSP

Vous le savez, pour pouvoir utiliser les balises de la bibliothèqe Core dans vos pages JSP, il est nécessaire de faire intervenir la directive **include** en tête de page :

Code : JSP

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

Admettons-le : dans une application, rares seront les vues qui ne nécessiteront pas l'utilisation de balises issues de la JSTL. Afin d'éviter d'avoir à dupliquer cette ligne dans l'intégralité de vos vues, il existe un moyen de rendre cette inclusion automatique ! C'est dans le fichier **web.xml** que vous avez la possibilité de spécifier une telle section :

Code : XML

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
 <jsp-config>
  <jsp-property-group>
   <url-pattern>*.jsp</url-pattern>
   <include-prelude>/WEB-INF/taglibs.jsp</include-prelude>
  </jsp-property-group>
 </jsp-config>
```

```
...
```

Le fonctionnement est très simple, la balise **<jsp-property-group>** ne contenant dans notre cas que deux balises :

- **<url-pattern>**, qui permet comme vous vous en doutez de **spécifier à quels fichiers appliquer l'inclusion** automatique. Ici, j'ai choisi de l'appliquer à tous les fichiers JSP de l'application !
- **<include-prelude>**, qui permet de préciser l'emplacement du fichier à inclure **en tête de chacune des pages** couvertes par le pattern précédemment défini. Ici, J'ai nommé ce fichier taglibs.jsp .

Il ne nous reste donc plus qu'à créer un fichier taglibs.jsp sous le répertoire **/WEB-INF** de notre application, et à y placer la directive taglib que nous souhaitons voir apparaître sur chacune de nos pages JSP :

Code : JSP - Contenu du fichier taglibs.jsp

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

Redémarrez Tomcat pour que les modifications apportées au fichier web.xml soient prises en compte, et vous n'aurez dorénavant plus besoin de préciser la directive en haut de vos pages JSP : cela se fera de manière transparente !

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

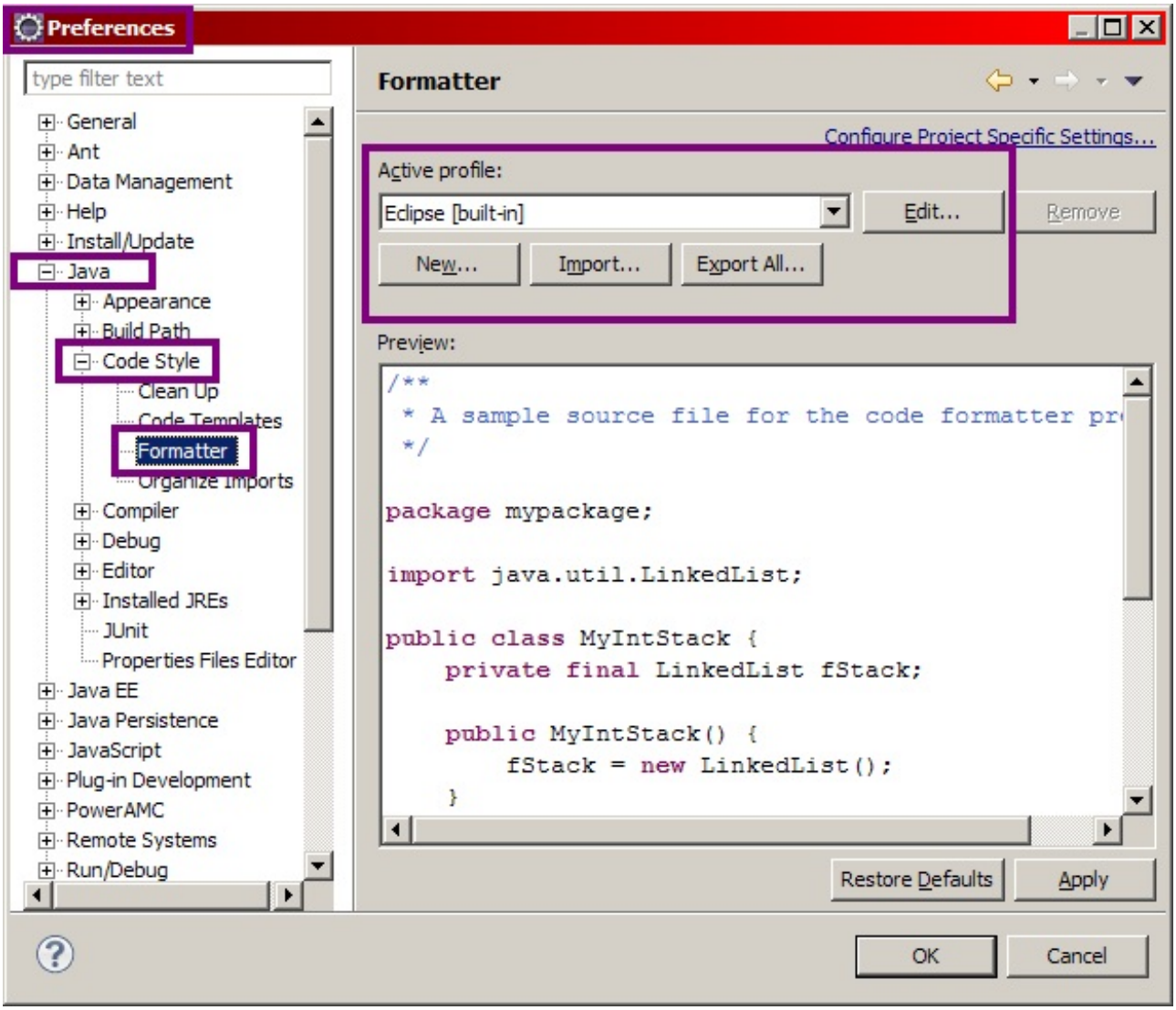
Nous n'en avons pas l'utilité ici, mais sachez qu'il est possible avec ce même système d'inclure automatiquement un fichier **en fin de page** : il faut pour cela préciser le fichier à inclure au sein d'une balise **<include-coda>**, et non plus **<include-prelude>** comme nous l'avons fait dans notre exemple. Le principe de fonctionnement reste identique, seul le nom de la balise diffère.

Formater proprement et automatiquement votre code avec Eclipse

Produire un code propre et lisible est très important lorsque vous travaillez sur un projet, et c'est d'autant plus vrai dans le cas d'un projet professionnel et en équipe. Toutefois, harmoniser son style d'écriture sur l'ensemble des classes que l'on rédige n'est pas toujours évident, il est difficile de faire preuve d'une telle rigueur. Pour nous faciliter la tâche, Eclipse propose un système de formatage automatique du code !

Créer un style de formatage

Sous Eclipse, rendez-vous dans le menu Window > Preferences > Java > Code Style > Formatter, comme indiqué sur la capture d'écran suivante :

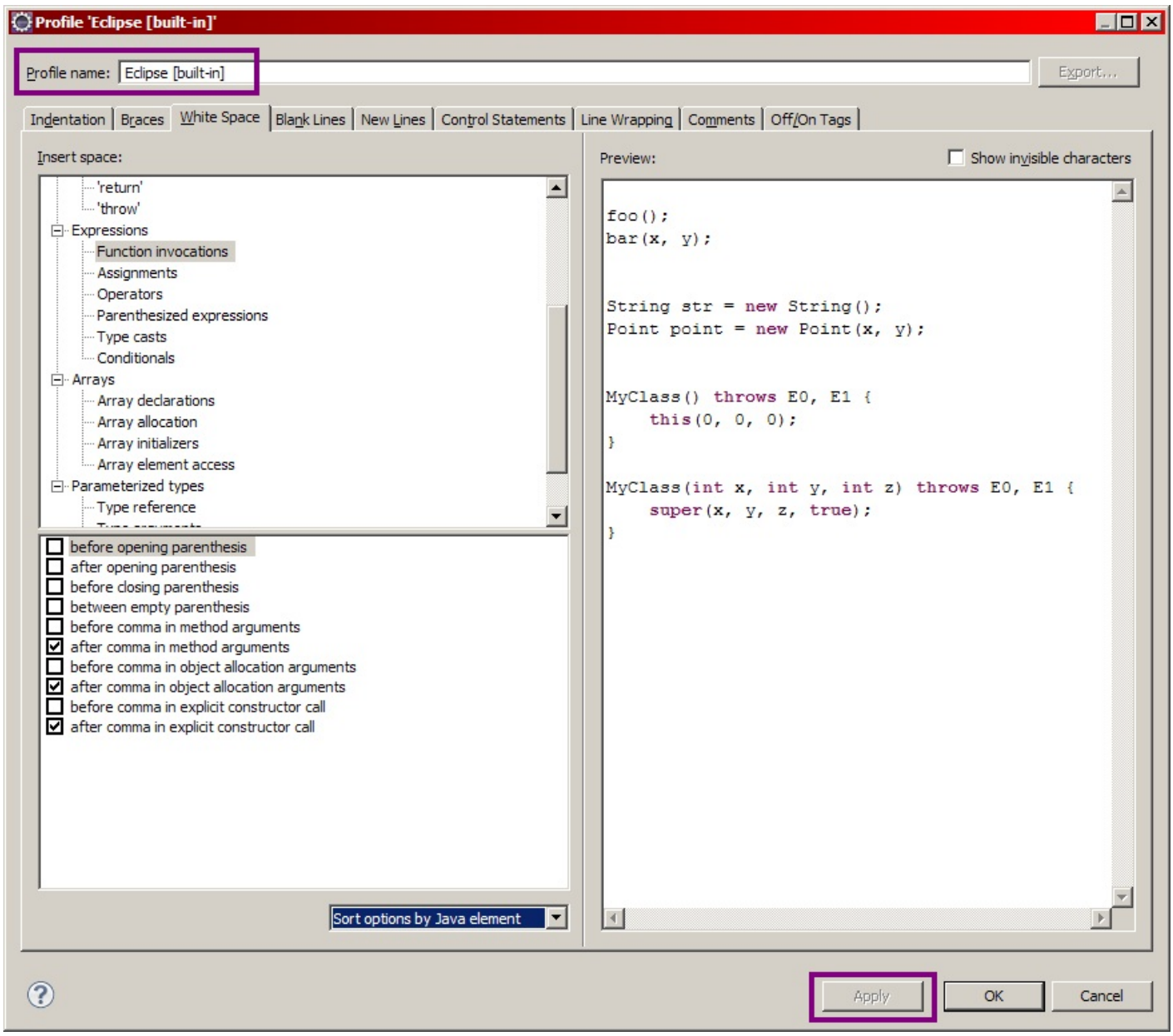


Eclipse format tool.

Le volet de droite de cette fenêtre est composé de plusieurs blocs :

- un lien intitulé "Configure Project Specific Settings...", qui vous redirige vers la fenêtre de configuration pour un projet en particulier uniquement ;
- un formulaire d'édition des profils de formatage existant ;
- un cadre d'aperçu qui vous montre l'apparence de votre code lorsque le profil actuellement en place est utilisé.

Pour modifier le style de formatage par défaut, il suffit de cliquer sur le bouton Edit.... Vous accédez alors à une vaste interface vous permettant de personnaliser un grand nombre d'options :

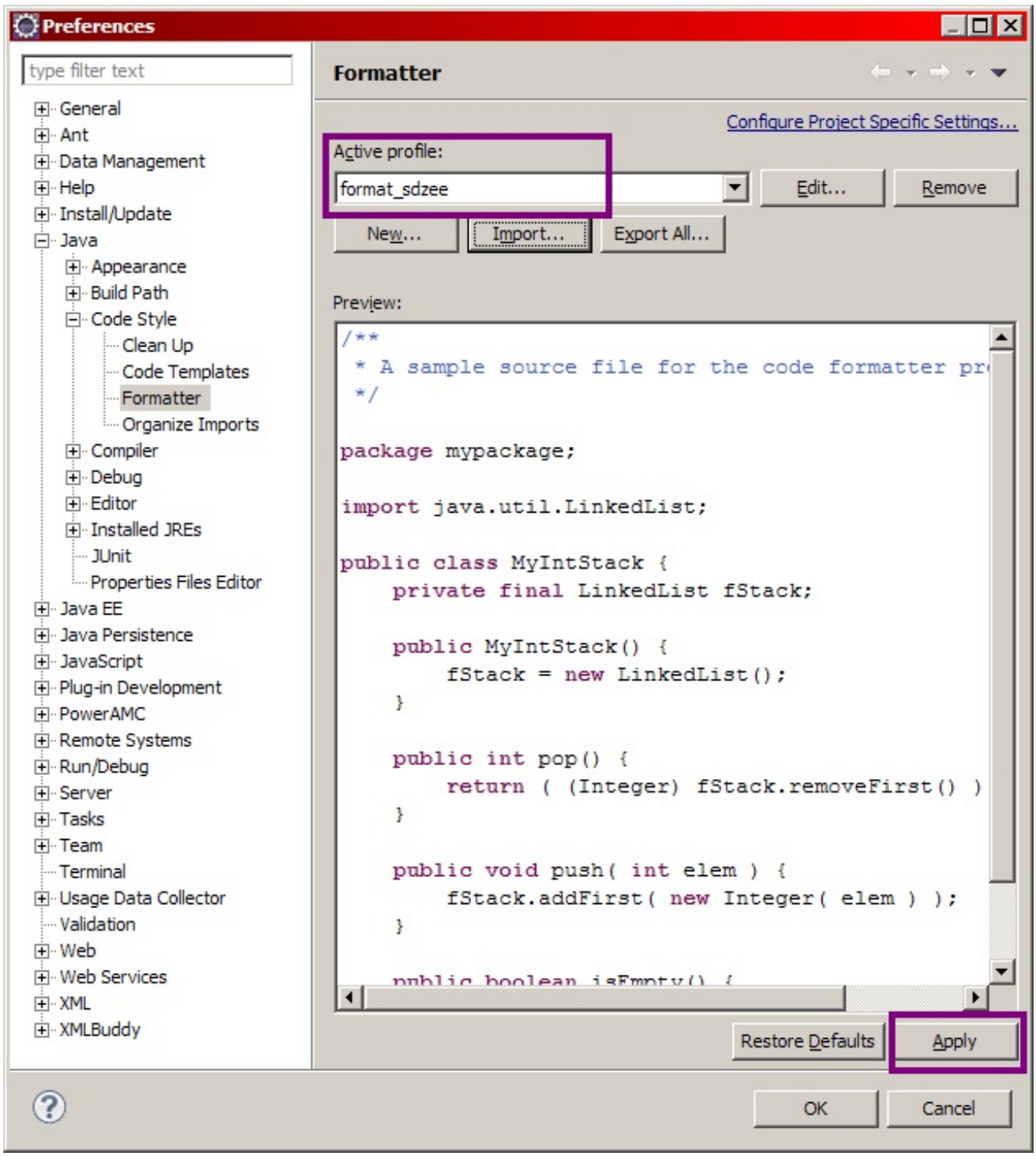


Options de formatage du code.

Je laisse aux plus motivés d'entre vous le loisir de parcourir les différents onglets et de modifier eux-mêmes le style de formatage. Vous devrez pour que vos modifications soient prises en compte changer le nom du profil actuel, dans l'encadré en haut de la fenêtre, puis valider les changements en cliquant sur le bouton Apply en bas de fenêtre.

Pour tous les autres, j'ai créé un modèle de formatage prêt à l'emploi, que je vous propose de mettre en place et d'utiliser pour formater vos fichiers sources :
=> Télécharger le fichier format_sdzee.xml (clic droit, puis "Enregistrer sous...")

Une fois le fichier téléchargé, il vous suffit de l'importer dans votre Eclipse en cliquant sur le bouton Import... dans le formulaire de la première fenêtre :



Import du modèle de formatage.

Le nom du profil change alors pour **format_sdzee**, et il vous reste enfin à appliquer les changements en cliquant sur le bouton Apply en bas de fenêtre.

Utiliser un style de formatage

Maintenant que le profil est en place, vous pouvez formater automatiquement votre code source Java. Pour cela, ouvrez un fichier Java quelconque de votre projet, et rendez-vous dans le menu Source > Format, ou tapez tout simplement le raccourci clavier Ctrl + Maj + F. Prenons pour exemple le code de notre servlet **Test** :

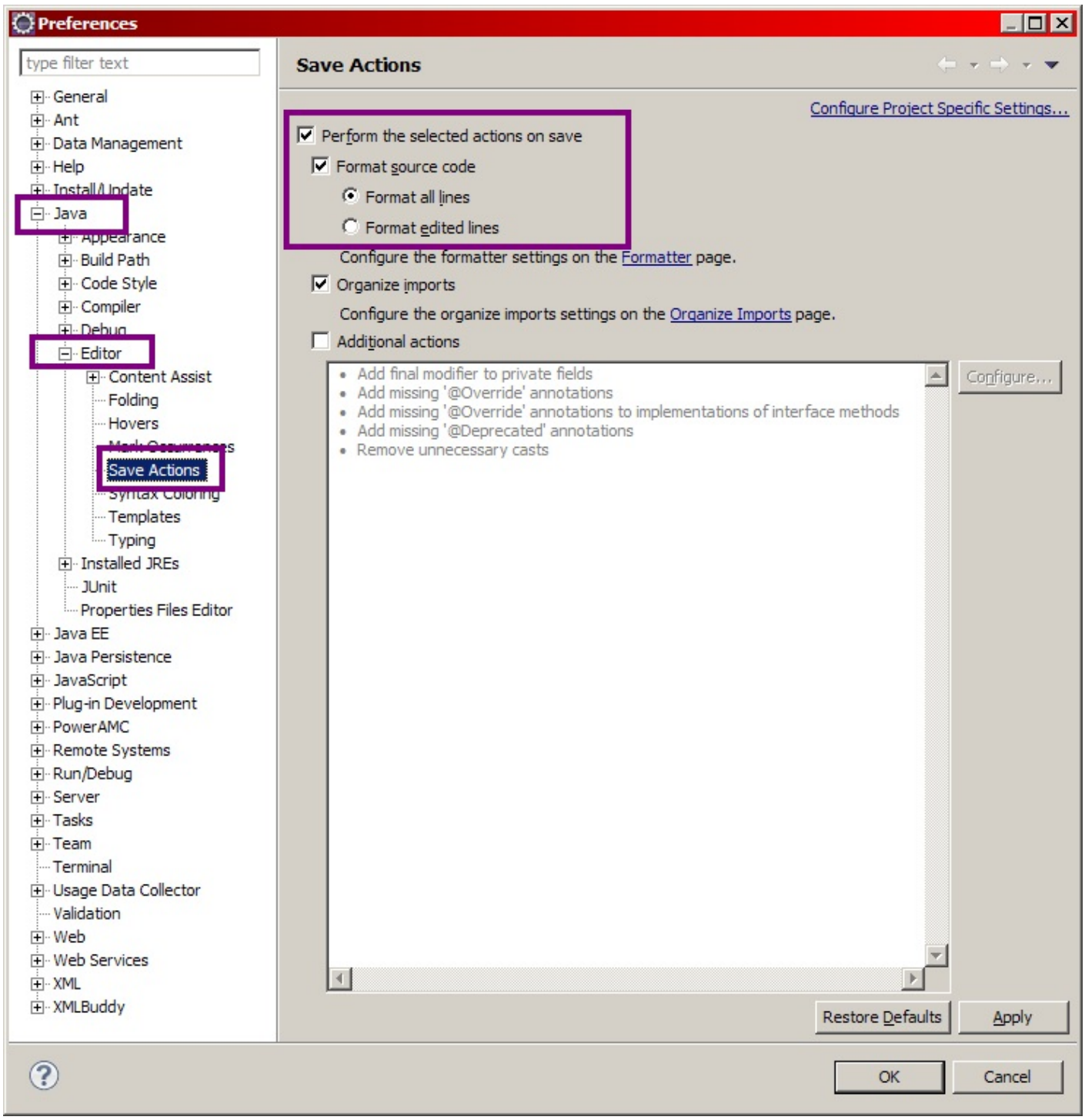
Rendu du formatage de la source.

À gauche la version non formatée, et à droite la version après formatage. La différence n'est pas énorme sur un code aussi court, d'autant plus que le code d'origine était déjà relativement bien organisé et indenté. Vous pouvez toutefois remarquer quelques changements aérant le code et facilitant sa lecture :

- l'alignement des valeurs des constantes en tête de classe ;
- l'ajout d'espaces après l'ouverture et avant la fermeture de parenthèses.

Automatiser le formatage à chaque sauvegarde

Nous voilà mieux équipés, mais il reste un détail pénible : il nous faut encore taper Ctrl + Maj + F à chaque fois que nous effectuons des modifications dans le code source, afin de conserver un formatage parfait. Comme nous sommes fénéants, nous allons demander à Eclipse d'y penser pour nous ! Rendez-vous dans le menu Window > Preferences > Java > Editor > Save Actions :



Formatage automatique.

Comme indiqué dans l'encadré, cochez les cases "Perform the selected actions on save" et "Format source code". Validez les changements, et c'est terminé : votre code source Java sera formaté automatiquement selon les règles définies dans votre profil, à chaque fois que vous enregistrerez des modifications effectuées sur un fichier.

Vous n'avez dorénavant plus aucune excuse : votre code doit être correctement formaté, organisé et indenté !

Documentation

Les tutoriaux d'auteurs différents vous feront profiter de nouveaux points de vue et angles d'attaque, et les documentations officielles vous permettront un accès à des informations justes et maintenues à jour (en principe).

Liens utiles

- JSTL 1.1 Tag Reference (un Javadoc-like bien pratique), sur oracle.com
- JSP Standard Tag Library, sur java.sun.com

- À propos de la JSTL, sur stackoverflow.com
- Une introduction à la JSTL, sur developer.com
- Tutoriel sur la JSTL, sur developpez.com
- Tutoriel sur les TagLib, sur developpez.com
- FAQ à propos des TagLib, sur developpez.com
- Tutoriel complet sur l'utilisation de la JSTL, sur ibm.com

Vous l'aurez compris, cette liste ne se veut pas exhaustive, et je vous recommande d'aller chercher par vous même l'information sur les forums et sites du web. En outre, faites bien attention aux dates de création des documents que vous lisez : **les ressources périmées font légion sur le web, notamment au sujet de la plate-forme Java EE**, en constante évolution. N'hésitez pas à demander à la communauté sur le forum Java du Site du Zéro, si vous ne parvenez pas à trouver l'information que vous cherchez.
Finie la détente, une partie extrêmement importante nous attend : la gestion des formulaires. Accrochez-vous, le rythme va s'accélérer !

TP Fil rouge - Étape 2

Dans ce second opus de notre fil rouge, vous allez appliquer ce que vous avez découvert à propos de la JSTL Core et des bonnes pratiques de développement.

Objectifs

Les précédents chapitres concernant uniquement la vue, vous allez ici principalement vous consacrer à la reprise des pages JSP que vous aviez créées lors du premier TP.

Je vous conseille de repartir sur la base de la correction que je vous ai donnée pour la première étape, cela facilitera votre compréhension des étapes et corrections à venir.

Utilisation de la JSTL

L'objectif est modeste et le programme léger, mais l'important est que vous compreniez ce que vous faites et que vous preniez de l'aise avec le système des balises de la JSTL. Je vous demande de :

- sécuriser contre les failles XSS l'affichage des données saisies par l'utilisateur dans vos pages **afficherClient.jsp** et **afficherCommande.jsp** ;
- gérer dynamiquement les différents liens et URL qui interviennent dans le code de vos pages JSP ;
- créer un menu, qui ne contiendra pour le moment que deux liens respectivement vers **creerClient.jsp** et **creerCommande.jsp**, et l'intégrer à toutes vos pages existantes ;
- mettre en place une condition à l'affichage du résultat de la validation : si les données ont été correctement saisies, alors afficher le message de succès et la fiche récapitulative, sinon afficher **uniquement** le message d'erreur.

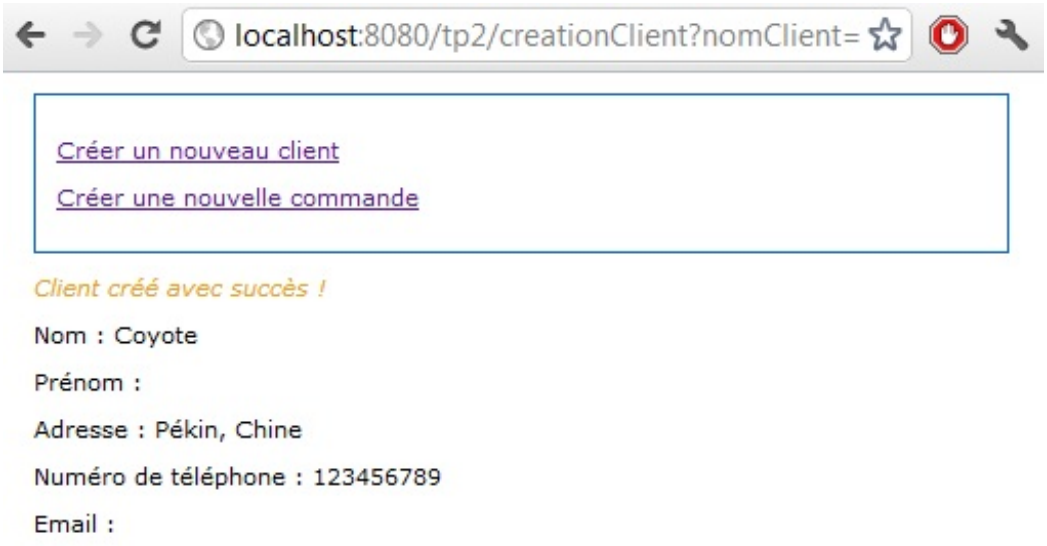
Application des bonnes pratiques

Je vous demande dans le code de vos servlets de mettre en place des constantes, pour remplacer les chaînes de caractères initialisées directement au sein du code des méthodes `doGet()`.

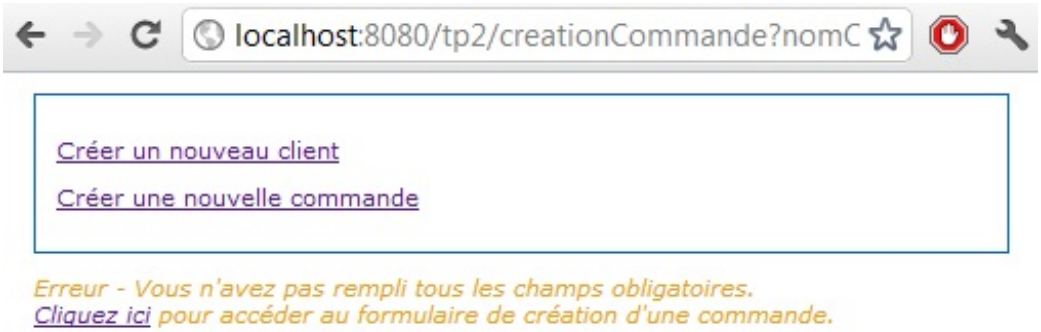
Exemples de rendus

Création de client avec erreurs :

Création de client sans erreur :



Création de commande avec erreurs :



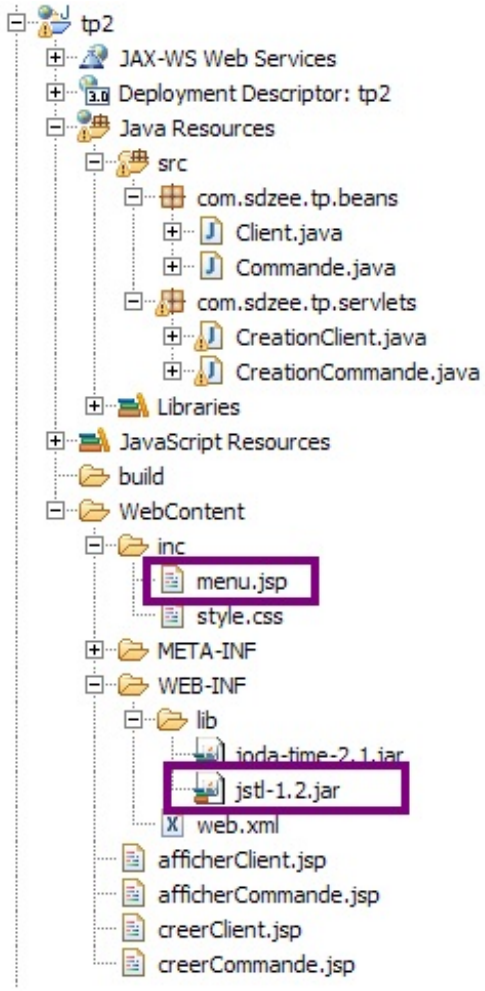
Création de commande sans erreur :

Conseils
Utilisation de la JSTL

Je vous donne tout de même quelques pistes pour vous guider, mais vous devriez être capables de vous en sortir même sans lire ce paragraphe :

- pour sécuriser l'affichage des données saisies, pensez à la balise `<c:out/>` ;
- pour la gestion des liens, pensez à la balise `<c:url/>` ;
- pour le menu, vous pouvez créer une page **menu.jsp** que vous placerez dans le répertoire **/inc** et que vous inclurez dans toutes les autres pages grâce à la balise `<c:import/>` ;
- pour la condition, vous pouvez modifier vos servlets pour qu'elles transmettent à vos JSP une information supplémentaire - pourquoi pas un booléen - afin que celles-ci puissent déterminer si la validation a été effectuée avec succès ou non grâce à la balise `<c:choose/>` ou `<c:if/>`.

Bien évidemment, vous n'oublierez pas d'inclure le jar de la JSTL au répertoire **lib** de votre projet, afin de rendre les balises opérationnelles. Voici l'allure de l'arborescence que vous devriez obtenir une fois le TP terminé :



Vous pouvez remarquer en encadré les deux nouveaux fichiers intervenant dans votre projet : le fichier jar de la JSTL et la page **menu.jsp**.

Application des bonnes pratiques

Il vous suffit ici de remplacer toutes les chaînes de caractères utilisées directement dans le code de vos servlets par des constantes définies en dehors des méthodes `doGet()`, comme je vous l'ai montré dans l'avant-dernier chapitre.

C'est tout ce dont vous avez besoin. Au travail !

Correction

Faible dose de travail cette fois, j'espère que vous avez bien pris le temps de relire les explications concernant les différentes balises à mettre en jeu. Encore une fois, ce n'est pas la seule manière de faire, ne vous inquiétez pas si vous avez procédé différemment, le principal est que vous ayez couvert tout ce qu'il fallait couvrir ! Pour que vous puissiez repérer rapidement ce qui a changé, j'ai surligné les modifications apportées aux codes existants.

Prenez le temps de réfléchir, de chercher et coder par vous-mêmes. Si besoin, n'hésitez pas à relire le sujet ou à retourner lire certains chapitres. La pratique est très importante, ne vous ruez pas sur la solution !

Code des servlets

Secret (cliquez pour afficher)

Servlet gérant le formulaire de création de client :

Code : Java - com.sdzee.tp.servlets.CreationClient

```
package com.sdzee.tp.servlets;

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.sdzee.tp.beans.Client;

public class CreationClient extends HttpServlet {
    /* Constantes */
public static final String CHAMP_NOM = "nomClient";
public static final String CHAMP_PRENOM = "prenomClient";
public static final String CHAMP_ADRESSE = "adresseClient";
public static final String CHAMP_TELEPHONE = "telephoneClient";
public static final String CHAMP_EMAIL = "emailClient";

public static final String ATT_CLIENT = "client";
public static final String ATT_MESSAGE = "message";
public static final String ATT_ERREUR = "erreur";

public static final String VUE = "/afficherClient.jsp";

    public void doGet( HttpServletRequest request,
HttpServletResponse response ) throws ServletException,
IOException {
        /*
* Récupération des données saisies, envoyées en tant que
paramètres de
* la requête GET générée à la validation du formulaire
*/
String nom = request.getParameter( CHAMP_NOM );
String prenom = request.getParameter( CHAMP_PRENOM );
String adresse = request.getParameter( CHAMP_ADRESSE );
String telephone = request.getParameter( CHAMP_TELEPHONE );
String email = request.getParameter( CHAMP_EMAIL );

        String message;
boolean erreur;
        /*
* Initialisation du message à afficher : si un des champs
obligatoires
* du formulaire n'est pas renseigné, alors on affiche un message
* d'erreur, sinon on affiche un message de succès
*/
        if ( nom.trim().isEmpty() || adresse.trim().isEmpty() ||
telephone.trim().isEmpty() ) {
```

```
            message = "Erreur - Vous n'avez pas rempli tous les
champs obligatoires. <br> <a href=\"creerClient.jsp\">Cliquez
ici</a> pour accéder au formulaire de création d'un client.";
erreur = true;
        } else {
            message = "Client créé avec succès !";
erreur = false;
        }
        /*
* Création du bean Client et initialisation avec les données
récupérées
*/
        Client client = new Client();
        client.setNom( nom );
        client.setPrenom( prenom );
        client.setAdresse( adresse );
        client.setTelephone( telephone );
        client.setEmail( email );

        /* Ajout du bean et du message à l'objet requête */
request.setAttribute( ATT_CLIENT, client );
request.setAttribute( ATT_MESSAGE, message );
request.setAttribute( ATT_ERREUR, erreur );

        /* Transmission à la page JSP en charge de l'affichage
des données */
this.getServletContext().getRequestDispatcher( VUE ).forward(
request, response );
    }
}
```

Servlet gérant le formulaire de création de commande :

Code : Java - com.sdzee.tp.servlets.CreationCommande

```
package com.sdzee.tp.servlets;

import java.io.IOException;

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

import org.joda.time.DateTime;
import org.joda.time.format.DateTimeFormat;
import org.joda.time.format.DateTimeFormatter;

import com.sdzee.tp.beans.Client;
import com.sdzee.tp.beans.Commande;

public class CreationCommande extends HttpServlet {
    /* Constantes */
public static final String CHAMP_NOM = "nomClient";
public static final String CHAMP_PRENOM = "prenomClient";
public static final String CHAMP_ADRESSE = "adresseClient";
public static final String CHAMP_TELEPHONE = "telephoneClient";
public static final String CHAMP_EMAIL = "emailClient";

public static final String CHAMP_DATE = "dateCommande";
public static final String CHAMP_MONTANT = "montantCommande";
public static final String CHAMP_MODE_PAIEMENT =
"modePaiementCommande";
public static final String CHAMP_STATUT_PAIEMENT =
"statutPaiementCommande";
public static final String CHAMP_MODE_LIVRAISON =
```

```
"modeLivraisonCommande";
public static final String CHAMP_STATUT_LIVRAISON =
"statutLivraisonCommande";

public static final String ATT_COMMANDE = "commande";
public static final String ATT_MESSAGE = "message";
public static final String ATT_ERREUR = "erreur";

public static final String FORMAT_DATE = "dd/MM/yyyy HH:mm:ss";

public static final String VUE = "/afficherCommande.jsp";

    public void doGet( HttpServletRequest request,
HttpServletResponse response ) throws ServletException,
IOException {
        /*
* Récupération des données saisies, envoyées en tant que
paramètres de
* la requête GET générée à la validation du formulaire
*/
String nom = request.getParameter( CHAMP_NOM );
String prenom = request.getParameter( CHAMP_PRENOM );
String adresse = request.getParameter( CHAMP_ADRESSE );
String telephone = request.getParameter( CHAMP_TELEPHONE );
String email = request.getParameter( CHAMP_EMAIL );

        /* Récupération de la date courante */
        DateTime dt = new DateTime();
        /* Conversion de la date en String selon le format
choisi */
DateTimeFormatter formatter = DateTimeFormat.forPattern(
FORMAT_DATE );
        String date = dt.toString( formatter );
        double montant;
        try {
            /* Récupération du montant */
montant = Double.parseDouble( request.getParameter( CHAMP_MONTANT
) );
        } catch ( NumberFormatException e ) {
            /* Initialisation à -1 si le montant n'est pas un
nombre correct */
            montant = -1;
        }
String modePaiement = request.getParameter( CHAMP_MODE_PAIEMENT );
String statutPaiement = request.getParameter(
CHAMP_STATUT_PAIEMENT );
String modeLivraison = request.getParameter( CHAMP_MODE_LIVRAISON
);
String statutLivraison = request.getParameter(
CHAMP_STATUT_LIVRAISON );

        String message;
boolean erreur;
        /*
* Initialisation du message à afficher : si un des champs
obligatoires
* du formulaire n'est pas renseigné, alors on affiche un message
* d'erreur, sinon on affiche un message de succès
*/
        if ( nom.trim().isEmpty() || adresse.trim().isEmpty() ||
telephone.trim().isEmpty() || montant == -1
                || modePaiement.isEmpty() ||
modeLivraison.isEmpty() ) {
            message = "Erreur - Vous n'avez pas rempli tous les
champs obligatoires. <br> <a href=\"creerCommande.jsp\">Cliquez
ici</a> pour accéder au formulaire de création d'une commande.";
erreur = true;
        } else {
            message = "Commande créée avec succès !";
erreur = false;
```

```
            }
            /*
        * Création des beans Client et Commande et initialisation avec
        les
        * données récupérées
        */
            Client client = new Client();
            client.setNom( nom );
            client.setPrenom( prenom );
            client.setAdresse( adresse );
            client.setTelephone( telephone );
            client.setEmail( email );

            Commande commande = new Commande();
            commande.setClient( client );
            commande.setDate( date );
            commande.setMontant( montant );
            commande.setModePaiement( modePaiement );
            commande.setStatutPaiement( statutPaiement );
            commande.setModeLivraison( modeLivraison );
            commande.setStatutLivraison( statutLivraison );

            /* Ajout du bean et du message à l'objet requête */
        request.setAttribute( ATT_COMMANDE, commande );
        request.setAttribute( ATT_MESSAGE, message );
        request.setAttribute( ATT_ERREUR, erreur );

            /* Transmission à la page JSP en charge de l'affichage
        des données */
        this.getServletContext().getRequestDispatcher( VUE ).forward(
        request, response );
        }
    }
```

Code des JSP

Secret (cliquez pour afficher)

Page de création d'un client :

Code : JSP - /creerClient.jsp

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Création d'un client</title>
<link type="text/css" rel="stylesheet" href="<c:url
value="/inc/style.css"/>" />
    </head>
    <body>
<c:import url="/inc/menu.jsp" />
        <div>
<form method="get" action="<c:url value="/creationClient"/>">
                <fieldset>
                    <legend>Informations client</legend>

                    <label for="nomClient">Nom <span
class="requis">*</span></label>
                    <input type="text" id="nomClient"
name="nomClient" value="" size="30" maxlength="30" />
```

```
                        <br />

                        <label for="prenomClient">Prenom </label>
                        <input type="text" id="prenomClient"
name="prenomClient" value="" size="30" maxlength="30" />
                        <br />

                        <label for="adresseClient">Adresse de
livraison <span class="requis">*</span></label>
                        <input type="text" id="adresseClient"
name="adresseClient" value="" size="30" maxlength="60" />
                        <br />

                        <label for="telephoneClient">Numéro de
téléphone <span class="requis">*</span></label>
                        <input type="text" id="telephoneClient"
name="telephoneClient" value="" size="30" maxlength="30" />
                        <br />

                        <label for="emailClient">Adresse email</label>
                        <input type="email" id="emailClient"
name="emailClient" value="" size="30" maxlength="60" />
                        <br />
                    </fieldset>
                    <input type="submit" value="Valider"  />
                    <input type="reset" value="Remettre à zéro" /> <br
/>
                </form>
            </div>
        </body>
    </html>
```

Page de création d'une commande :

Code : JSP - /creerCommande.jsp

```
    <%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
    <!DOCTYPE html>
    <html>
        <head>
            <meta charset="utf-8" />
            <title>Création d'une commande</title>
    <link type="text/css" rel="stylesheet" href="<c:url
    value="/inc/style.css"/>" />
        </head>
        <body>
    <c:import url="/inc/menu.jsp" />
            <div>
    <form method="get" action="<c:url value="/creationCommande"/>">
                    <fieldset>
                        <legend>Informations client</legend>

                        <label for="nomClient">Nom <span
    class="requis">*</span></label>
                        <input type="text" id="nomClient"
    name="nomClient" value="" size="30" maxlength="30" />
                        <br />

                        <label for="prenomClient">Prenom </label>
                        <input type="text" id="prenomClient"
    name="prenomClient" value="" size="30" maxlength="30" />
                        <br />

                        <label for="adresseClient">Adresse de
    livraison <span class="requis">*</span></label>
                        <input type="text" id="adresseClient"
```

```
        name="adresseClient" value="" size="30" maxlength="60" />
                        <br />

                        <label for="telephoneClient">Numéro de
        téléphone <span class="requis">*</span></label>
                        <input type="text" id="telephoneClient"
        name="telephoneClient" value="" size="30" maxlength="30" />
                        <br />

                        <label for="emailClient">Adresse email</label>
                        <input type="email" id="emailClient"
        name="emailClient" value="" size="30" maxlength="60" />
                        <br />
                    </fieldset>
                    <fieldset>
                        <legend>Informations commande</legend>

                        <label for="dateCommande">Date <span
        class="requis">*</span></label>
                        <input type="text" id="dateCommande"
        name="dateCommande" value="" size="30" maxlength="30" disabled />
                        <br />

                        <label for="montantCommande">Montant <span
        class="requis">*</span></label>
                        <input type="text" id="montantCommande"
        name="montantCommande" value="" size="30" maxlength="30" />
                        <br />

                        <label for="modePaiementCommande">Mode de
        paiement <span class="requis">*</span></label>
                        <input type="text" id="modePaiementCommande"
        name="modePaiementCommande" value="" size="30" maxlength="30" />
                        <br />

                        <label for="statutPaiementCommande">Statut du
        paiement</label>
                        <input type="text" id="statutPaiementCommande"
        name="statutPaiementCommande" value="" size="30" maxlength="30" />
                        <br />

                        <label for="modeLivraisonCommande">Mode de
        livraison <span class="requis">*</span></label>
                        <input type="text" id="modeLivraisonCommande"
        name="modeLivraisonCommande" value="" size="30" maxlength="30" />
                        <br />

                        <label for="statutLivraisonCommande">Statut de
        la livraison</label>
                        <input type="text"
        id="statutLivraisonCommande" name="statutLivraisonCommande"
        value="" size="30" maxlength="30" />
                        <br />
                    </fieldset>
                    <input type="submit" value="Valider"  />
                    <input type="reset" value="Remettre à zéro" /> <br
        />
                </form>
            </div>
        </body>
    </html>
```

Page d'affichage d'un client :

Code : JSP - /afficherClient.jsp

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Affichage d'un client</title>
<link type="text/css" rel="stylesheet" href="<c:url
value="/inc/style.css"/>" />
    </head>
    <body>
<c:import url="/inc/menu.jsp" />
<div id="corps">
            <p class="info">${ message }</p>
<c:if test="${ !erreur }">
<p>Nom : <c:out value="${ client.nom }"/></p>
<p>Prenom : <c:out value="${ client.prenom }"/></p>
<p>Adresse : <c:out value="${ client.adresse }"/></p>
<p>Numéro de téléphone : <c:out value="${ client.telephone
}"/></p>
<p>Email : <c:out value="${ client.email }"/></p>
</c:if>
</div>
    </body>
</html>
```

Page d'affichage d'une commande :

Code : JSP - /afficherCommande.jsp

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Affichage d'une commande</title>
<link type="text/css" rel="stylesheet" href="<c:url
value="/inc/style.css"/>" />
    </head>
    <body>
<c:import url="/inc/menu.jsp" />
<div id="corps">
            <p class="info">${ message }</p>
<c:if test="${ !erreur }">
<p>Client</p>
<p>Nom : <c:out value="${ commande.client.nom }"/></p>
<p>Prenom : <c:out value="${ commande.client.prenom }"/></p>
<p>Adresse : <c:out value="${ commande.client.adresse }"/></p>
<p>Numéro de téléphone : <c:out value="${
commande.client.telephone }"/></p>
<p>Email : <c:out value="${ commande.client.email }"/></p>
<p>Commande</p>
<p>Date : <c:out value="${ commande.date }"/></p>
<p>Montant : <c:out value="${ commande.montant }"/></p>
<p>Mode de paiement : <c:out value="${ commande.modePaiement
}"/></p>
<p>Statut du paiement : <c:out value="${ commande.statutPaiement
}"/></p>
<p>Mode de livraison : <c:out value="${ commande.modeLivraison
}"/></p>
<p>Statut de la livraison : <c:out value="${
commande.statutLivraison }"/></p>
</c:if>
</div>
    </body>
</html>
```

Page contenant le nouveau menu :

Code : JSP - /inc/menu.jsp

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<div id="menu">
    <p><a href="<c:url value="/creerClient.jsp"/>">Créer un
nouveau client</a></p>
    <p><a href="<c:url value="/creerCommande.jsp"/>">Créer une
nouvelle commande</a></p>
</div>
```

Ajout de styles pour la mise en page du menu :

Code : CSS - /inc/style.css

```
/* Général ------------------------------------------------------------
------------------------------ */

body, p, legend, label, input {
    font: normal 8pt verdana, helvetica, sans-serif;
}

/* Forms --------------------------------------------------------------
------------------------------ */
fieldset {
    padding: 10px;
    border: 1px #0568CD solid;
    margin: 10px;
}

legend {
    font-weight: bold;
    color: #0568CD;
}

form label {
    float: left;
    width: 200px;
    margin: 3px 0px 0px 0px;
}

form input {
    margin: 3px 3px 0px 0px;
    border: 1px #999 solid;
}

form input.sansLabel {
    margin-left: 200px;
}

/* Styles et couleurs -------------------------------------------------
------------------------------ */
.requis {
    color: #c00;
}

.erreur {
    color: #900;
}

.succes {
    color: #090;
```

```
}

.info {
    font-style: italic;
    color: #E8A22B;
}

/* Blocs constituants ----------------------------------------
---------------------------- */
div#menu{
border: 1px solid #0568CD;
padding: 10px;
margin: 10px;
}
div#corps{
margin: 10px;
}
```

Nous y voilà enfin : nous sommes capables de créer des vues qui suivent le modèle MVC .
Seulement maintenant que nous sommes au point, nous aimerions bien interagir avec notre client : comment récupérer et gérer les données qu'il va nous envoyer ?

Rendez-vous dans la partie suivante, nous avons du pain sur la planche !

Partie 4 : Une application interactive !

Il est temps de réellement faire entrer en jeu nos servlets, de donner un sens à leur existence : nous allons ici apprendre à gérer les informations envoyées par les clients à notre application. Et dans une application web, vous le savez déjà, qui dit interaction dit formulaire : cette partie aurait presque pu s'intituler *"Le formulaire dans tous ses états"*, car nous allons l'examiner sous toutes les coutures !

Au passage, nous en profiterons pour découvrir au travers d'applications pratiques l'utilisation des sessions, des cookies et d'un nouveau composant cousin de la servlet : le filtre.

Formulaires : le b.a.-ba

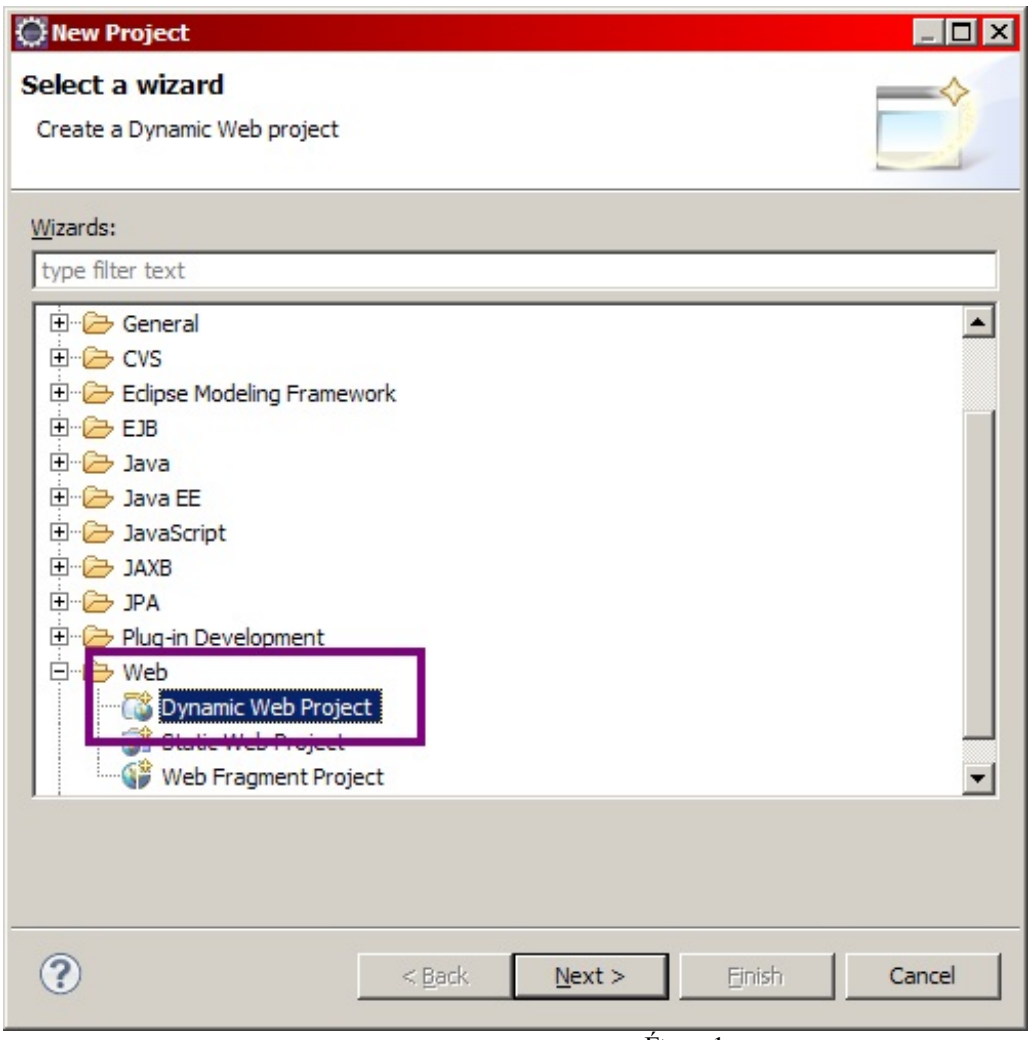
Dans cette partie, nous allons littéralement faire table rase. Laissons tomber nos précédents exemples, et attaquons l'étude des formulaires par quelque chose de plus concret : un formulaire d'inscription. Création, mise en place, récupération des données, affichage et vérifications nous attendent !

Bien entendu, nous n'allons pas pouvoir réaliser un vrai système d'inscription de A à Z : il nous manque encore pour cela la gestion des données, que nous n'allons découvrir que dans la prochaine partie de ce cours. Toutefois, nous pouvons d'ores et déjà réaliser proprement tout ce qui concerne les aspects vue, contrôle et traitement d'un tel système. Allons-y !

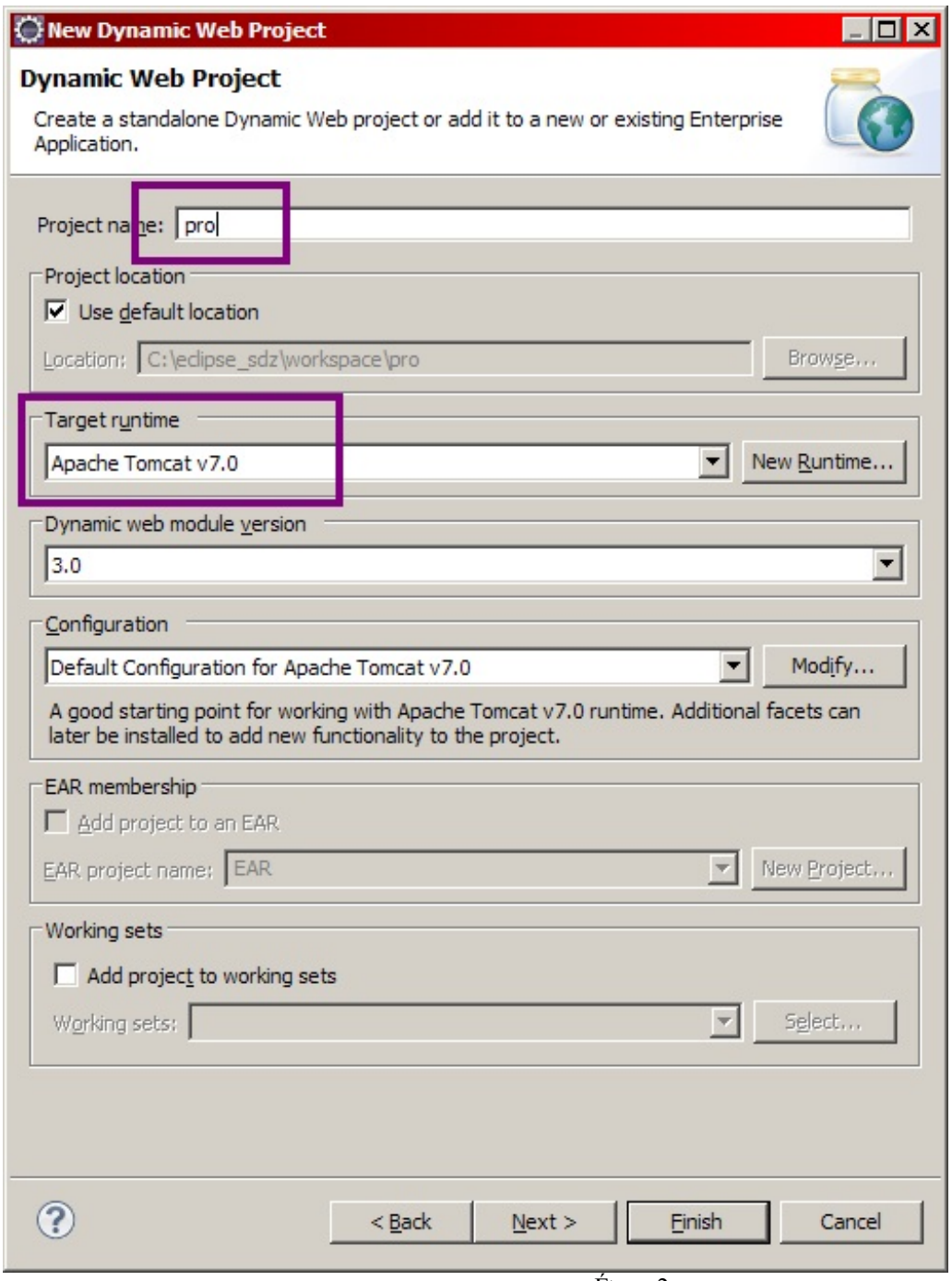
Mise en place

Je vous propose de mettre en place une base sérieuse qui nous servira d'exemple tout au long de cette partie du cours, ainsi que dans la partie suivante. Plutôt que de travailler une énième fois sur un embryon de page sans intérêt, je vais tenter ici de vous placer dans un contexte plus proche du monde professionnel : nous allons travailler de manière propre et organisée, et à la fin de ce cours nous aurons produit un exemple utilisable dans une réelle application.

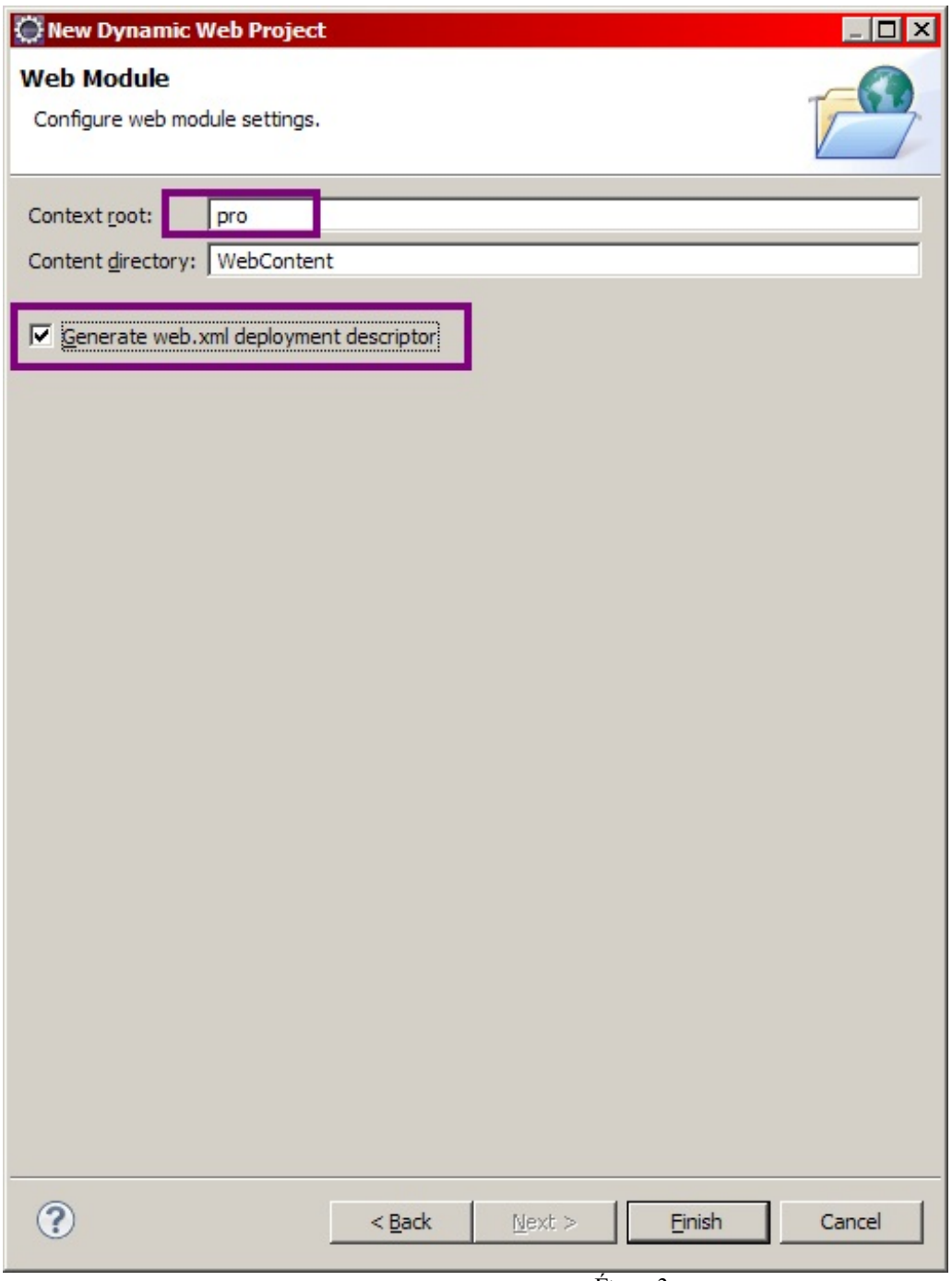
Pour commencer, je vous demande de créer **un nouveau projet dynamique sous Eclipse**. Laissez tomber le bac à sable que nous avions nommé **test**, et repartez de zéro : cela aura le double avantage de vous permettre de construire quelque chose de propre, et de vous faire pratiquer l'étape de mise en place d'un projet (création, build-path, bibliothèques, etc.). Je vous propose de nommer ce nouveau projet **pro**. Si vous ne vous souvenez pas de toutes les étapes à suivre, voici un aperçu des principaux écrans intervenant dans le processus de mise en place d'un projet :



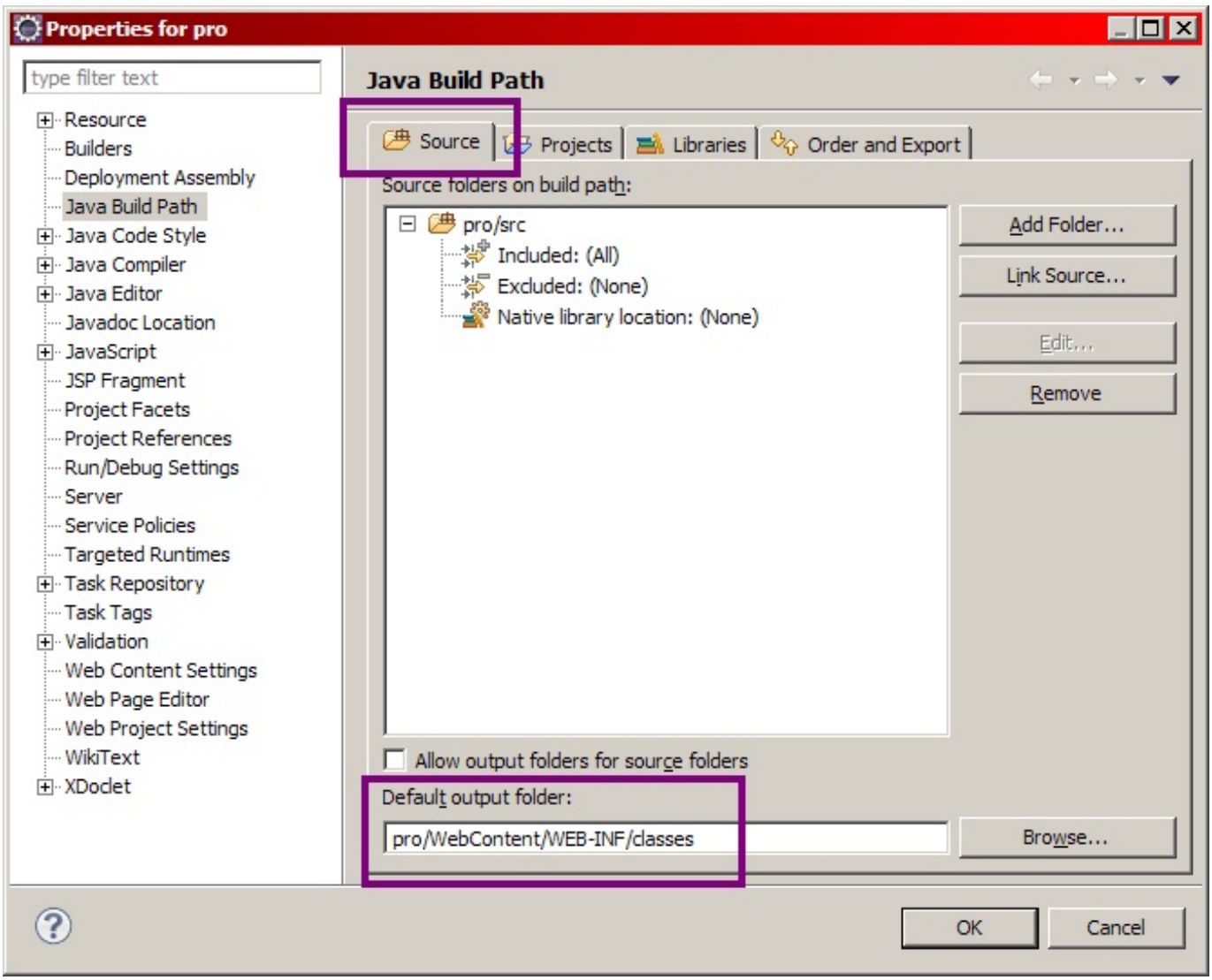
Création du projet dynamique - Étape 1



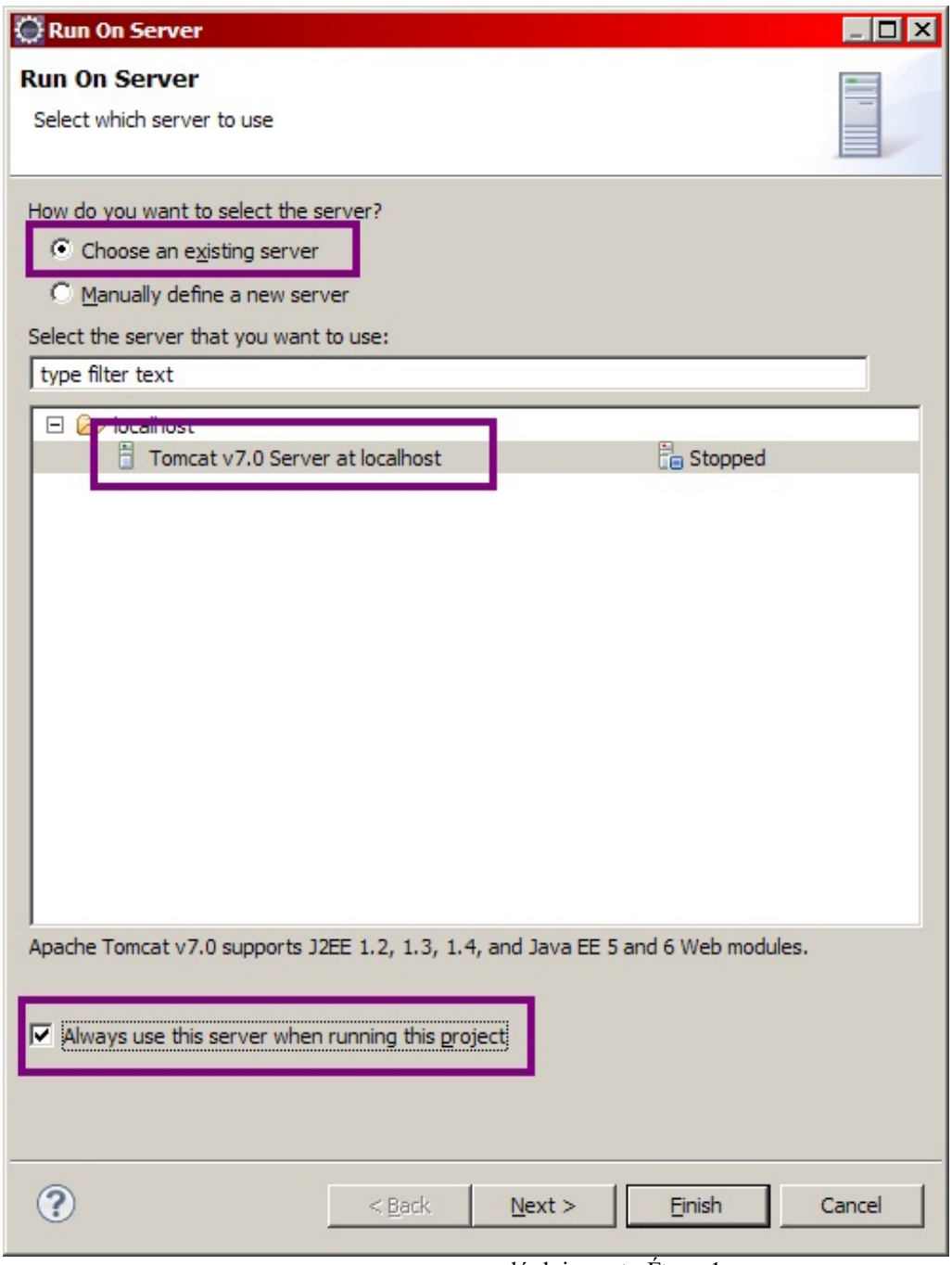
Création du projet dynamique - Étape 2



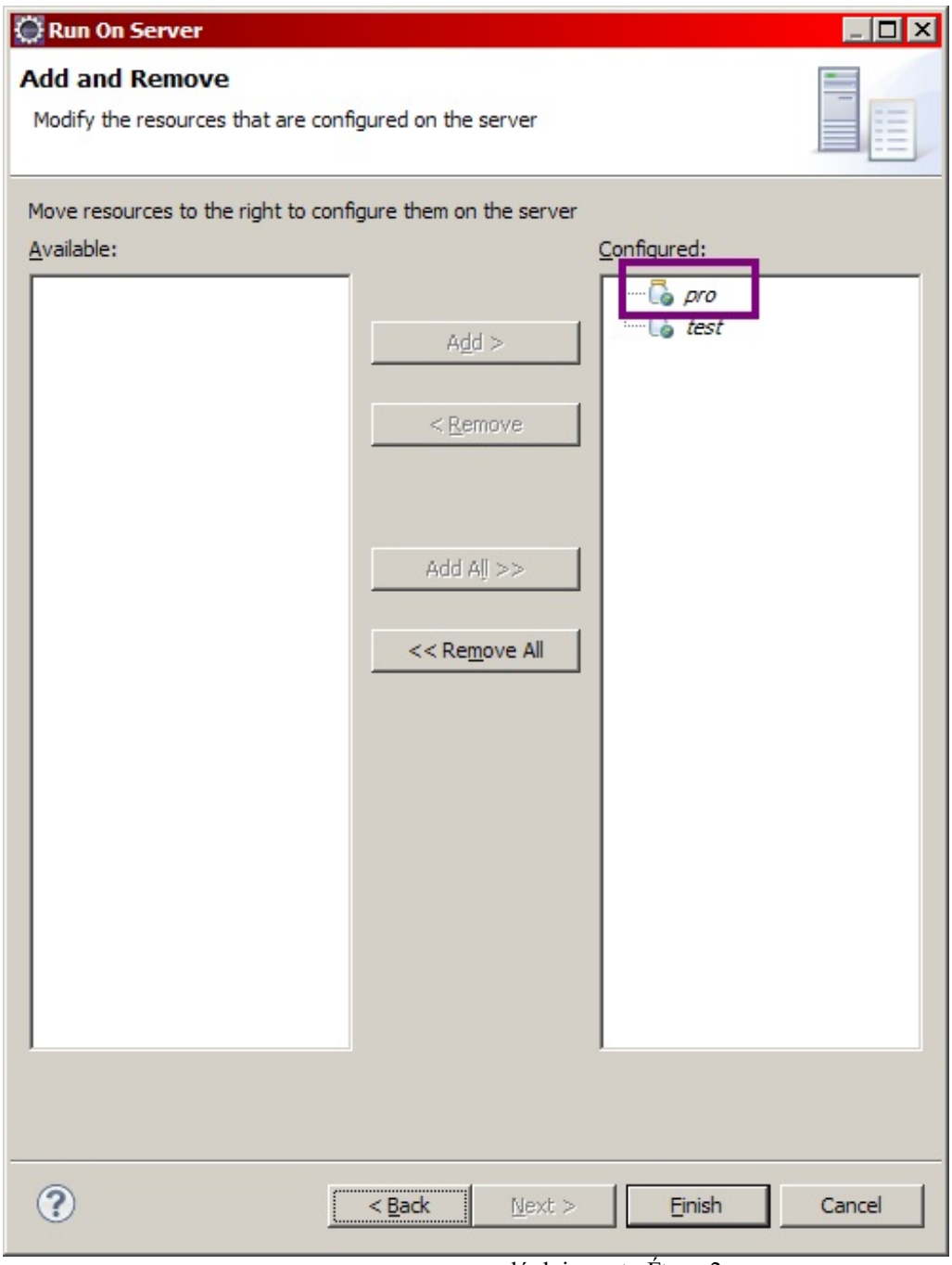
Création du projet dynamique - Étape 3



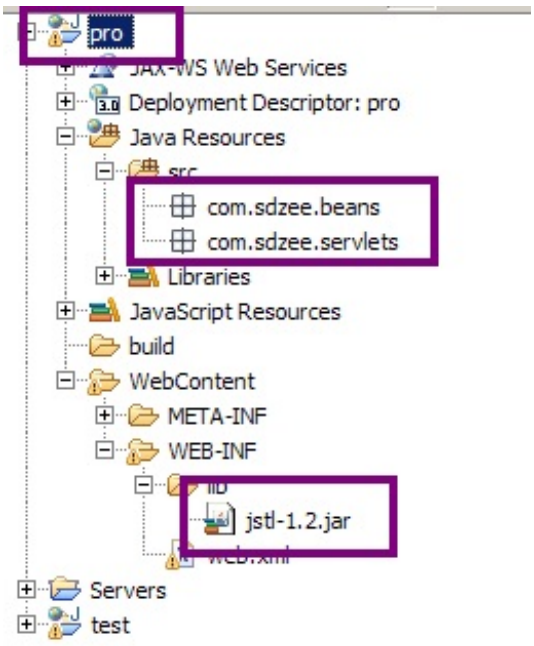
Modification du build-path



Modification du serveur de déploiement - Étape 1



Modification du serveur de déploiement - Étape 2



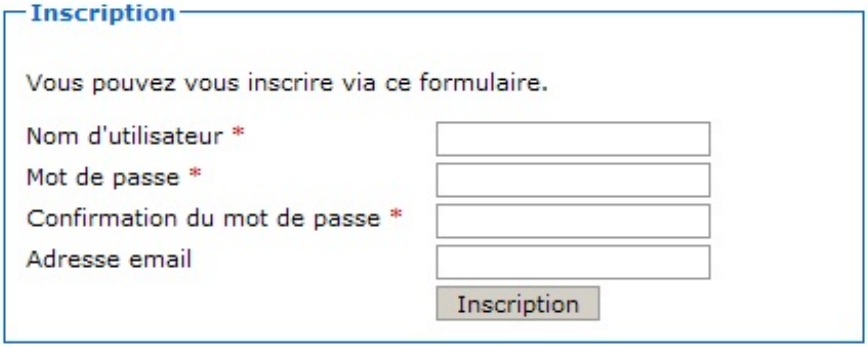
Arborescence finale du nouveau projet

N'oubliez pas les changements à effectuer sur le build-path et le serveur de déploiement, et remarquez bien sur le dernier écran l'ajout du .jar de la JSTL à notre projet, ainsi que la création des deux packages vides qui accueilleront par la suite nos servlets et beans.

Revenons maintenant à notre base. En apparence, elle consistera en une simple page web contenant un formulaire destiné à l'inscription du visiteur sur le site. Ce formulaire proposera :

- un champ texte recueillant le nom d'utilisateur du visiteur ;
- un champ texte recueillant son mot de passe ;
- un champ texte recueillant la confirmation de son mot de passe ;
- un champ texte recueillant son adresse mail.

Voici un aperçu du design que je vous propose de mettre en place :



Formulaire d'inscription

Si vous avez été assidu lors de vos premiers pas, vous devez vous souvenir que nous placerons dorénavant toujours nos pages JSP sous le répertoire **/WEB-INF** de l'application, et qu'à chaque JSP créée nous associerons une servlet. Je vous ai ainsi préparé une page JSP chargée de l'affichage du formulaire d'inscription, une feuille CSS pour sa mise en forme et une servlet pour l'accompagner.

JSP & CSS

Voici le code HTML de base du formulaire d'inscription que nous allons utiliser tout au long de cette partie :

Code : JSP - /WEB-INF/inscription.jsp

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Inscription</title>
        <link type="text/css" rel="stylesheet" href="form.css" />
    </head>
    <body>
<form method="post" action="inscription">
            <fieldset>
                <legend>Inscription</legend>
                <p>Vous pouvez vous inscrire via ce formulaire.</p>

                <label for="nom">Nom d'utilisateur <span
class="requis">*</span></label>
                <input type="text" id="nom" name="nom" value=""
size="20" maxlength="20" />
                <br />

                <label for="motdepasse">Mot de passe <span
class="requis">*</span></label>
                <input type="password" id="motdepasse"
name="motdepasse" value="" size="20" maxlength="20" />
                <br />

                <label for="confirmation">Confirmation du mot de
passe <span class="requis">*</span></label>
                <input type="password" id="confirmation"
name="confirmation" value="" size="20" maxlength="20" />
                <br />

                <label for="email">Adresse email</label>
                <input type="text" id="email" name="email" value=""
size="20" maxlength="60" />
                <br />

                <input type="submit" value="Inscription"
class="sansLabel" />
                <br />
            </fieldset>
        </form>
    </body>
</html>
```

Et voici le code de la feuille de style CSS accompagnant ce formulaire :

Code : CSS - /form.css

```
/* Général ------------------------------------------------------------
---------------------------- */

body, p, legend, label, input {
    font: normal 8pt verdana, helvetica, sans-serif;
}

fieldset {
    padding: 10px;
    border: 1px #0568CD solid;
}

legend {
    font-weight: bold;
    color: #0568CD;
}

/* Forms --------------------------------------------------------------
---------------------------- */
```

```
form label {
    float: left;
    width: 200px;
    margin: 3px 0px 0px 0px;
}

form input {
    margin: 3px 3px 0px 0px;
    border: 1px #999 solid;
}

form input.sansLabel {
    margin-left: 200px;
}

form .requis {
    color: #c00;
}
```

Contrairement à la page JSP, la feuille de style **ne doit pas être placée sous le répertoire /WEB-INF** ! Eh oui, vous devez vous souvenir que ce répertoire a la particularité de cacher ce qu'il contient à l'extérieur :

- dans le cas d'une page JSP c'est pratique, cela rend les pages inaccessibles directement depuis leur URL et nous permet de forcer le passage par une servlet.
- dans le cas de notre feuille CSS par contre, c'est une autre histoire ! Car ce que vous ne savez peut-être pas encore, c'est qu'en réalité lorsque vous accédez à une page web sur laquelle est attachée une feuille de style, votre navigateur va dans les coulisses envoyer une requête GET au serveur pour récupérer silencieusement cette feuille, en se basant sur l'URL précisée dans la balise `<link href="..." />`. Et donc fatalement, si vous placez le fichier sous **/WEB-INF** la requête va échouer, puisque le fichier sera caché du public et ne sera pas accessible par une URL.

De toute manière, dans la très grande majorité des cas le contenu d'une feuille CSS est fixe, il ne dépend pas de codes dynamiques et ne nécessite pas de pré-traitements depuis une servlet comme nous le faisons jusqu'à présent pour nos pages JSP. Nous pouvons donc rendre nos fichiers CSS accessibles directement aux navigateurs en les plaçant dans un répertoire public de l'application.

En l'occurrence, ici je l'ai placée directement à la racine de notre application, désignée par le répertoire **WebContent** dans Eclipse.

Retenez donc bien que tous les éléments fixes utilisés par vos pages JSP, comme les feuilles de style CSS, les feuilles de scripts Javascript ou encore les images, doivent être placés dans un répertoire public, et donc pas sous **/WEB-INF**.

Avant de mettre en place la servlet, penchons-nous un instant sur les deux attributs de la balise `<form>`.

La méthode

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

- **get** : les données transiteront par l'URL via des paramètres dans une requête HTTP GET. Je vous l'ai déjà expliqué, en raison des limitations de la taille d'une URL cette méthode est peu utilisée pour l'envoi de données.
- **post** : les données ne transiteront cette fois pas par l'URL mais dans le corps d'une requête HTTP POST, l'utilisateur ne les verra donc pas dans la barre d'adresse de son navigateur.

Malgré leur invisibilité apparente, les données envoyées via la méthode POST restent aisément accessibles, et ne sont donc pas plus sécurisées qu'avec la méthode GET : nous devrons donc **toujours vérifier la présence et la validité des paramètres avant de les utiliser**. La règle d'or à suivre lorsqu'on développe une application web, c'est de **ne jamais faire confiance à l'utilisateur**.

Voilà donc pourquoi nous utiliserons la plupart du temps la méthode POST pour envoyer les données de nos formulaires. En l'occurrence, nous avons bien précisé `<form method="post" ... >` dans le code de notre formulaire.

La cible

L'attribut **action** de la balise **<form>** permet de définir la page à laquelle seront envoyées les données du formulaire. Puisque nous suivons le modèle MVC, vous devez savoir que l'étape suivant l'envoi de données par l'utilisateur est **le contrôle**. Autrement dit, direction la servlet ! C'est donc l'URL permettant de joindre cette servlet, c'est-à-dire l'URL que vous allez spécifier dans sa déclaration dans le fichier web.xml, qui doit être précisée dans le champ **action** du formulaire. En l'occurrence, nous avons précisé **<form ... action="inscription">** dans le code de notre formulaire, nous devrons donc associer l'URL **/inscription** à notre servlet dans le mapping du fichier web.xml.

Lançons-nous maintenant, et créons une servlet qui s'occupera de récupérer les données envoyées et de les valider.

La servlet

Voici le code de la servlet accompagnant la JSP qui affiche le formulaire :

Code : Java - com.sdzee.servlets.Inscription

```
package com.sdzee.servlets;

import java.io.IOException;

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

public class Inscription extends HttpServlet {
 public static final String VUE = "/WEB-INF/inscription.jsp";

 public void doGet( HttpServletRequest request, HttpServletResponse
response ) throws ServletException, IOException{
  /* Affichage de la page d'inscription */
  this.getServletContext().getRequestDispatcher( VUE ).forward(
request, response );
 }
}
```

Pour le moment, elle se contente d'afficher notre page JSP à l'utilisateur lorsqu'elle reçoit une requête GET de sa part. Bientôt, elle sera également capable de gérer la réception d'une requête POST, lorsque l'utilisateur enverra les données de son formulaire !

Avant d'attaquer le traitement des données, voici enfin la configuration de notre servlet dans le fichier web.xml :

Code : XML - /WEB-INF/web.xml

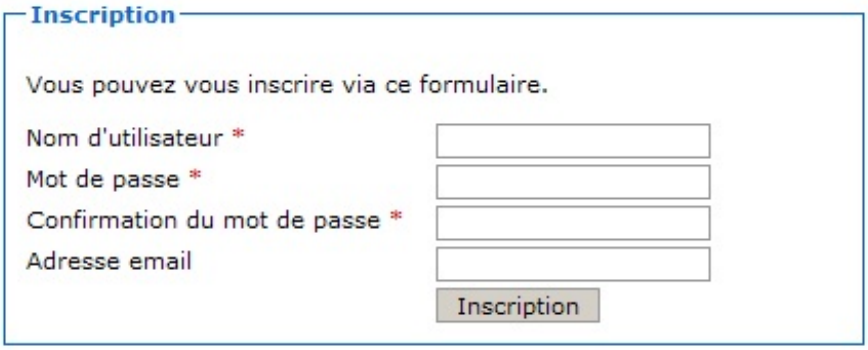
```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
 <servlet>
  <servlet-name>Inscription</servlet-name>
  <servlet-class>com.sdzee.servlets.Inscription</servlet-class>
 </servlet>

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

Souvenez-vous, l'adresse contenue dans le champ **<url-pattern>** est relative au contexte de l'application. Puisque nous avons nommé le contexte de notre projet **pro**, pour accéder à la JSP affichant le formulaire d'inscription il faut appeler l'URL suivante :

Code : URL

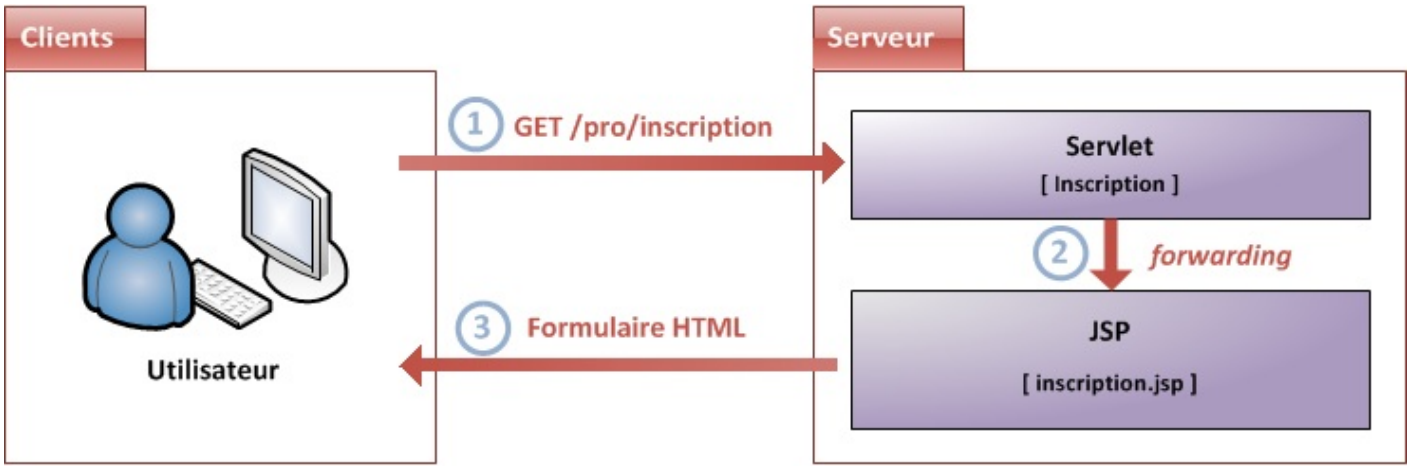
```
http://localhost:8080/pro/inscription
```

Vous devez si tout se passe bien visualiser le formulaire ci-dessous dans votre navigateur :



Formulaire d'inscription

Et voici sous forme d'un schéma ce que nous venons de réaliser :



L'envoi des données

Maintenant que nous avons accès à notre page d'inscription, nous pouvons saisir des données dans le formulaire et les envoyer au serveur. Remplissez les champs du formulaire avec un nom d'utilisateur, un mot de passe et une adresse mail de votre choix, et cliquez sur le bouton d'inscription. Voici la page que vous obtenez alors :

Code d'erreur HTTP 405

Eh oui, nous avons demandé un envoi des données du formulaire par la méthode POST, mais nous n'avons pas surchargé la méthode `doPost()` dans notre servlet, nous avons uniquement écrit une méthode `doGet()`. Par conséquent, **notre servlet n'est pas encore capable de traiter une requête POST** !

Nous savons donc ce qu'il nous reste à faire : il faut implémenter la méthode `doPost()`. Voici le code de notre servlet modifié :

Code : Java - com.sdzee.servlets.Inscription

```
package com.sdzee.servlets;

import java.io.IOException;

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

public class Inscription extends HttpServlet {
 public static final String VUE = "/WEB-INF/inscription.jsp";


 public void doGet( HttpServletRequest request, HttpServletResponse
response ) throws ServletException, IOException{
  /* Affichage de la page d'inscription */
  this.getServletContext().getRequestDispatcher( VUE ).forward(
request, response );
 }

public void doPost( HttpServletRequest request, HttpServletResponse
response ) throws ServletException, IOException{
/* Traitement des données du formulaire */
}
}
```

Maintenant que nous avons ajouté une méthode `doPost()`, nous pouvons dorénavant envoyer les données de notre formulaire, il n'y aura plus d'erreur HTTP !
Par contre, notre méthode `doPost()` étant vide, nous obtenons bien évidemment une simple page blanche en retour...

Contrôle : côté servlet

Maintenant que notre formulaire est accessible à l'utilisateur et que la servlet en charge de son contrôle est en place, nous pouvons nous attaquer à la vérification des données envoyées par le client.

Que souhaitons-nous vérifier ?

Nous travaillons sur un formulaire d'inscription qui contient quatre champs de type input, cela ne va pas être bien compliqué. Voici ce que je vous propose de vérifier :

- que le champ obligatoire nom n'est pas vide et qu'il contient au moins 3 caractères ;
- que les champs obligatoires mot de passe et confirmation ne sont pas vides, qu'ils contiennent au moins 3 caractères, et qu'ils sont égaux ;
- que le champ facultatif email, s'il est rempli, contienne une adresse mail valide.

Nous allons confier ces tâches à trois méthodes distinctes :

- une méthode `validationNom()`, chargée de valider le nom d'utilisateur saisi ;
- une méthode `validationMotsDePasse()`, chargée de valider les mots de passe saisis ;
- une méthode `validationEmail()`, chargée de valider l'adresse mail saisie.

Voici donc le code modifié de notre servlet, impliquant la méthode `doPost()`, des nouvelles constantes et les méthodes de validation créées pour l'occasion, en charge de récupérer le contenu des champs du formulaire et de les faire valider :

Code : Java - com.sdzee.servlets.Inscription

```
package com.sdzee.servlets;

import java.io.IOException;

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

```
public class Inscription extends HttpServlet {
 public static final String VUE = "/WEB-INF/inscription.jsp";
public static final String CHAMP_NOM = "nom";
public static final String CHAMP_PASS = "motdepasse";
public static final String CHAMP_CONF = "confirmation";
public static final String CHAMP_EMAIL = "email";

 public void doGet( HttpServletRequest request, HttpServletResponse
response ) throws ServletException, IOException{
  /* Affichage de la page d'inscription */
  this.getServletContext().getRequestDispatcher( VUE ).forward(
request, response );
 }

 public void doPost( HttpServletRequest request, HttpServletResponse
response ) throws ServletException, IOException{
  /* Récupération des champs du formulaire. */
String nom = request.getParameter( CHAMP_NOM );
String motDePasse = request.getParameter( CHAMP_PASS );
String confirmation = request.getParameter( CHAMP_CONF );
String email = request.getParameter( CHAMP_EMAIL );

  try {
   validationNom( nom );
   validationMotsDePasse( motDePasse, confirmation );
   validationEmail( email );
  } catch (Exception e) {
   /* Gérer les erreur de validation ici. */
  }
 }

 private void validationNom( String nom ) throws Exception{}
 private void validationMotsDePasse( String motDePasse, String
confirmation ) throws Exception{}
 private void validationEmail( String email ) throws Exception{}
}
```

J'ai surligné dans le code la partie en charge de la récupération des champs du formulaire : il s'agit tout simplement d'appels à la méthode `getParameter()`. Il nous reste maintenant à implémenter nos trois dernières méthodes de validation, qui sont vides pour le moment. Voilà un premier jet de ce que cela pourrait donner :

Code : Java - com.sdzee.servlets.Inscription

```
...

/**
* Valide le nom d'utilisateur saisi.
*/
private void validationNom( String nom ) throws Exception{
    if (nom != null && nom.trim.length() != 0) {
        if (nom.length() < 3) {
            throw new Exception("Le nom d'utilisateur doit contenir
au moins 3 caractères.");
        }
    } else {
        throw new Exception("Merci d'entrer un nom d'utilisateur.");
    }
}

/**
* Valide les mots de passe saisis.
*/
private void validationMotsDePasse( String motDePasse, String
confirmation ) throws Exception{
    if (motDePasse != null && motDePasse.trim.length() != 0 &&
confirmation != null && confirmation.trim.length() != 0) {
```

```
            if (!motDePasse.equals(confirmation)) {
                throw new Exception("Les mots de passe entrés sont
différents, merci de les saisir à nouveau.");
            } else if (motDePasse.length() < 3) {
                throw new Exception("Les mots de passe doivent contenir
au moins 3 caractères.");
            }
        } else {
            throw new Exception("Merci de saisir et confirmer votre mot
de passe.");
        }
    }

    /**
    * Valide l'adresse mail saisie.
    */
    private void validationEmail( String email ) throws Exception{
        if (email != null && email.trim.length() != 0 &&
!email.matches("([^.@]+)(\\.[^.@]+)*@([^.@]+\\.)+([^.@]+)")) {
            throw new Exception("Merci de saisir une adresse mail
valide.");
        }
    }
```

Je ne détaille pas le code de ces trois courtes méthodes. Si vous ne comprenez pas leur fonctionnement, vous devez impérativement revenir par vous-mêmes sur ces notions basiques du langage Java avant de continuer ce tutoriel. Le forum Java du site du zéro est là pour vous, si vous avez besoin d'aide ou d'explications.

J'ai ici fait en sorte que dans chaque méthode, lorsqu'une erreur de validation se produit, le code envoie une exception contenant un message explicitant l'erreur. Ce n'est pas la seule solution envisageable, mais c'est une solution qui a le mérite de tirer parti de la gestion des exceptions en Java. À ce niveau, un peu de réflexion sur la conception de notre système de validation s'impose ici :

Que faire de ces exceptions envoyées ?

Ou en d'autres termes, quelles informations souhaitons-nous renvoyer à l'utilisateur en cas d'erreur ? Pour un formulaire d'inscription, a priori nous aimerions bien que l'utilisateur soit au courant du succès ou de l'échec de l'inscription, et en cas d'échec qu'il soit informé des erreurs commises sur les champs posant problème.

Comment procéder ? Là encore, il y a bien des manières de faire. Je vous propose ici le mode de fonctionnement suivant :

- une chaîne **resultat** contenant le statut final de la validation des champs ;
- une map **erreurs** contenant les éventuels messages d'erreur renvoyés par nos différentes méthodes se chargeant de la validation des champs. Une `HashMap` convient très bien dans ce cas d'utilisation : en l'occurrence, la clé sera le nom du champ et la valeur sera le message d'erreur correspondant.

Mettons tout cela en musique, toujours dans la méthode `doPost()` de notre servlet :

Code : Java - com.sdzee.servlets.Inscription

```
package com.sdzee.servlets;

import java.io.IOException;
import java.util.HashMap;
import java.util.Map;

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

public class Inscription extends HttpServlet {
 public static final String VUE = "/WEB-INF/inscription.jsp";
```

```
    public static final String CHAMP_NOM = "nom";
    public static final String CHAMP_PASS = "motdepasse";
    public static final String CHAMP_CONF = "confirmation";
    public static final String CHAMP_EMAIL = "email";
public static final String ATT_ERREURS = "erreurs";
public static final String ATT_RESULTAT = "resultat";

    public void doGet( HttpServletRequest request, HttpServletResponse
response ) throws ServletException, IOException{
      /* Affichage de la page d'inscription */
      this.getServletContext().getRequestDispatcher( VUE ).forward(
request, response );
    }

    public void doPost( HttpServletRequest request, HttpServletResponse
response ) throws ServletException, IOException{
String resultat;
Map<String, String> erreurs = new HashMap<String, String>();

      /* Récupération des champs du formulaire. */
      String nom = request.getParameter( CHAMP_NOM );
      String motDePasse = request.getParameter( CHAMP_PASS );
      String confirmation = request.getParameter( CHAMP_CONF );
      String email = request.getParameter( CHAMP_EMAIL );

      /* Validation du champ nom. */
      try {
       validationNom( nom );
      } catch (Exception e) {
erreurs.put( CHAMP_NOM, e.getMessage() );
      }

      /* Validation des champs mot de passe et confirmation. */
      try {
       validationMotsDePasse( motDePasse, confirmation );
      } catch (Exception e) {
erreurs.put( CHAMP_PASS, e.getMessage() );
      }

      /* Validation du champ email. */
      try {
       validationEmail( email );
      } catch (Exception e) {
erreurs.put( CHAMP_EMAIL, e.getMessage() );
      }

      /* Initialisation du résultat global de la validation. */
if( erreurs.isEmpty() ){
resultat = "Succès de l'inscription.";
} else {
resultat = "Échec de l'inscription.";
}

      /* Stockage du résultat et des messages d'erreur dans l'objet
request */
request.setAttribute( ATT_ERREURS, erreurs );
request.setAttribute( ATT_RESULTAT, resultat );

      /* Transmission de la paire d'objets request/response à notre JSP
*/
      this.getServletContext().getRequestDispatcher( VUE ).forward(
request, response );
    }

    ...
}
```

J'ai surligné les modifications importantes du code pour que vous visualisiez bien ce qui intervient dans ce processus de gestion

des exceptions :

- chaque appel à une méthode de validation d'un champ est entourée d'un bloc **try/catch** ;
- à chaque entrée dans un **catch**, c'est-à-dire dès lors qu'une méthode de validation envoie une exception, on ajoute à la map **erreurs** le message de description inclus dans l'exception courante, avec pour clé l'intitulé du champ du formulaire concerné ;
- le message **resultat** contenant le résultat global de la validation est initialisé selon que la map **erreurs** contient des messages d'erreurs ou non ;
- les deux objets **erreurs** et **resultat** sont enfin inclus en tant qu'attributs à la requête avant l'appel final à la vue.

Le contrôle des données dans notre servlet est maintenant fonctionnel : avec les données que nous transmettons à notre JSP, nous pouvons y déterminer si des erreurs de validation ont eu lieu et sur quels champs.

Affichage : côté JSP

Ce qu'il nous reste maintenant à réaliser, c'est l'affichage de nos différents messages au sein de la page JSP, après que l'utilisateur a saisi et envoyé ses données. Voici ce que je vous propose :

1. En cas d'erreur, affichage du message d'erreur à côté de chacun des champs concernés.
2. Ré-affichage dans les champs **<input/>** des données auparavant saisies par l'utilisateur.
3. Affichage du résultat de l'inscription en bas du formulaire.

1. Afficher les messages d'erreurs

L'attribut **erreurs** que nous recevons de la servlet ne contient des messages concernant les différents champs de notre formulaire que si des erreurs ont été rencontrées lors de la validation de leur contenu, c'est-à-dire uniquement si des exceptions ont été envoyées. Ainsi, il nous suffit d'afficher les entrées de la map correspondant à chacun des champs **nom**, **motdepasse**, **confirmation** et **email** :

Code : JSP - /WEB-INF/inscription.jsp

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Inscription</title>
        <link type="text/css" rel="stylesheet" href="form.css" />
    </head>
    <body>
        <form method="post" action="inscription">
     <fieldset>
                <legend>Inscription</legend>
                <p>Vous pouvez vous inscrire via ce formulaire.</p>

                <label for="nom">Nom d'utilisateur <span
class="requis">*</span></label>
                <input type="text" id="nom" name="nom" value=""
size="20" maxlength="20" />
<span class="erreur">${erreurs['nom']}</span>
                <br />

                <label for="motdepasse">Mot de passe <span
class="requis">*</span></label>
                <input type="password" id="motdepasse"
name="motdepasse" value="" size="20" maxlength="20" />
<span class="erreur">${erreurs['motdepasse']}</span>
                <br />

                <label for="confirmation">Confirmation du mot de
passe <span class="requis">*</span></label>
                <input type="password" id="confirmation"
name="confirmation" value="" size="20" maxlength="20" />
<span class="erreur">${erreurs['confirmation']}</span>
                <br />

                <label for="email">Adresse email</label>
```

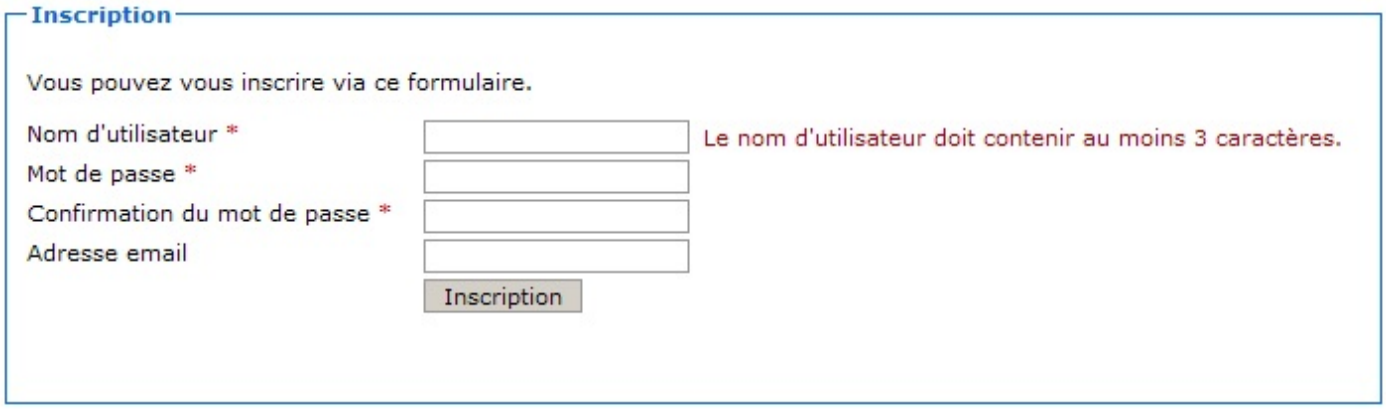
```
                    <input type="email" id="email" name="email" value=""
size="20" maxlength="60" />
<span class="erreur">${erreurs['email']}</span>
                    <br />

                    <input type="submit" value="Inscription"
class="sansLabel" />
                    <br />
                </fieldset>
            </form>
        </body>
</html>
```

Vous retrouvez ici l'utilisation des crochets pour accéder aux entrées de la map, comme nous l'avions déjà fait lors de notre apprentissage de la JSTL. De cette manière, si aucun message ne correspond dans la map à un champ du formulaire donné, c'est qu'il n'y a pas eu d'erreur lors de sa validation côté serveur et dans ce cas la balise **** sera vide et aucun message ne sera affiché à l'utilisateur. Comme vous pouvez le voir, j'en ai profité pour ajouter un style à notre feuille **form.css**, afin de mettre en avant les erreurs :

Code : CSS

```
form .erreur {
    color: #900;
}
```

Vous pouvez dorénavant faire le test : remplissez votre formulaire avec des données erronées (un nom d'utilisateur trop court ou des mots de passe différents, par exemple) et contemplez le résultat ! Ci-dessous, le rendu attendu lorsque vous entrez un nom d'utilisateur trop court :



Erreur de validation du formulaire

2. Ré-afficher les données saisies par l'utilisateur

Comme vous le constatez sur cette dernière image, les données saisies par l'utilisateur avant validation du formulaire disparaissent des champs après validation. En ce qui concerne les champs mot de passe et confirmation, c'est très bien ainsi : après une erreur de validation, il est courant de demander à l'utilisateur de saisir à nouveau cette information sensible. Dans le cas du nom et de l'adresse mail par contre, ce n'est vraiment pas ergonomique et nous allons tâcher de les faire réapparaître. Pour cette étape, nous pourrions être tentés de simplement ré-afficher directement ce qu'a saisi l'utilisateur dans chacun des champs "value" des **<input/>** du formulaire. En effet, nous savons que ces données sont directement accessibles via l'objet implicite **param**, qui donne accès aux paramètres de la requête HTTP. Le problème, et c'est un problème de taille, c'est qu'en procédant ainsi nous nous exposons aux failles XSS. Souvenez-vous, je vous en ai déjà parlé lorsque nous avons découvert la balise **<c:out/>** de la JSTL !

Quel est le problème exactement ?

Ok... Puisque vous semblez amnésique et sceptique, faisons comme si de rien n'était, et ré-affichons le contenu des paramètres

de la requête HTTP (c'est-à-dire le contenu saisi par l'utilisateur dans les champs **<input/>** du formulaire) en y accédant directement via l'objet implicite **param** :

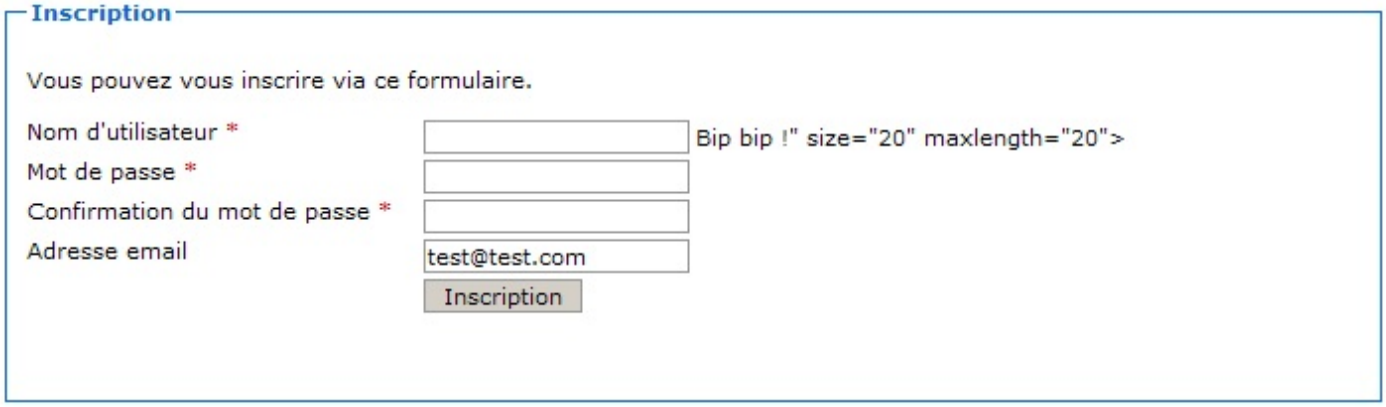
Code : JSP - /WEB-INF/inscription.jsp

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Inscription</title>
        <link type="text/css" rel="stylesheet" href="form.css" />
    </head>
    <body>
        <form method="post" action="inscription">
            <fieldset>
                <legend>Inscription</legend>
                <p>Vous pouvez vous inscrire via ce formulaire.</p>

                <label for="nom">Nom d'utilisateur <span
class="requis">*</span></label>
<input type="text" id="nom" name="nom" value="${param.nom}"
size="20" maxlength="20" />
                <span class="erreur">${erreurs['nom']}</span>
                <br />

                <label for="motdepasse">Mot de passe <span
class="requis">*</span></label>
                <input type="password" id="motdepasse"
name="motdepasse" value="" size="20" maxlength="20" />
                <span class="erreur">${erreurs['motdepasse']}</span>
                <br />

                <label for="confirmation">Confirmation du mot de
passe <span class="requis">*</span></label>
                <input type="password" id="confirmation"
name="confirmation" value="" size="20" maxlength="20" />
                <span
class="erreur">${erreurs['confirmation']}</span>
                <br />

                <label for="email">Adresse email</label>
<input type="email" id="email" name="email" value="${param.email}"
size="20" maxlength="60" />
                <span class="erreur">${erreurs['email']}</span>
                <br />

                <input type="submit" value="Inscription"
class="sansLabel" />
                <br />
            </fieldset>
        </form>
    </body>
</html>
```

Faites alors à nouveau le test en remplissant et validant votre formulaire. Dorénavant, les données que vous avez entrées sont bien présentes dans les champs du formulaire après validation :

Erreur de validation du formulaire avec affichage des données saisies

En apparence ça tient la route, mais je vous ai lourdement averti : **en procédant ainsi, votre code est vulnérable aux failles XSS**. Vous voulez un exemple ? Remplissez le champ nom d'utilisateur par le contenu suivant : ">Bip bip ! . Validez ensuite votre formulaire, et contemplez alors ce triste et désagréable résultat :



Erreur de validation du formulaire avec faille XSS

Que s'est-il passé ?

Une faille XSS, pardi ! Eh oui, côté serveur, le contenu que vous avez saisi dans le champ du formulaire a été copié tel quel dans le code généré par notre JSP. Il a ensuite été interprété par le navigateur côté client, qui a alors naturellement considéré que le guillemet (") et le chevron (>) contenus en début de saisie correspondaient à la fermeture de la balise **<input/>** ! Si vous êtes encore dans le flou, voyez plutôt le code HTML produit sur la ligne posant problème :

Code : HTML

```
<input type="text" id="nom" name="nom" value="">Bip bip !" size="20"
maxlength="20" />
```

Vous devez maintenant comprendre le problème : le contenu de notre champ a été copié/collé tel quel dans la source de notre fichier HTML final, lors de l'interprétation par le serveur de l'expression EL que nous avons mise en place (c'est-à-dire `${param.nom}`). Et logiquement, puisque le navigateur ferme la balise **<input/>** prématurément, notre joli formulaire s'en retrouve défiguré. Alors certes, ici ce n'est pas bien grave, je n'ai fait que casser l'affichage de la page. Mais vous devez savoir qu'en utilisant ce type de failles, il est possible de causer bien plus de dommages, notamment en injectant du code Javascript dans la page à l'insu du client.

Je vous le répète : la règle d'or, c'est de **ne jamais faire confiance à l'utilisateur**.

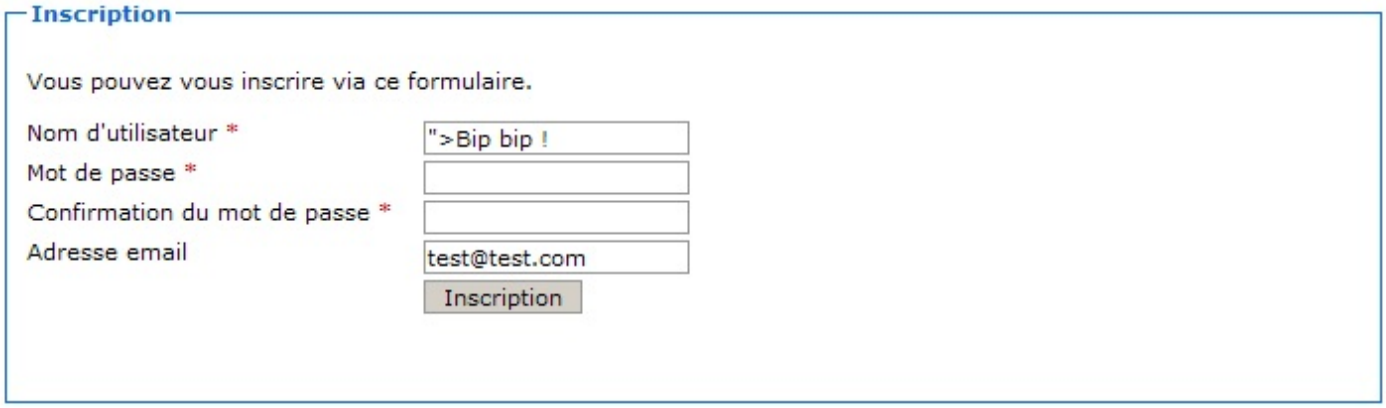
Pour pallier ce problème, il suffit d'utiliser la balise **<c:out/>** de la JSTL Core pour procéder à l'affichage des données.

Voici ce que donne alors le code modifié de notre JSP :

Code : JSP - /WEB-INF/inscription.jsp

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Inscription</title>
        <link type="text/css" rel="stylesheet" href="form.css" />
    </head>
    <body>
        <form method="post" action="inscription">
            <fieldset>
                <legend>Inscription</legend>
                <p>Vous pouvez vous inscrire via ce formulaire.</p>

                <label for="nom">Nom d'utilisateur <span
class="requis">*</span></label>
<input type="text" id="nom" name="nom" value="<c:out
value="${param.nom}"/>" size="20" maxlength="20" />
                <span class="erreur">${erreurs['nom']}</span>
                <br />

                <label for="motdepasse">Mot de passe <span
class="requis">*</span></label>
                <input type="password" id="motdepasse"
name="motdepasse" value="" size="20" maxlength="20" />
                <span class="erreur">${erreurs['motdepasse']}</span>
                <br />

                <label for="confirmation">Confirmation du mot de
passe <span class="requis">*</span></label>
                <input type="password" id="confirmation"
name="confirmation" value="" size="20" maxlength="20" />
                <span
class="erreur">${erreurs['confirmation']}</span>
                <br />

                <label for="email">Adresse email</label>
<input type="email" id="email" name="email" value="<c:out
value="${param.email}"/>" size="20" maxlength="60" />
                <span class="erreur">${erreurs['email']}</span>
                <br />

                <input type="submit" value="Inscription"
class="sansLabel" />
                <br />
            </fieldset>
        </form>
    </body>
</html>
```

La balise **<c:out/>** se chargeant par défaut d'échapper les caractères spéciaux, le problème est réglé. Notez l'ajout de la directive **taglib** en haut de page, pour que notre JSP puisse utiliser les balises de la JSTL Core. Faites à nouveau le test avec le nom d'utilisateur précédent, et vous obtiendrez bien cette fois :



Erreur de validation du formulaire sans faille XSS

Dorénavant, l'affichage n'est plus cassé, et si nous regardons le code HTML généré, nous observons bien la transformation du " et du > en leurs codes HTML respectifs par la balise **<c:out/>** :

Code : HTML

```
<input type="text" id="nom" name="nom" value="&#034;&gt;Bip bip !"
size="20" maxlength="20" />
```

Ainsi, le navigateur de l'utilisateur reconnaît que les caractères " et > font bien partie du contenu du champ, et qu'ils ne doivent pas être interprétés en tant qu'éléments de fermeture de la balise **<input/>** !

À l'avenir, n'oubliez jamais ceci : protégez toujours les données que vous affichez à l'utilisateur !

3. Afficher le résultat final de l'inscription

Il ne nous reste maintenant qu'à confirmer le statut de l'inscription. Pour ce faire, il suffit d'afficher l'entrée **resultat** de la map dans laquelle nous avons initialisé le message :

Code : JSP - /WEB-INF/inscription.jsp

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Inscription</title>
        <link type="text/css" rel="stylesheet" href="form.css" />
    </head>
    <body>
        <form method="post" action="inscription">
            <fieldset>
                <legend>Inscription</legend>
                <p>Vous pouvez vous inscrire via ce formulaire.</p>

                <label for="nom">Nom d'utilisateur <span
class="requis">*</span></label>
                <input type="text" id="nom" name="nom" value="<c:out
value="${param.nom}"/>" size="20" maxlength="20" />
                <span class="erreur">${erreurs['nom']}</span>
                <br />

                <label for="motdepasse">Mot de passe <span
class="requis">*</span></label>
                <input type="password" id="motdepasse"
```

```
name="motdepasse" value="" size="20" maxlength="20" />
                <span class="erreur">${erreurs['motdepasse']}</span>
                <br />

                <label for="confirmation">Confirmation du mot de
passe <span class="requis">*</span></label>
                <input type="password" id="confirmation"
name="confirmation" value="" size="20" maxlength="20" />
                <span
class="erreur">${erreurs['confirmation']}</span>
                <br />

                <label for="email">Adresse email</label>
                <input type="email" id="email" name="email"
value="<c:out value="${param.email}"/>" size="20" maxlength="60" />
                <span class="erreur">${erreurs['email']}</span>
                <br />

                <input type="submit" value="Inscription"
class="sansLabel" />
                <br />

<p class="${empty erreurs ? 'succes' : 'erreur'}">${resultat}</p>
            </fieldset>
        </form>
    </body>
</html>
```

Vous remarquez ici l'utilisation d'un test ternaire sur notre map **erreurs** au sein de la première expression EL mise en place, afin de déterminer la classe CSS à appliquer au paragraphe : si la map **erreurs** est vide, alors cela signifie qu'aucune erreur n'a eu lieu et on utilise le style nommé **succes**, et sinon on utilise le style **erreur**. J'en ai en effet profité pour ajouter un dernier style à notre feuille **form.css**, pour mettre en avant le succès de l'inscription :

Code : CSS

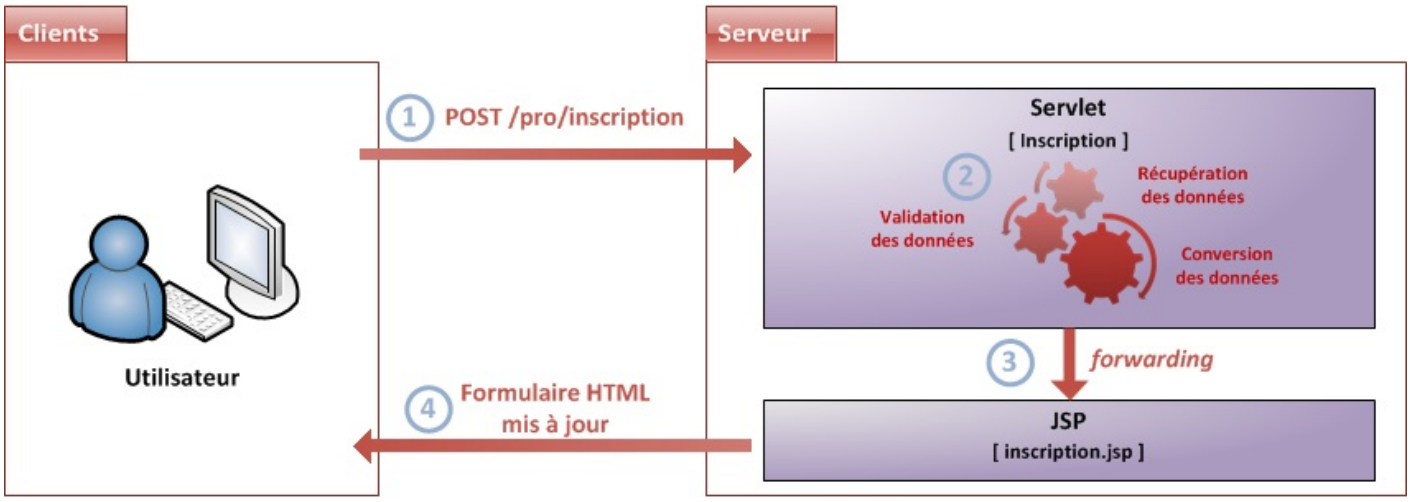
```
form .succes {
    color: #090;
}
```

Et sous vos yeux ébahis, voici le résultat final en cas de succès et d'erreur :

Succes de la validation du formulaire

Erreur dans la validation du formulaire

Pour terminer, voici sous forme d'un schéma ce que nous venons de réaliser :



Maintenant que notre base est fonctionnelle, prenons le temps de réfléchir à ce que nous avons créé, et à ce que MVC nous impose de modifier !

Formulaires : à la mode MVC

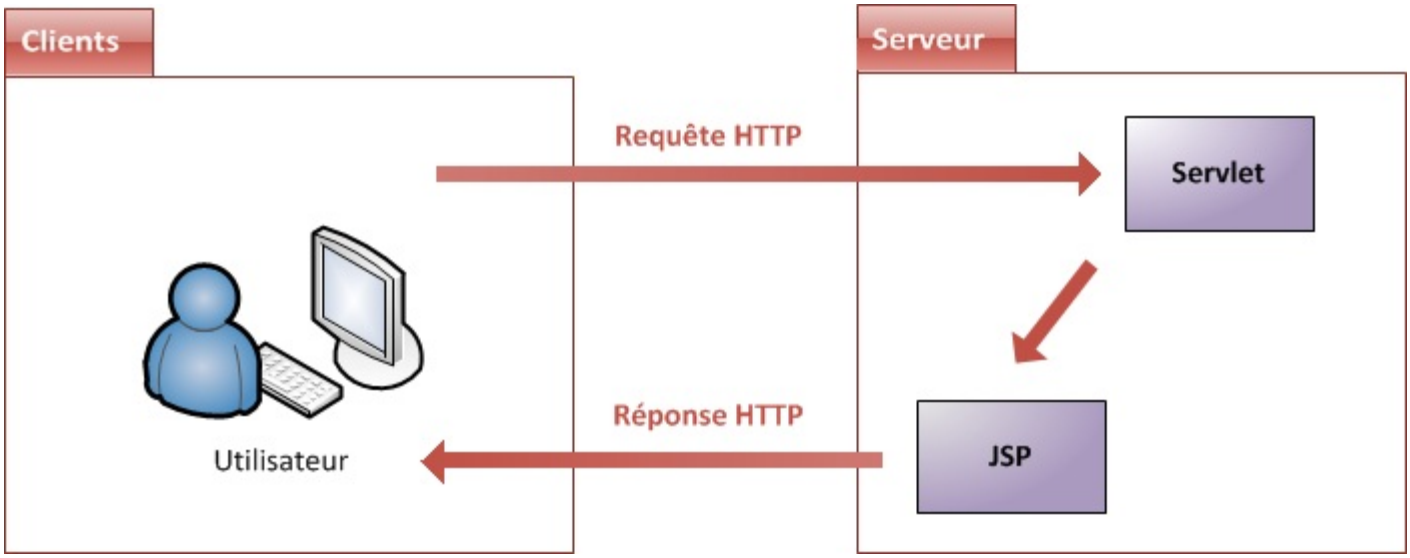
Le modèle MVC est très clair sur ce point : c'est le modèle qui doit s'occuper de traiter les données. Le contrôleur doit avoir pour unique but d'aiguiller les requêtes entrantes et d'appeler les éventuels traitements correspondants. Nous devons donc nous pencher sur la conception que nous venons de mettre en place afin d'en identifier les défauts, et la rectifier afin de suivre les recommandations MVC.

Analyse de notre conception

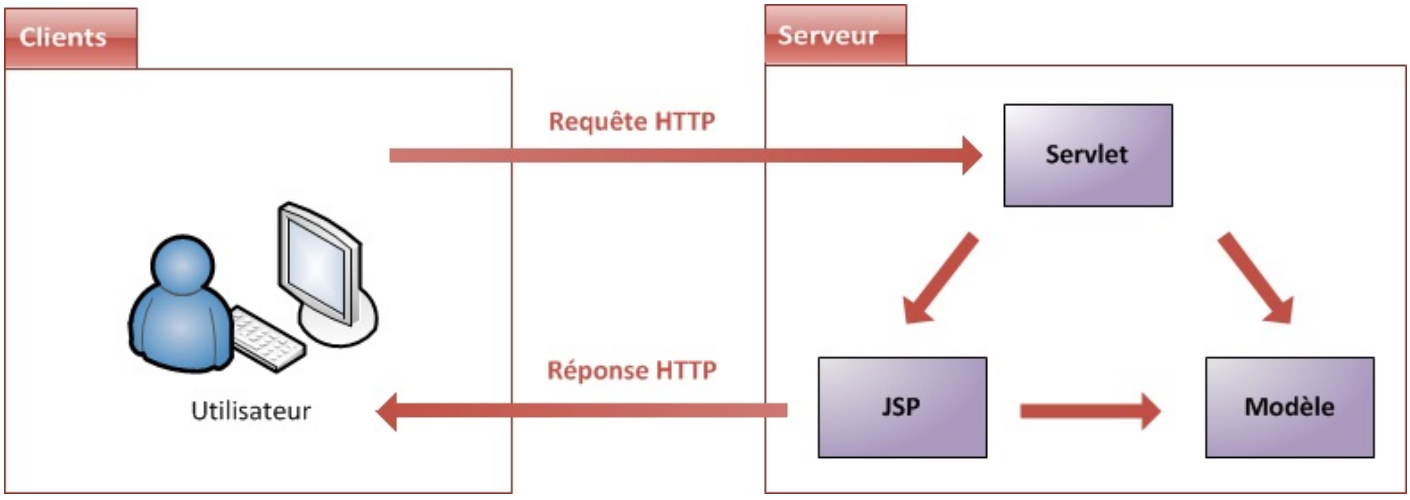
La base que nous avons réalisée souffre de plusieurs maux :

- la récupération et le traitement des données sont effectués directement au sein de la servlet. Or nous savons que d'après MVC, la servlet est un contrôleur, et n'est donc pas censée intervenir directement sur les données, elle doit uniquement aiguiller les requêtes entrantes vers les traitements correspondants ;
- aucun modèle (bean ou objet métier) n'intervient dans le système mis en place ! Pourtant, nous savons que d'après MVC, les données sont représentées dans le modèle par des objets...

Pour rappel, voici le schéma représentant ce que nous avons créé :



Et voici le schéma représentant ce vers quoi nous souhaitons parvenir :



Nous allons donc reprendre notre système d'inscription pour y mettre en place un modèle :

1. création d'un bean qui enregistre les données saisies et validées ;
2. création d'un objet métier comportant les méthodes de récupération/conversion/validation des contenus des champs du formulaire ;
3. modification de la servlet pour qu'elle n'intervienne plus directement sur les données de la requête, mais aiguille simplement la requête entrante ;
4. modification de la JSP pour qu'elle s'adapte au modèle fraîchement créé.

Création du modèle

L'utilisateur

Pour représenter un utilisateur dans notre modèle, nous allons naturellement créer un bean nommé **Utilisateur** et placé dans le package `com.sdzee.beans`, contenant trois champs de type `String` : **nom**, **motDePasse** et **email**. Si vous ne vous souvenez plus des règles à respecter lors de la création d'un bean, n'hésitez pas à relire le chapitre qui y est dédié. Voici le résultat attendu :

Code : Java - com.sdzee.beans.Utilisateur

```
package com.sdzee.beans;

public class Utilisateur {

    private String nom;
    private String motDePasse;
    private String email;

    public void setNom(String nom) {
 this.nom = nom;
    }
    public String getNom() {
 return nom;
    }

    public void setMotDePasse(String motDePasse) {
 this.motDePasse = motDePasse;
    }
    public String getMotDePasse() {
 return motDePasse;
    }

    public void setEmail(String email) {
 this.email = email;
    }
    public String getEmail() {
 return email;
    }
}
```

C'est tout ce dont nous avons besoin pour représenter les données d'un utilisateur dans notre application.

Dans notre formulaire, il y a un quatrième champ : la confirmation du mot de passe. Pourquoi ne stockons-nous pas cette information dans notre bean ?

Tout simplement parce que ce bean ne représente pas le formulaire, il représente un utilisateur. Et un utilisateur final, il a un mot de passe et point barre : la confirmation est une information temporaire propre à l'étape d'inscription seulement, il n'y a par conséquent aucun intérêt à la stocker dans notre modèle.

Le formulaire

Maintenant, il nous faut créer dans notre modèle un objet "métier", c'est-à-dire un objet chargé de traiter les données envoyées par le client via le formulaire. Cet objet va dans notre cas devoir contenir :

- les constantes identifiant les champs du formulaire ;
- la chaîne **resultat** et la map **erreurs** que nous avions mises en place dans la servlet ;
- la logique de validation que nous avions utilisée dans la méthode `doPost()` de la servlet ;
- les trois méthodes de validation que nous avions créées dans la servlet.

Nous allons donc déporter la majorité du code que nous avions écrit dans notre servlet dans cet objet métier, en l'adaptant afin de le faire interagir avec notre bean fraîchement créé. Nous allons nommer cet objet **InscriptionForm**, et le placer dans un nouveau package `com.sdzee.forms` :

Code : Java - com.sdzee.forms.InscriptionForm

```
package com.sdzee.forms;

import java.util.HashMap;
import java.util.Map;

import javax.servlet.http.HttpServletRequest;

public final class InscriptionForm {
    private static final String CHAMP_NOM    = "nom";
    private static final String CHAMP_PASS   = "motdepasse";
    private static final String CHAMP_CONF   = "confirmation";
    private static final String CHAMP_EMAIL  = "email";

    private String              resultat;
    private Map<String, String> erreurs      = new HashMap<String,
String>();

public String getResultat() {
return resultat;
}

public Map<String, String> getErreurs() {
return erreurs;
}

    public Utilisateur inscrireUtilisateur( HttpServletRequest
request ) {
String nom = getValeurChamp( request, CHAMP_NOM );
String motDePasse = getValeurChamp( request, CHAMP_PASS );
String confirmation = getValeurChamp( request, CHAMP_CONF );
String email = getValeurChamp( request, CHAMP_EMAIL );

Utilisateur utilisateur = new Utilisateur();

        try {
            validationNom( nom );
        } catch ( Exception e ) {
setErreur( CHAMP_NOM, e.getMessage() );
        }
utilisateur.setNom( nom );

        try {
            validationMotsDePasse( motDePasse, confirmation );
        } catch ( Exception e ) {
setErreur( CHAMP_PASS, e.getMessage() );
setErreur( CHAMP_CONF, null );
        }
utilisateur.setMotDePasse( motDePasse );

        try {
            validationEmail( email );
        } catch ( Exception e ) {
setErreur( CHAMP_EMAIL, e.getMessage() );
        }
utilisateur.setEmail( email );

        if ( erreurs.isEmpty() ) {
            resultat = "Succès de l'inscription.";
        } else {
            resultat = "Échec de l'inscription.";
        }

return utilisateur;
    }

    private void validationNom( String nom ) throws Exception {
        if ( nom != null ) {
```

```
            if ( nom.length() < 3 ) {
                throw new Exception( "Le nom d'utilisateur doit
contenir au moins 3 caractères." );
            }
        } else {
            throw new Exception( "Merci d'entrer un nom
d'utilisateur." );
        }
    }

    private void validationMotsDePasse( String motDePasse, String
confirmation ) throws Exception {
        if ( motDePasse != null && confirmation != null ) {
            if ( !motDePasse.equals( confirmation ) ) {
                throw new Exception( "Les mots de passe entrés sont
différents, merci de les saisir à nouveau." );
            } else if ( motDePasse.length() < 3 ) {
                throw new Exception( "Les mots de passe doivent
contenir au moins 3 caractères." );
            }
        } else {
            throw new Exception( "Merci de saisir et confirmer votre
mot de passe." );
        }
    }

    private void validationEmail( String email ) throws Exception {
        if ( email != null && !email.matches(
"([^.@]+)(\\.[^.@]+)*@([^.@]+\\.)+([^.@]+)" ) ) {
            throw new Exception( "Merci de saisir une adresse mail
valide." );
        }
    }

    /*
* Ajoute un message correspondant au champ spécifié à la map des
erreurs.
*/
private void setErreur( String champ, String message ) {
erreurs.put( champ, message );
}

    /*
* Méthode utilitaire qui retourne null si un champ est vide, et son
contenu
* sinon.
*/
private static String getValeurChamp( HttpServletRequest request,
String nomChamp ) {
String valeur = request.getParameter( nomChamp );
if ( valeur == null || valeur.trim().length() == 0 ) {
return null;
} else {
return valeur;
}
}
}
```

J'ai surligné pour vous dans le code les ajouts qui ont été effectués. Vous remarquerez qu'au final, il y a très peu de changements :

- ajout de *getters* publics pour les attributs privés **resultat** et **erreurs**, afin de les rendre accessibles depuis notre JSP via des expressions EL ;
- la logique de validation a été regroupée dans une méthode `inscrireUtilisateur()`, qui retourne un bean **Utilisateur** ;
- la méthode utilitaire `getValeurChamp()` se charge désormais de vérifier si le contenu d'un champ est vide ou non, ce qui nous permet aux lignes 65, 75 et 87 de ne plus avoir à effectuer la vérification sur la longueur des chaînes, et de simplement vérifier si elles sont à **null** ;

- dans les blocs **catch** englobant la validation de chaque champ du formulaire, nous utilisons désormais une méthode `setErreur()` qui se charge de mettre à jour la map **erreurs** en cas d'envoi d'une exception ;
- après la validation de chaque champ du formulaire, nous procédons dorénavant à l'initialisation de la propriété correspondante dans le bean **Utilisateur**, peu importe le résultat de la validation (lignes 38, 46 et 53).

Voilà tout ce qu'il est nécessaire de mettre en place dans notre modèle. Prochaine étape : il nous faut nettoyer notre servlet !

Le découpage en méthodes via `setErreur()` et `getValeurChamp()` n'est pas une obligation, mais puisque nous avons déplacé notre code dans un objet métier, autant en profiter pour coder un peu plus proprement.

Reprise de la servlet

Puisque nous avons déporté la majorité du code présent dans notre servlet vers le modèle, nous pouvons l'épurer grandement ! Il nous suffit d'instancier un objet métier responsable du traitement du formulaire, et de lui passer la requête courante en appelant sa méthode `inscrireUtilisateur()` :

Code : Java - com.sdzee.servlets.Inscription

```
package com.sdzee.servlets;

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.sdzee.beans.Utilisateur;
import com.sdzee.forms.InscriptionForm;

public class Inscription extends HttpServlet {
public static final String ATT_USER = "utilisateur";
public static final String ATT_FORM = "form";
 public static final String VUE = "/WEB-INF/inscription.jsp";


 public void doGet( HttpServletRequest request, HttpServletResponse
response ) throws ServletException, IOException{
  /* Affichage de la page d'inscription */
  this.getServletContext().getRequestDispatcher( VUE ).forward(
request, response );
 }

 public void doPost( HttpServletRequest request, HttpServletResponse
response ) throws ServletException, IOException{
  /* Préparation de l'objet formulaire */
InscriptionForm form = new InscriptionForm();

  /* Appel aux traitement et validation de la requête, et
récupération du bean en résultant */
Utilisateur utilisateur = form.inscrireUtilisateur( request );

  /* Stockage du formulaire et du bean dans l'objet request */
request.setAttribute( ATT_FORM, form );
request.setAttribute( ATT_USER, utilisateur );

  this.getServletContext().getRequestDispatcher( VUE ).forward(
request, response );
 }
}
```

Après initialisation de notre objet métier, la seule chose que notre servlet effectue est un appel à la méthode `inscrireUtilisateur()` qui lui retourne alors un bean **Utilisateur**. Elle stocke finalement ces deux objets dans l'objet requête afin de rendre accessibles à la JSP les données validées et les messages d'erreur retournés.

Dorénavant, notre servlet joue bien uniquement le rôle d'aiguilleur : elle contrôle les données, en se contentant

d'appeler les traitements présents dans le modèle. Elle ne fait que transmettre la requête à un objet métier, à aucun moment elle n'agit directement sur ses données.

doGet() n'est pas doPost(), et vice-versa !

Avant de passer à la suite, je tiens à vous signaler une mauvaise pratique, tristement très couramment rencontrée sur le web. Dans énormément d'exemples de servlets, vous pourrez trouver ce genre de codes :

Code : Java - Exemple de mauvaise pratique dans une servlet

```
import java.io.IOException;

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

public class ExempleServlet extends HttpServlet {
 public void doGet( HttpServletRequest request, HttpServletResponse
response ) throws ServletException, IOException {
  /* Ne fait rien d'autre qu'appeler une JSP */
  this.getServletContext().getRequestDispatcher( "/page.jsp"
).forward( request, response );
 }

 public void doPost( HttpServletRequest request, HttpServletResponse
response ) throws ServletException, IOException {
  /**
* Ici éventuellement des traitements divers, puis au lieu
* d'appeler tout simplement un forwarding...
*/
doGet(request, response);
 }
}
```

Vous comprenez ce qui a été réalisé dans cet exemple ? Puisque la méthode `doGet()` ne fait rien d'autre qu'appeler la vue, le développeur n'a rien trouvé de mieux que d'appeler `doGet()` depuis la méthode `doPost()` pour réaliser le *forwarding* vers la vue... Eh bien cette manière de faire, dans une application qui respecte MVC, est totalement dénuée de sens ! Si vous souhaitez que votre servlet réalise la même chose quelque soit le type de la requête HTTP reçue, alors :

- soit vous surchargez directement la méthode `service()` de la classe mère `HttpServlet`, afin qu'elle ne redirige plus les requêtes entrantes vers les différentes méthodes doXXX() de votre servlet. Vous n'aurez ainsi plus à implémenter `doPost()` et `doGet()` dans votre servlet, et pourrez directement implémenter un traitement unique dans la méthode `service()`.

- soit vous faites en sorte que vos méthodes `doGet()` et `doPost()` appellent une troisième et même méthode, qui effectuera un traitement commun à toutes les requêtes entrantes.

Quel que soit votre choix parmi ces solutions, cela sera toujours mieux que d'écrire que `doGet()` appelle `doPost()`, ou vice-versa !

Pour résumer, retenez bien que croiser ainsi les appels est une mauvaise pratique, qui complique la lisibilité et la maintenance du code de votre application !

Reprise de la JSP

La dernière étape de notre mise à niveau est la modification des appels aux différents attributs au sein de notre page JSP. En effet, auparavant notre servlet transmettait directement la chaîne **resultat** et la map **erreurs** à notre page, ce qui impliquait que :

- nous accédions directement à ces attributs via nos expressions EL ;
- nous accédions aux données saisies par l'utilisateur via l'objet implicite **param**.

Dorénavant, notre servlet transmet notre bean et notre objet métier à notre page, objets qui à leur tour contiennent les données saisies, le résultat et les erreurs. Ainsi, nous allons devoir modifier nos expressions EL afin qu'elles accèdent aux informations à travers nos deux nouveaux objets :

Code : JSP - /WEB-INF/inscription.jsp

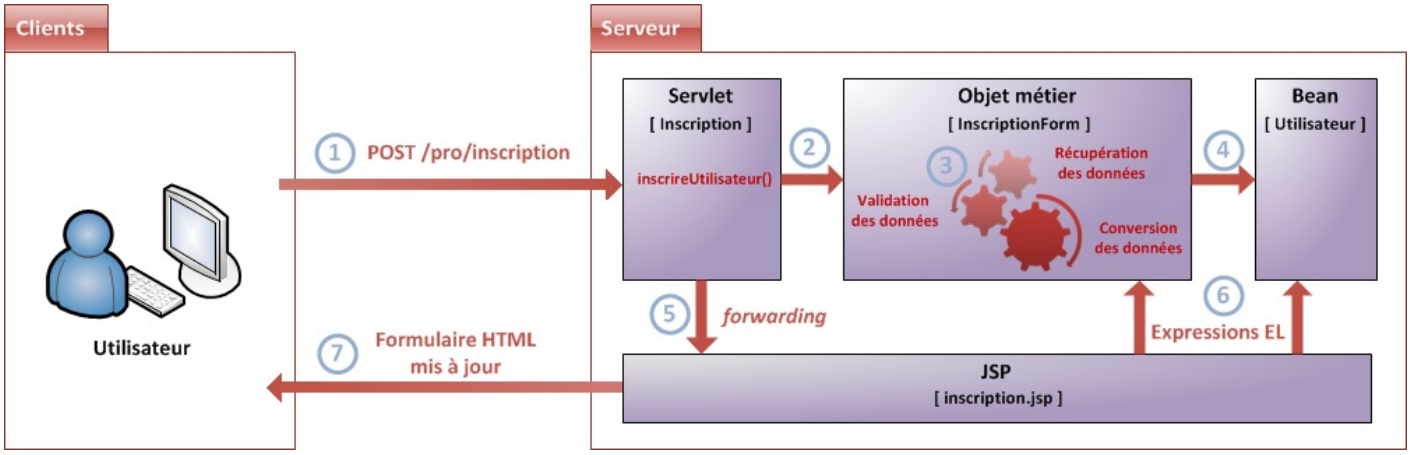
```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Inscription</title>
        <link type="text/css" rel="stylesheet" href="form.css" />
    </head>
    <body>
        <form method="post" action="inscription">
            <fieldset>
                <legend>Inscription</legend>
                <p>Vous pouvez vous inscrire via ce formulaire.</p>

                <label for="nom">Nom d'utilisateur <span
class="requis">*</span></label>
<input type="text" id="nom" name="nom" value="<c:out
value="${utilisateur.nom}"/>" size="20" maxlength="20" />
<span class="erreur">${form.erreurs['nom']}</span>
                <br />

                <label for="motdepasse">Mot de passe <span
class="requis">*</span></label>
                <input type="password" id="motdepasse"
name="motdepasse" value="" size="20" maxlength="20" />
<span class="erreur">${form.erreurs['motdepasse']}</span>
                <br />

                <label for="confirmation">Confirmation du mot de
passe <span class="requis">*</span></label>
                <input type="password" id="confirmation"
name="confirmation" value="" size="20" maxlength="20" />
<span class="erreur">${form.erreurs['confirmation']}</span>
                <br />

                <label for="email">Adresse email</label>
<input type="email" id="email" name="email" value="<c:out
value="${utilisateur.email}"/>" size="20" maxlength="60" />
<span class="erreur">${form.erreurs['email']}</span>
                <br />

                <input type="submit" value="Inscription"
class="sansLabel" />
                <br />

<p class="${empty form.erreurs ? 'succes' :
'erreur'}">${form.resultat}</p>
            </fieldset>
        </form>
    </body>
</html>
```

Les modifications apportées semblent donc mineur :

- l'accès aux erreurs et au résultat se fait à travers l'objet **form** ;
- l'accès aux données se fait à travers le bean **utilisateur**.

Mais en réalité, elles reflètent un changement fondamental dans le principe : notre JSP lit désormais directement les données depuis le modèle !

Voici sous forme d'un schéma ce que nous avons réalisé :



Nous avons ainsi avec succès mis en place une architecture MVC pour le traitement de notre formulaire :

1. les données saisies et envoyées par le client arrivent à la méthode `doPost()` de la servlet.
2. celle-ci ordonne alors le contrôle des données reçues en appelant la méthode `inscrireUtilisateur()` de l'objet métier **InscriptionForm**.
3. l'objet **InscriptionForm** effectue les traitements de validation de chacune des données reçues.
4. il les enregistre par la même occasion dans le bean **Utilisateur**.
5. la méthode `doPost()` récupère enfin les deux objets du modèle, et les transmet à la JSP via la portée requête.
6. la JSP va piocher les données dont elle a besoin grâce aux différentes expressions EL mises en place, qui lui donnent un accès direct aux objets du modèle transmis par la servlet.
7. pour finir, la JSP met à jour l'affichage du formulaire en se basant sur les nouvelles données.

Ne vous fiez pas aux apparences : même si le rendu final de notre exemple n'a pas changé d'un poil, le code a quant à lui bénéficié d'une amélioration conséquente ! Il respecte maintenant à la lettre le modèle MVC.

TP Fil rouge - Étape 3

Avant d'aller plus loin, retour sur le fil rouge à travers lequel vous tenez une belle occasion de mettre en pratique tout ce que vous venez de découvrir dans ces deux chapitres. Vous allez reprendre le code que vous avez développé au cours des étapes précédentes pour y ajouter des vérifications sur le contenu des champs, et l'adapter pour qu'il respecte MVC.

Objectifs

Fonctionnalités

Pour commencer, vous allez devoir modifier vos pages et classes afin d'utiliser la méthode POST pour l'envoi des données depuis vos formulaires de création de clients et de commandes, et non plus la méthode GET.

Deuxièmement, je vous demande de mettre en place des vérifications sur les champs des formulaires :

- chaque champ marqué d'une étoile dans les formulaires devra obligatoirement être renseigné ;
- les champs **nom** et **prenom** devront contenir au moins 2 caractères ;
- le champ **adresse** devra contenir au moins 10 caractères ;
- le champ **telephone** devra être un nombre et contenir au moins 4 numéros ;
- le champ **email** devra contenir une adresse dont le format est valide ;
- le **montant** devra être un nombre positif, éventuellement décimal ;
- les champs **modePaiement**, **statutPaiement**, **modeLivraison** et **statutLivraison** devront contenir au moins 2 caractères ;
- le champ **date** restera désactivé.

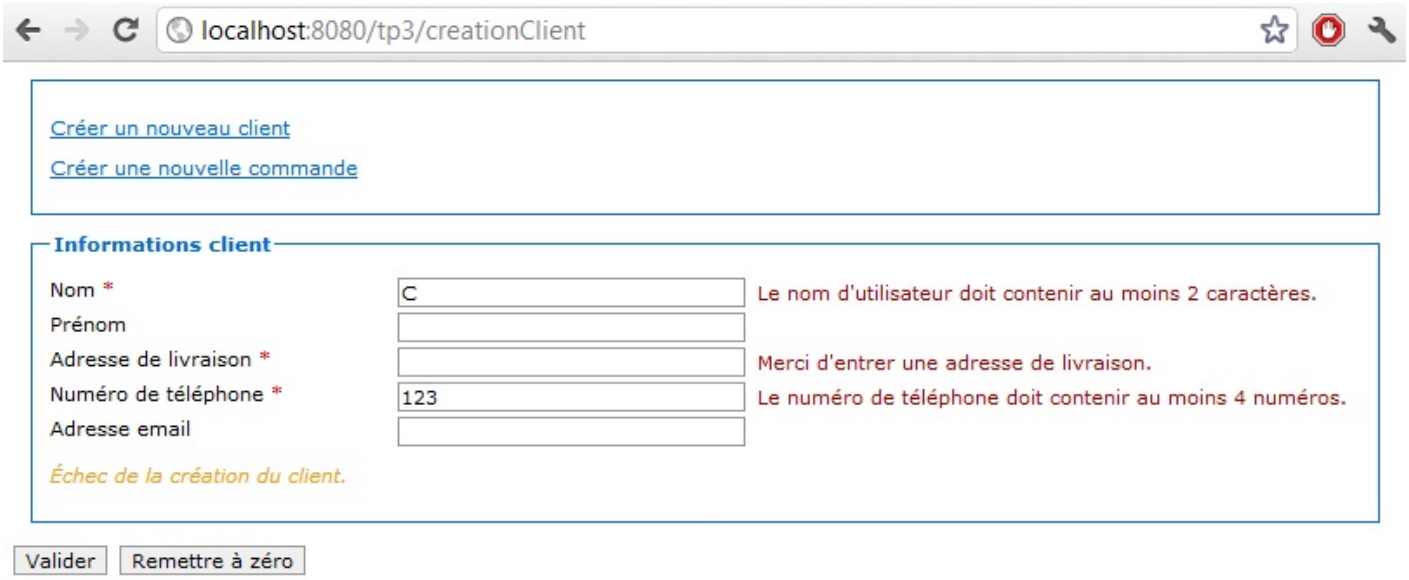
Troisièmement, je vous demande de changer le principe de votre petite application :

- en cas d'erreur lors de la validation (champs manquants ou erronés), vous devrez faire retourner l'utilisateur au formulaire de saisie en lui ré-affichant - sans faille XSS ! - les données qu'il a saisies, et en précisant un message signalant les erreurs sur chaque champ qui pose problème ;
- en cas de succès, vous devrez envoyer l'utilisateur vers la page qui affiche la fiche récapitulative.

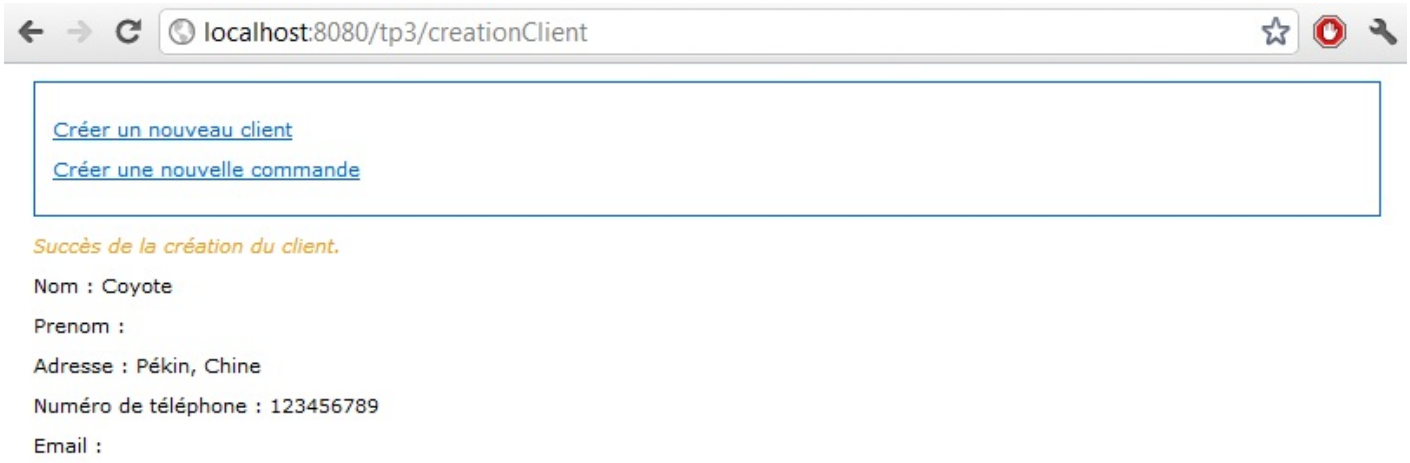
Enfin bien évidemment, tout cela se fera en respectant MVC !

Exemples de rendus

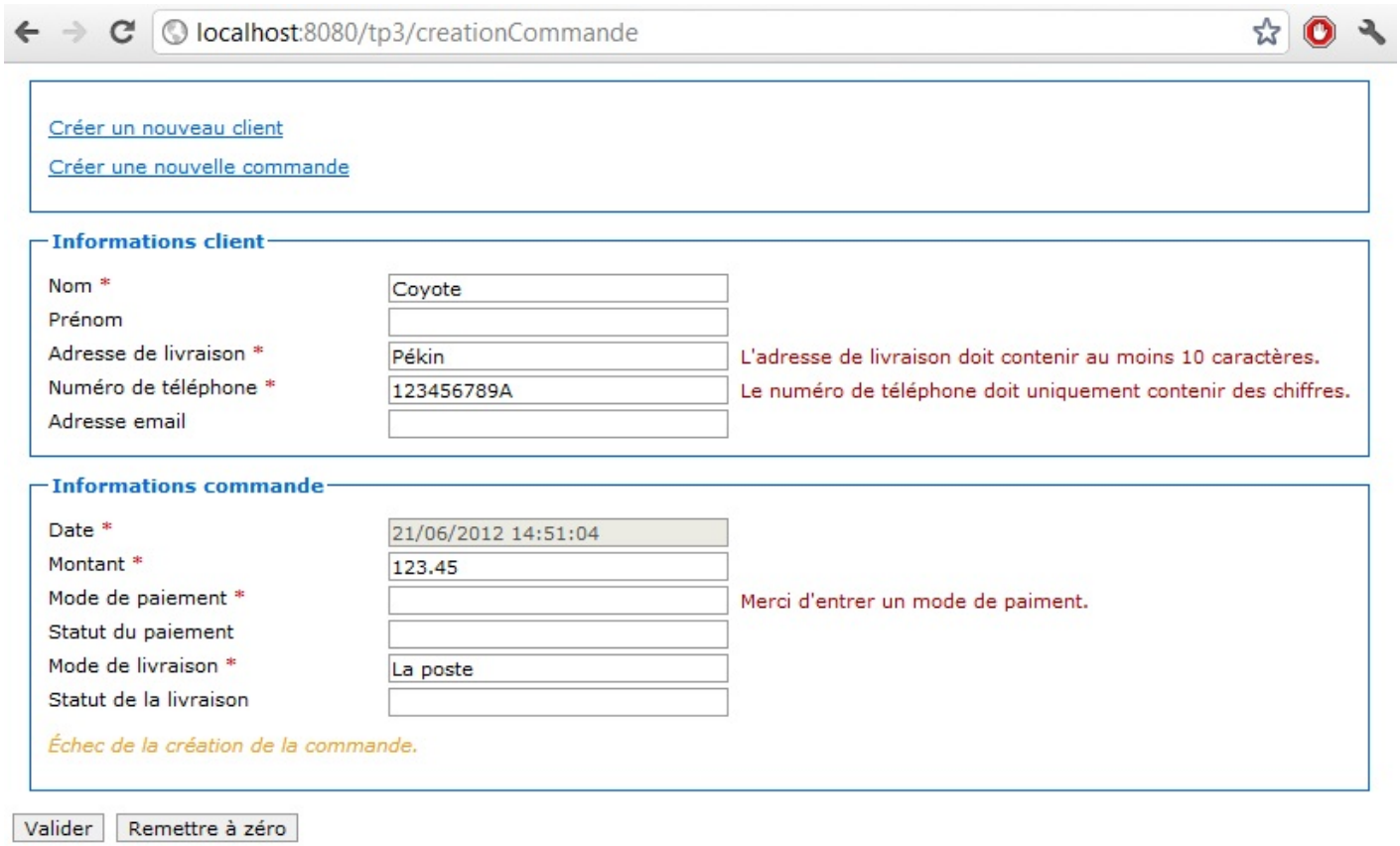
Création de client avec erreurs :



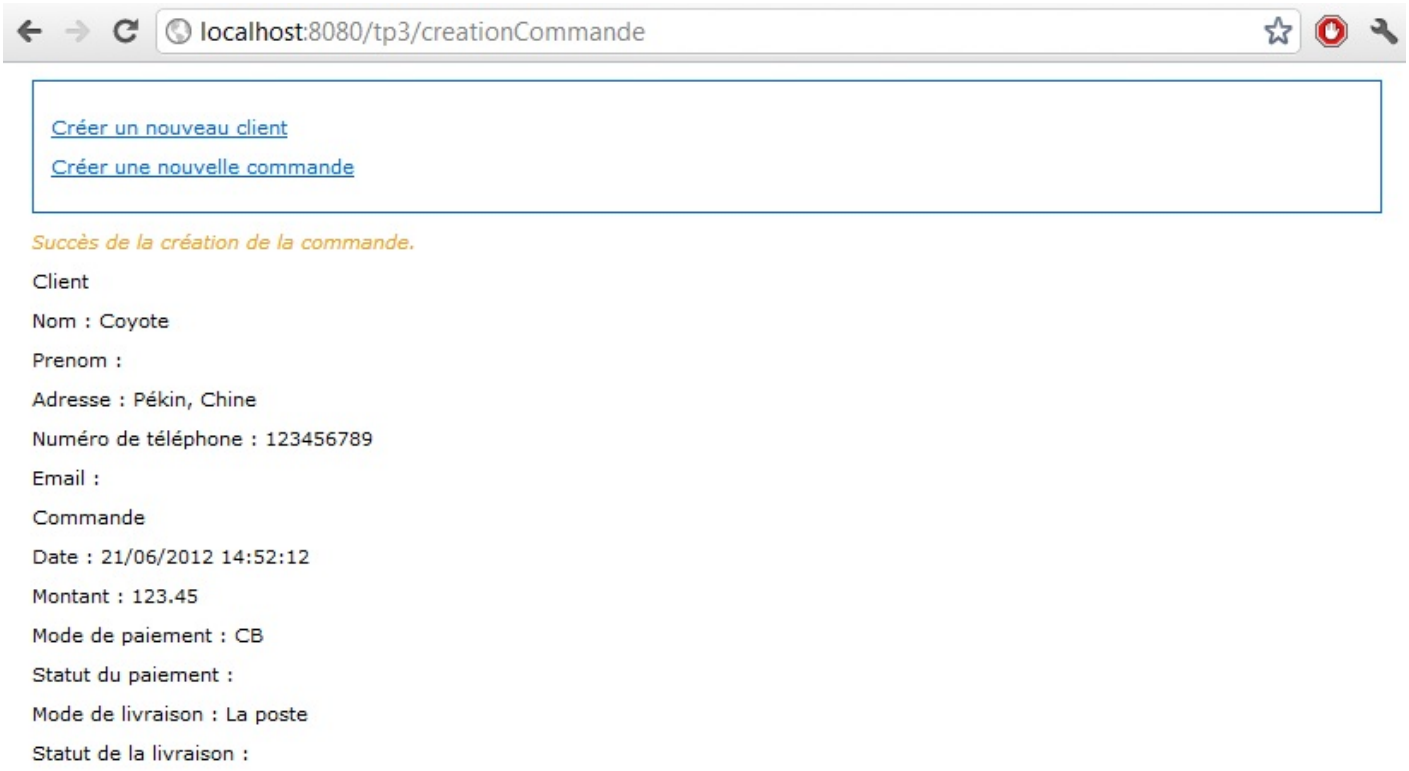
Création de client sans erreur :



Création de commande avec erreurs :



Création de commande sans erreur :



Conseils

Concernant le changement de méthode d'envoi de GET vers POST, pensez bien à ce que cela va impliquer dans vos formulaires et dans vos servlets !

Concernant les vérifications sur le contenu des champs, vous pouvez bien évidemment grandement vous inspirer des méthodes de validation que nous avons mises en place dans le chapitre précédent dans notre système d'inscription. Le principe est très semblable, seules certaines conditions de vérifications changent. De même, afin de respecter MVC vous pourrez prendre exemple sur la conception utilisée dans le chapitre précédent : beans, objets métiers et servlets "nettoyées" !

Enfin concernant le renvoi vers le formulaire de création en cas d'erreur(s), avec affichage des erreurs spécifiques à chaque champ posant problème, là encore vous pouvez vous inspirer de ce que nous avons développé dans le chapitre précédent !

Bref, vous l'aurez compris, ce TP est une application pure et simple de ce que vous venez de découvrir à travers la mise en place de notre système d'inscription. Je m'arrête donc ici pour les conseils, vous avez toutes les informations et tous les outils en main pour remplir votre mission ! Lancez-vous, ne vous découragez pas et surpassez-vous !

Correction

La longueur du sujet est trompeuse : le travail que vous devez fournir est en réalité assez important ! J'espère que vous avez bien pris le temps de réfléchir à l'architecture que vous mettez en place, à la manière dont vos classes et objets s'interconnectent et à la logique de validation à mettre en place. Comme toujours, ce n'est pas la seule manière de faire, le principal est que votre solution respecte les consignes que je vous ai données !

Prenez le temps de réfléchir, de chercher et coder par vous-mêmes. Si besoin, n'hésitez pas à relire le sujet ou à retourner lire les deux précédents chapitres. La pratique est très importante, ne vous ruez pas sur la solution !

Code des objets métier

Secret (cliquez pour afficher)

Objet métier gérant le formulaire de création d'un client :

Code : Java - com.sdzee.tp.forms.CreationClientForm

```
package com.sdzee.tp.forms;
```

```
import java.util.HashMap;
import java.util.Map;

import javax.servlet.http.HttpServletRequest;

import com.sdzee.tp.beans.Client;

public final class CreationClientForm {
    private static final String CHAMP_NOM       = "nomClient";
    private static final String CHAMP_PRENOM    = "prenomClient";
    private static final String CHAMP_ADRESSE   = "adresseClient";
    private static final String CHAMP_TELEPHONE =
"telephoneClient";
    private static final String CHAMP_EMAIL     = "emailClient";

    private String              resultat;
    private Map<String, String> erreurs         = new
HashMap<String, String>();

    public Map<String, String> getErreurs() {
        return erreurs;
    }

    public String getResultat() {
        return resultat;
    }

    public Client creerClient( HttpServletRequest request ) {
        String nom = getValeurChamp( request, CHAMP_NOM );
        String prenom = getValeurChamp( request, CHAMP_PRENOM );
        String adresse = getValeurChamp( request, CHAMP_ADRESSE );
        String telephone = getValeurChamp( request,
CHAMP_TELEPHONE );
        String email = getValeurChamp( request, CHAMP_EMAIL );

        Client client = new Client();

        try {
            validationNom( nom );
        } catch ( Exception e ) {
            setErreur( CHAMP_NOM, e.getMessage() );
        }
        client.setNom( nom );

        try {
            validationPrenom( prenom );
        } catch ( Exception e ) {
            setErreur( CHAMP_PRENOM, e.getMessage() );
        }
        client.setPrenom( prenom );

        try {
            validationAdresse( adresse );
        } catch ( Exception e ) {
            setErreur( CHAMP_ADRESSE, e.getMessage() );
        }
        client.setAdresse( adresse );

        try {
            validationTelephone( telephone );
        } catch ( Exception e ) {
            setErreur( CHAMP_TELEPHONE, e.getMessage() );
        }
        client.setTelephone( telephone );

        try {
            validationEmail( email );
        } catch ( Exception e ) {
            setErreur( CHAMP_EMAIL, e.getMessage() );
        }
```

```
            client.setEmail( email );

            if ( erreurs.isEmpty() ) {
                resultat = "Succès de la création du client.";
            } else {
                resultat = "Échec de la création du client.";
            }

            return client;
        }

        private void validationNom( String nom ) throws Exception {
            if ( nom != null ) {
                if ( nom.length() < 2 ) {
                    throw new Exception( "Le nom d'utilisateur doit
contenir au moins 2 caractères." );
                }
            } else {
                throw new Exception( "Merci d'entrer un nom
d'utilisateur." );
            }
        }

        private void validationPrenom( String prenom ) throws
Exception {
            if ( prenom != null && prenom.length() < 2 ) {
                throw new Exception( "Le prénom d'utilisateur doit
contenir au moins 2 caractères." );
            }
        }

        private void validationAdresse( String adresse ) throws
Exception {
            if ( adresse != null ) {
                if ( adresse.length() < 10 ) {
                    throw new Exception( "L'adresse de livraison doit
contenir au moins 10 caractères." );
                }
            } else {
                throw new Exception( "Merci d'entrer une adresse de
livraison." );
            }
        }

        private void validationTelephone( String telephone ) throws
Exception {
            if ( telephone != null ) {
                if ( !telephone.matches( "^\\d+$" ) ) {
                    throw new Exception( "Le numéro de téléphone doit
uniquement contenir des chiffres." );
                } else if ( telephone.length() < 4 ) {
                    throw new Exception( "Le numéro de téléphone doit
contenir au moins 4 numéros." );
                }
            } else {
                throw new Exception( "Merci d'entrer un numéro de
téléphone." );
            }
        }

        private void validationEmail( String email ) throws Exception
{
            if ( email != null && !email.matches(
"([^.@]+)(\\.[^.@]+)*@([^.@]+\\.)+([^.@]+)" ) ) {
                throw new Exception( "Merci de saisir une adresse mail
valide." );
            }
        }

        /*
```

```
    * Ajoute un message correspondant au champ spécifié à la map des
    erreurs.
    */
        private void setErreur( String champ, String message ) {
            erreurs.put( champ, message );
        }

        /*
    * Méthode utilitaire qui retourne null si un champ est vide, et
    son contenu
    * sinon.
    */
        private static String getValeurChamp( HttpServletRequest
    request, String nomChamp ) {
            String valeur = request.getParameter( nomChamp );
            if ( valeur == null || valeur.trim().length() == 0 ) {
                return null;
            } else {
                return valeur;
            }
        }
    }
```

Objet métier gérant le formulaire de création d'une commande :

Code : Java - com.sdzee.tp.forms.CreationCommandeForm

```
    package com.sdzee.tp.forms;

    import java.util.HashMap;
    import java.util.Map;

    import javax.servlet.http.HttpServletRequest;

    import org.joda.time.DateTime;
    import org.joda.time.format.DateTimeFormat;
    import org.joda.time.format.DateTimeFormatter;

    import com.sdzee.tp.beans.Client;
    import com.sdzee.tp.beans.Commande;

    public final class CreationCommandeForm {
        private static final String CHAMP_DATE             =
    "dateCommande";
        private static final String CHAMP_MONTANT          =
    "montantCommande";
        private static final String CHAMP_MODE_PAIEMENT    =
    "modePaiementCommande";
        private static final String CHAMP_STATUT_PAIEMENT  =
    "statutPaiementCommande";
        private static final String CHAMP_MODE_LIVRAISON   =
    "modeLivraisonCommande";
        private static final String CHAMP_STATUT_LIVRAISON =
    "statutLivraisonCommande";

        private static final String FORMAT_DATE            =
    "dd/MM/yyyy HH:mm:ss";

        private String              resultat;
        private Map<String, String> erreurs                = new
    HashMap<String, String>();

        public Map<String, String> getErreurs() {
            return erreurs;
        }
```

```
    public String getResultat() {
        return resultat;
    }

    public Commande creerCommande( HttpServletRequest request ) {
        /*
* L'objet métier pour valider la création d'un client existe
déjà, il
* est donc déconseillé de dupliquer ici son contenu ! A la place,
il
* suffit de passer la requête courante à l'objet métier existant
et de
* récupérer l'objet Client créé.
*/
        CreationClientForm clientForm = new CreationClientForm();
        Client client = clientForm.creerClient( request );

        /*
* Et très important, il ne faut pas oublier de récupérer le
contenu de
* la map d'erreur créée par l'objet métier CreationClientForm
dans la
* map d'erreurs courante, actuellement vide.
*/
        erreurs = clientForm.getErreurs();

        /*
* Ensuite, il suffit de procéder normalement avec le reste des
champs
* spécifiques à une commande.
*/

        /*
* Récupération et conversion de la date en String selon le
format
* choisi.
*/
        DateTime dt = new DateTime();
        DateTimeFormatter formatter = DateTimeFormat.forPattern(
FORMAT_DATE );
        String date = dt.toString( formatter );

        String montant = getValeurChamp( request, CHAMP_MONTANT );
        String modePaiement = getValeurChamp( request,
CHAMP_MODE_PAIEMENT );
        String statutPaiement = getValeurChamp( request,
CHAMP_STATUT_PAIEMENT );
        String modeLivraison = getValeurChamp( request,
CHAMP_MODE_LIVRAISON );
        String statutLivraison = getValeurChamp( request,
CHAMP_STATUT_LIVRAISON );

        Commande commande = new Commande();

        commande.setClient( client );

        commande.setDate( date );

        double valeurMontant = -1;
        try {
            valeurMontant = validationMontant( montant );
        } catch ( Exception e ) {
            setErreur( CHAMP_MONTANT, e.getMessage() );
        }
        commande.setMontant( valeurMontant );

        try {
            validationModePaiement( modePaiement );
        } catch ( Exception e ) {
            setErreur( CHAMP_MODE_PAIEMENT, e.getMessage() );
```

```
            }
            commande.setModePaiement( modePaiement );

            try {
                validationStatutPaiement( statutPaiement );
            } catch ( Exception e ) {
                setErreur( CHAMP_STATUT_PAIEMENT, e.getMessage() );
            }
            commande.setStatutPaiement( statutPaiement );

            try {
                validationModeLivraison( modeLivraison );
            } catch ( Exception e ) {
                setErreur( CHAMP_MODE_LIVRAISON, e.getMessage() );
            }
            commande.setModeLivraison( modeLivraison );

            try {
                validationStatutLivraison( statutLivraison );
            } catch ( Exception e ) {
                setErreur( CHAMP_STATUT_LIVRAISON, e.getMessage() );
            }
            commande.setStatutLivraison( statutLivraison );

            if ( erreurs.isEmpty() ) {
                resultat = "Succès de la création de la commande.";
            } else {
                resultat = "Échec de la création de la commande.";
            }
            return commande;
        }

        private double validationMontant( String montant ) throws
    Exception {
            double temp;
            if ( montant != null ) {
                try {
                    temp = Double.parseDouble( montant );
                    if ( temp < 0 ) {
                        throw new Exception( "Le montant doit être un
    nombre positif." );
                    }
                } catch ( NumberFormatException e ) {
                    temp = -1;
                    throw new Exception( "Le montant doit être un
    nombre." );
                }
            } else {
                temp = -1;
                throw new Exception( "Merci d'entrer un montant." );
            }
            return temp;
        }

        private void validationModePaiement( String modePaiement )
    throws Exception {
            if ( modePaiement != null ) {
                if ( modePaiement.length() < 2 ) {
                    throw new Exception( "Le mode de paiement doit
    contenir au moins 2 caractères." );
                }
            } else {
                throw new Exception( "Merci d'entrer un mode de
    paiment." );
            }
        }

        private void validationStatutPaiement( String statutPaiement )
    throws Exception {
            if ( statutPaiement != null && statutPaiement.length() < 2
```

```
        ) {
                throw new Exception( "Le statut de paiement doit
contenir au moins 2 caractères." );
            }
        }

        private void validationModeLivraison( String modeLivraison )
throws Exception {
            if ( modeLivraison != null ) {
                if ( modeLivraison.length() < 2 ) {
                    throw new Exception( "Le mode de livraison doit
contenir au moins 2 caractères." );
                }
            } else {
                throw new Exception( "Merci d'entrer un mode de
livraison." );
            }
        }

        private void validationStatutLivraison( String statutLivraison
) throws Exception {
            if ( statutLivraison != null && statutLivraison.length() <
2 ) {
                throw new Exception( "Le statut de livraison doit
contenir au moins 2 caractères." );
            }
        }

        /*
* Ajoute un message correspondant au champ spécifié à la map des
erreurs.
*/
        private void setErreur( String champ, String message ) {
            erreurs.put( champ, message );
        }

        /*
* Méthode utilitaire qui retourne null si un champ est vide, et
son contenu
* sinon.
*/
        private static String getValeurChamp( HttpServletRequest
request, String nomChamp ) {
            String valeur = request.getParameter( nomChamp );
            if ( valeur == null || valeur.trim().length() == 0 ) {
                return null;
            } else {
                return valeur;
            }
        }
}
```

Code des servlets

Secret (cliquez pour afficher)

Servlet gérant la création de client :

Code : Java - com.sdzee.tp.servlets.CreationClient

```
package com.sdzee.tp.servlets;

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.sdzee.tp.beans.Client;
import com.sdzee.tp.forms.CreationClientForm;

public class CreationClient extends HttpServlet {

    public static final String ATT_CLIENT = "client";
    public static final String ATT_FORM   = "form";

    public static final String VUE_SUCCES = "/afficherClient.jsp";
    public static final String VUE_FORM   = "/creerClient.jsp";

    public void doPost( HttpServletRequest request,
HttpServletResponse response ) throws ServletException,
IOException {
        /* Préparation de l'objet formulaire */
        CreationClientForm form = new CreationClientForm();

        /* Traitement de la requête et récupération du bean en
résultant */
        Client client = form.creerClient( request );

        /* Ajout du bean et de l'objet métier à l'objet requête
*/
        request.setAttribute( ATT_CLIENT, client );
        request.setAttribute( ATT_FORM, form );

        if ( form.getErreurs().isEmpty() ) {
            /* Si aucune erreur, alors affichage de la fiche
récapitulative */
            this.getServletContext().getRequestDispatcher(
VUE_SUCCES ).forward( request, response );
        } else {
            /* Sinon, ré-affichage du formulaire de création avec
les erreurs */
            this.getServletContext().getRequestDispatcher(
VUE_FORM ).forward( request, response );
        }
    }
}
```

Servlet gérant la création de commande :

Code : Java - com.sdzee.tp.servlets.CreationCommande

```
package com.sdzee.tp.servlets;

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.sdzee.tp.beans.Commande;
import com.sdzee.tp.forms.CreationCommandeForm;

public class CreationCommande extends HttpServlet {
    public static final String ATT_COMMANDE = "commande";
    public static final String ATT_FORM     = "form";
```

```
    public static final String VUE_SUCCES    =
"/afficherCommande.jsp";
    public static final String VUE_FORM      =
"/creerCommande.jsp";

    public void doPost( HttpServletRequest request,
HttpServletResponse response ) throws ServletException,
IOException {
        /* Préparation de l'objet formulaire */
        CreationCommandeForm form = new CreationCommandeForm();

        /* Traitement de la requête et récupération du bean en
résultant */
        Commande commande = form.creerCommande( request );

        /* Ajout du bean et de l'objet métier à l'objet requête
*/
        request.setAttribute( ATT_COMMANDE, commande );
        request.setAttribute( ATT_FORM, form );

        if ( form.getErreurs().isEmpty() ) {
            /* Si aucune erreur, alors affichage de la fiche
récapitulative */
            this.getServletContext().getRequestDispatcher(
VUE_SUCCES ).forward( request, response );
        } else {
            /* Sinon, ré-affichage du formulaire de création avec
les erreurs */
            this.getServletContext().getRequestDispatcher(
VUE_FORM ).forward( request, response );
        }
    }
}
```

Code des JSP

Secret (cliquez pour afficher)

Page de création d'un client :

Code : JSP - /creerClient.jsp

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Création d'un client</title>
        <link type="text/css" rel="stylesheet" href="<c:url
value="/inc/style.css"/>" />
    </head>
    <body>
        <c:import url="/inc/menu.jsp" />
        <div>
            <form method="post" action="<c:url
value="/creationClient"/>">
                <fieldset>
                    <legend>Informations client</legend>

                    <label for="nomClient">Nom <span
class="requis">*</span></label>
                    <input type="text" id="nomClient"
name="nomClient" value="<c:out value="${client.nom}"/>" size="30"
```

```
maxlength="30" />
                        <span
class="erreur">${form.erreurs['nomClient']}</span>
                        <br />

                        <label for="prenomClient">Prénom </label>
                        <input type="text" id="prenomClient"
name="prenomClient" value="<c:out value="${client.prenom}"/>"
size="30" maxlength="30" />
                        <span
class="erreur">${form.erreurs['prenomClient']}</span>
                        <br />

                        <label for="adresseClient">Adresse de
livraison <span class="requis">*</span></label>
                        <input type="text" id="adresseClient"
name="adresseClient" value="<c:out value="${client.adresse}"/>"
size="30" maxlength="60" />
                        <span
class="erreur">${form.erreurs['adresseClient']}</span>
                        <br />

                        <label for="telephoneClient">Numéro de
téléphone <span class="requis">*</span></label>
                        <input type="text" id="telephoneClient"
name="telephoneClient" value="<c:out
value="${client.telephone}"/>" size="30" maxlength="30" />
                        <span
class="erreur">${form.erreurs['telephoneClient']}</span>
                        <br />

                        <label for="emailClient">Adresse email</label>
                        <input type="email" id="emailClient"
name="emailClient" value="<c:out value="${client.email}"/>"
size="30" maxlength="60" />
                        <span
class="erreur">${form.erreurs['emailClient']}</span>
                        <br />

                        <p class="info">${ form.resultat }</p>
                    </fieldset>
                    <input type="submit" value="Valider"  />
                    <input type="reset" value="Remettre à zéro" /> <br
/>
                </form>
            </div>
        </body>
</html>
```

Page de création d'une commande :

Code : JSP - /creerCommande.jsp

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Création d'une commande</title>
        <link type="text/css" rel="stylesheet" href="<c:url
value="/inc/style.css"/>" />
    </head>
    <body>
        <c:import url="/inc/menu.jsp" />
        <div>
            <form method="post" action="<c:url
```

```
value="/creationCommande"/>">
                <fieldset>
                    <legend>Informations client</legend>

                    <label for="nomClient">Nom <span
class="requis">*</span></label>
                    <input type="text" id="nomClient"
name="nomClient" value="<c:out value="${commande.client.nom}"/>"
size="30" maxlength="30" />
                    <span
class="erreur">${form.erreurs['nomClient']}</span>
                    <br />

                    <label for="prenomClient">Prénom </label>
                    <input type="text" id="prenomClient"
name="prenomClient" value="<c:out
value="${commande.client.prenom}"/>" size="30" maxlength="30" />
                    <span
class="erreur">${form.erreurs['prenomClient']}</span>
                    <br />

                    <label for="adresseClient">Adresse de
livraison <span class="requis">*</span></label>
                    <input type="text" id="adresseClient"
name="adresseClient" value="<c:out
value="${commande.client.adresse}"/>" size="30" maxlength="60" />
                    <span
class="erreur">${form.erreurs['adresseClient']}</span>
                    <br />

                    <label for="telephoneClient">Numéro de
téléphone <span class="requis">*</span></label>
                    <input type="text" id="telephoneClient"
name="telephoneClient" value="<c:out
value="${commande.client.telephone}"/>" size="30" maxlength="30"
/>
                    <span
class="erreur">${form.erreurs['telephoneClient']}</span>
                    <br />

                    <label for="emailClient">Adresse email</label>
                    <input type="email" id="emailClient"
name="emailClient" value="<c:out
value="${commande.client.email}"/>" size="30" maxlength="60" />
                    <span
class="erreur">${form.erreurs['emailClient']}</span>
                    <br />
                </fieldset>
                <fieldset>
                    <legend>Informations commande</legend>

                    <label for="dateCommande">Date <span
class="requis">*</span></label>
                    <input type="text" id="v" name="dateCommande"
value="<c:out value="${commande.date}"/>" size="30" maxlength="30"
disabled />
                    <span
class="erreur">${form.erreurs['dateCommande']}</span>
                    <br />

                    <label for="montantCommande">Montant <span
class="requis">*</span></label>
                    <input type="text" id="montantCommande"
name="montantCommande" value="<c:out
value="${commande.montant}"/>" size="30" maxlength="30" />
                    <span
class="erreur">${form.erreurs['montantCommande']}</span>
                    <br />

                    <label for="modePaiementCommande">Mode de
```

```
        paiement <span class="requis">*</span></label>
                            <input type="text" id="modePaiementCommande"
        name="modePaiementCommande" value="<c:out
        value="${commande.modePaiement}"/>" size="30" maxlength="30" />
                            <span
        class="erreur">${form.erreurs['modePaiementCommande']}</span>
                            <br />

                            <label for="statutPaiementCommande">Statut du
        paiement</label>
                            <input type="text" id="statutPaiementCommande"
        name="statutPaiementCommande" value="<c:out
        value="${commande.statutPaiement}"/>" size="30" maxlength="30" />
                            <span
        class="erreur">${form.erreurs['statutPaiementCommande']}</span>
                            <br />

                            <label for="modeLivraisonCommande">Mode de
        livraison <span class="requis">*</span></label>
                            <input type="text" id="modeLivraisonCommande"
        name="modeLivraisonCommande" value="<c:out
        value="${commande.modeLivraison}"/>" size="30" maxlength="30" />
                            <span
        class="erreur">${form.erreurs['modeLivraisonCommande']}</span>
                            <br />

                            <label for="statutLivraisonCommande">Statut de
        la livraison</label>
                            <input type="text"
        id="statutLivraisonCommande" name="statutLivraisonCommande"
        value="<c:out value="${commande.statutLivraison}"/>" size="30"
        maxlength="30" />
                            <span
        class="erreur">${form.erreurs['statutLivraisonCommande']}</span>
                            <br />

                            <p class="info">${ form.resultat }</p>
                        </fieldset>
                        <input type="submit" value="Valider"  />
                        <input type="reset" value="Remettre à zéro" /> <br
        />
                    </form>
                </div>
            </body>
        </html>
```

Page d'affichage d'un client :

Code : JSP - /afficherClient.jsp

```
        <%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
        <!DOCTYPE html>
        <html>
            <head>
                <meta charset="utf-8" />
                <title>Affichage d'un client</title>
                <link type="text/css" rel="stylesheet" href="<c:url
        value="/inc/style.css"/>" />
            </head>
            <body>
                <c:import url="/inc/menu.jsp" />
                <div id="corps">
                    <p class="info">${ form.resultat }</p>
                    <p>Nom : <c:out value="${ client.nom }"/></p>
                    <p>Prénom : <c:out value="${ client.prenom }"/></p>
                    <p>Adresse : <c:out value="${ client.adresse }"/></p>
```

```
            <p>Numéro de téléphone : <c:out value="${
client.telephone }"/></p>
            <p>Email : <c:out value="${ client.email }"/></p>
        </div>
    </body>
</html>
```

Page d'affichage d'une commande :

Code : JSP - /afficherCommande.jsp

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Affichage d'une commande</title>
        <link type="text/css" rel="stylesheet" href="<c:url
value="/inc/style.css"/>" />
    </head>
    <body>
        <c:import url="/inc/menu.jsp" />
        <div id="corps">
            <p class="info">${ form.resultat }</p>
            <p>Client</p>
            <p>Nom : <c:out value="${ commande.client.nom }"/></p>
            <p>Prénom : <c:out value="${ commande.client.prenom
}"/></p>
            <p>Adresse : <c:out value="${ commande.client.adresse
}"/></p>
            <p>Numéro de téléphone : <c:out value="${
commande.client.telephone }"/></p>
            <p>Email : <c:out value="${ commande.client.email
}"/></p>
            <p>Commande</p>
            <p>Date  : <c:out value="${ commande.date }"/></p>
            <p>Montant  : <c:out value="${ commande.montant
}"/></p>
            <p>Mode de paiement  : <c:out value="${
commande.modePaiement }"/></p>
            <p>Statut du paiement  : <c:out value="${
commande.statutPaiement }"/></p>
            <p>Mode de livraison  : <c:out value="${
commande.modeLivraison }"/></p>
            <p>Statut de la livraison  : <c:out value="${
commande.statutLivraison }"/></p>
        </div>
    </body>
</html>
```

La session : connectez vos clients

Nous allons ici découvrir par l'exemple comment utiliser **la session**.

La situation que nous allons mettre en place est un système de connexion des utilisateurs. Nous allons grandement nous inspirer de ce que nous venons de faire dans le précédent chapitre avec notre système d'inscription, et allons directement appliquer les bonnes pratiques découvertes. Là encore, nous n'allons pas pouvoir mettre en place un système complet de A à Z, puisqu'il nous manque toujours la gestion des données. Mais ce n'est pas important : ce qui compte, c'est que vous tenez là une occasion de plus pour pratiquer la gestion des formulaires en suivant MVC !

Le formulaire

La première chose que nous allons mettre en place est le formulaire de connexion, autrement dit la vue. Cela ne va pas nous demander trop d'efforts : nous allons reprendre l'architecture de la page JSP que nous avons créée dans le chapitre précédent, et l'adapter à nos nouveaux besoins !

Voici le code de notre page **connexion.jsp**, à placer sous le répertoire **/WEB-INF** :

Code : JSP - /WEB-INF/connexion.jsp

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Connexion</title>
        <link type="text/css" rel="stylesheet" href="form.css" />
    </head>
    <body>
        <form method="post" action="connexion">
            <fieldset>
                <legend>Connexion</legend>
                <p>Vous pouvez vous connecter via ce formulaire.</p>

                <label for="nom">Adresse email <span
class="requis">*</span></label>
<input type="email" id="email" name="email" value="<c:out
value="${utilisateur.email}"/>" size="20" maxlength="60" />
<span class="erreur">${form.erreurs['email']}</span>
                <br />

                <label for="motdepasse">Mot de passe <span
class="requis">*</span></label>
                <input type="password" id="motdepasse"
name="motdepasse" value="" size="20" maxlength="20" />
<span class="erreur">${form.erreurs['motdepasse']}</span>
                <br />

                <input type="submit" value="Connexion"
class="sansLabel" />
                <br />

<p class="${empty form.erreurs ? 'succes' :
'erreur'}">${form.resultat}</p>
            </fieldset>
        </form>
    </body>
</html>
```

Nous reprenons la même architecture que pour le système d'inscription : notre JSP exploite un objet **form** contenant les éventuels messages d'erreurs, et un objet **utilisateur** contenant les données saisies et validées.

Par ailleurs, nous réutilisons la même feuille de style.

Le principe de la session

Avant d'aller plus loin, nous devons nous attarder un instant sur ce qu'est une session.

Pourquoi les sessions existent-elles ?

Notre application web est basée sur le protocole HTTP, qui est un protocole dit "sans état" : cela signifie que le serveur, une fois qu'il a envoyé une réponse à la requête d'un client, ne conserve pas les données le concernant. Autrement dit, le serveur traite les clients requête par requête et est absolument incapable de faire un rapprochement entre leur origine : pour lui, chaque nouvelle requête émane d'un nouveau client, puisqu'il oublie le client après l'envoi de chaque réponse... Oui, le serveur HTTP est un peu gâteux !

C'est pour pallier cette lacune que le concept de session a été créé : il permet au serveur de mémoriser des informations relatives au client d'une requête à l'autre.

Qu'est-ce qu'une session en Java EE ?

Souvenez-vous, nous en avions déjà parlé dans ce paragraphe dédié à la portée des objets ainsi que dans ce chapitre sur la JSTL Core :

- la session représente un espace mémoire alloué pour chaque utilisateur, permettant de sauvegarder des informations tout le long de leur visite ;
- le contenu d'une session est conservé jusqu'à ce que l'utilisateur ferme son navigateur, reste inactif trop longtemps, ou encore lorsqu'il se déconnecte du site ;
- l'objet Java sur lequel se base une session est l'objet `HttpSession` ;
- il existe un objet implicite **sessionScope** permettant d'accéder directement au contenu de la session depuis une expression EL dans une page JSP.

Comment manipuler cet objet depuis une servlet ?

Pour commencer, il faut le récupérer depuis l'objet `HttpServletRequest`. Cet objet propose en effet une méthode `getSession()`, qui permet de récupérer la session associée à la requête HTTP en cours si elle existe, ou d'en créer une si elle n'existe pas encore :

Code : Java - Récupération de la session depuis la requête

```
HttpSession session = request.getSession();
```

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

Ensuite, lorsque nous étudions attentivement la documentation de cet objet, nous remarquons entre autres :

- qu'il propose un couple de méthodes `setAttribute()` / `getAttribute()`, permettant la mise en place d'objets au sein de la session et leur récupération, tout comme dans l'objet `HttpServletRequest` ;
- qu'il propose une méthode `getId()`, retournant un identifiant unique permettant de déterminer à qui appartient telle session.

Nous savons donc maintenant qu'il nous suffit d'appeler le code suivant pour enregistrer un objet en session depuis notre servlet, puis le récupérer :

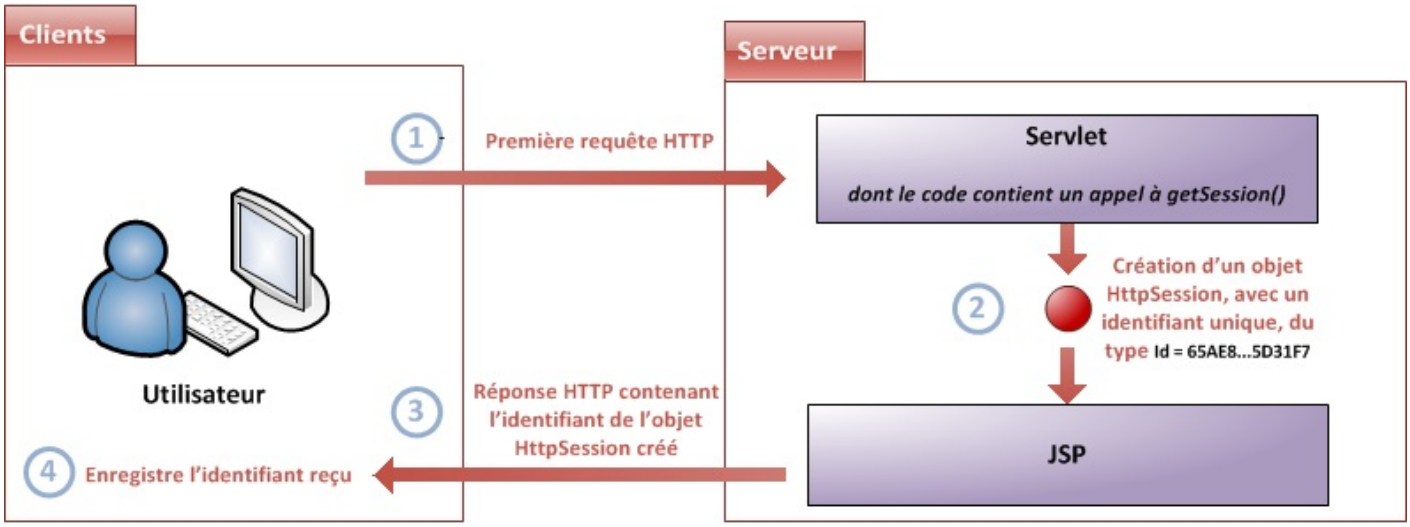
Code : Java - Exemple de manipulations d'objets en session

```
/* Création ou récupération de la session */
HttpSession session = request.getSession();

/* Mise en session d'une chaîne de caractères */
String exemple = "abc";
session.setAttribute( "chaine", exemple );
```

```
        /* Récupération de l'objet depuis la session */
        String chaine = (String) session.getAttribute( "chaine" );
```

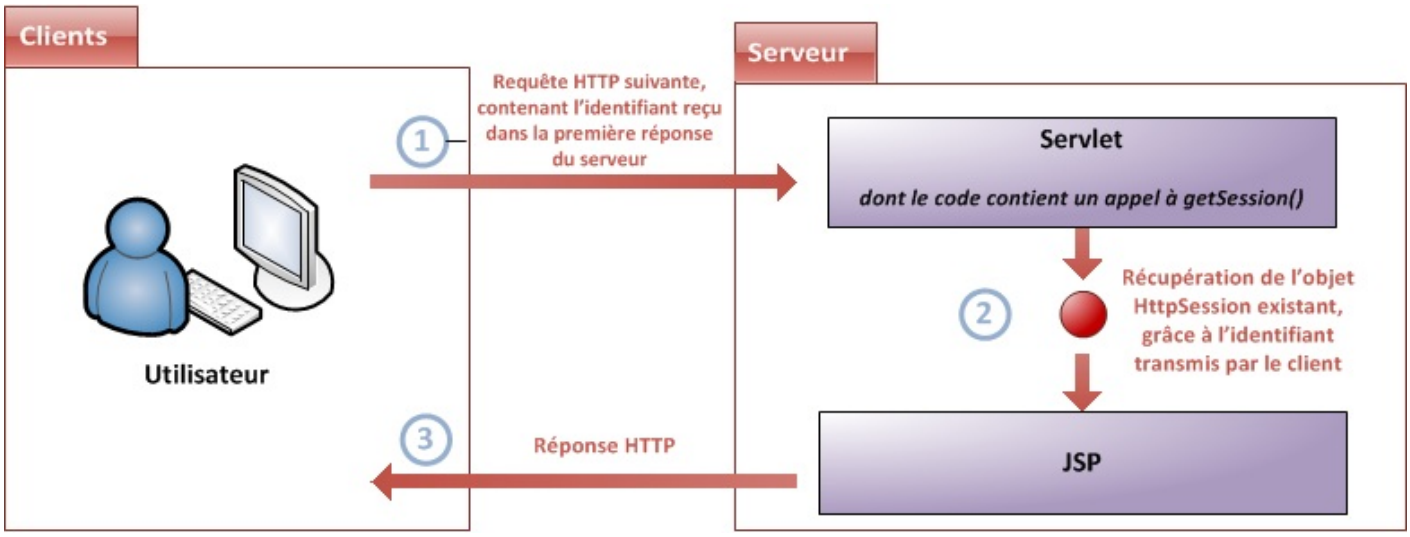
C'est tout ce que nous avons besoin de savoir pour le moment.

Observez l'enchaînement lors de la première visite d'un utilisateur sur une page ou servlet contenant un appel à `request.getSession()` :



1. le navigateur de l'utilisateur envoie une requête au serveur ;
2. la servlet ne trouve aucune session existante lors de l'appel à `getSession()`, et crée donc un nouvel objet `HttpSession` qui contient un identifiant unique ;
3. le serveur place automatiquement l'identifiant de l'objet session dans la réponse renvoyée au navigateur de l'utilisateur ;
4. le navigateur enregistre l'identifiant que le serveur lui a envoyé.

Observez alors ce qui se passe lors des visites suivantes :



1. le navigateur place automatiquement l'identifiant enregistré dans la requête qu'il envoie au serveur ;
2. la servlet retrouve la session associée à l'utilisateur lors de l'appel à `getSession()`, grâce à l'identifiant unique que le navigateur a placé dans la requête ;
3. le serveur sait que le navigateur du client a déjà enregistré l'identifiant de la session courante, et renvoie donc une réponse classique à l'utilisateur : il sait qu'il n'est pas nécessaire de lui transmettre à nouveau l'identifiant !

Vous avez maintenant tout en main pour comprendre comment l'établissement d'une session fonctionne. En ce qui concerne les rouages du système, chaque chose en son temps : dans la dernière partie de ce chapitre, nous analyserons comment tout cela s'organise dans les coulisses !

Très bien, nous avons compris comment ça marche. Maintenant dans notre cas, qu'avons-nous besoin d'enregistrer en session ?

En effet, c'est une très bonne question : qu'est-il intéressant et utile de stocker en session ? Rappelons-le, notre objectif est de connecter un utilisateur : nous souhaitons donc être capable de le reconnaître d'une requête à l'autre.

La première intuition qui nous vient à l'esprit, c'est naturellement de sauvegarder l'adresse mail et le mot de passe de l'utilisateur dans un objet, et de placer cet objet dans la session !

Le modèle

D'après ce que nous venons de déduire, nous pouvons nous inspirer ce que nous avons créé dans le chapitre précédent. Il va nous falloir :

- un bean représentant un utilisateur, que nous placerons en session lors de la connexion ;
- un objet métier représentant le formulaire de connexion, pour traiter et valider les données et connecter l'utilisateur.

En ce qui concerne l'utilisateur, nous n'avons besoin de rien de nouveau : nous disposons déjà du bean créé pour le système d'inscription ! Vous devez maintenant bien mieux saisir le caractère **réutilisable** du JavaBean, que je vous vantais dans ce chapitre.

En ce qui concerne le formulaire, là par contre nous allons devoir créer un nouvel objet métier. Eh oui, nous n'y coupons pas : à chaque nouveau formulaire, nous allons devoir mettre en place un nouvel objet. Vous découvrez ici un des inconvénients majeurs de l'application de MVC dans une application Java EE uniquement basée sur le trio objets métier - servlets - pages JSP : il faut réécrire les méthodes de récupération, conversion et validation des paramètres de la requête HTTP à chaque nouvelle requête traitée !

Plus loin dans ce cours, lorsque nous aurons acquis un bagage assez important pour nous lancer au-delà du Java EE "nu", nous découvrirons comment les développeurs ont réussi à rendre ces étapes entièrement automatisées en utilisant un **framework MVC** pour construire leurs applications. En attendant, vous allez devoir faire preuve de patience et être assidu, nous avons encore du pain sur la planche !

Nous devons donc créer un objet métier, que nous allons nommer **ConnexionForm** et qui va grandement s'inspirer de l'objet **InscriptionForm** :

Code : Java - com.sdzee.forms.ConnexionForm

```
package com.sdzee.forms;

import java.util.HashMap;
import java.util.Map;

import javax.servlet.http.HttpServletRequest;

import com.sdzee.beans.Utilisateur;

public final class ConnexionForm {
    private static final String CHAMP_EMAIL  = "email";
    private static final String CHAMP_PASS   = "motdepasse";

    private String              resultat;
    private Map<String, String> erreurs      = new HashMap<String,
String>();

    public String getResultat() {
        return resultat;
    }

    public Map<String, String> getErreurs() {
        return erreurs;
    }

public Utilisateur connecterUtilisateur( HttpServletRequest request
```

```
        ) {
            /* Récupération des champs du formulaire */
            String email = getValeurChamp( request, CHAMP_EMAIL );
            String motDePasse = getValeurChamp( request, CHAMP_PASS );

            Utilisateur utilisateur = new Utilisateur();

            /* Validation du champ email. */
            try {
                validationEmail( email );
            } catch ( Exception e ) {
                setErreur( CHAMP_EMAIL, e.getMessage() );
            }
        utilisateur.setEmail( email );

            /* Validation du champ mot de passe. */
            try {
                validationMotDePasse( motDePasse );
            } catch ( Exception e ) {
                setErreur( CHAMP_PASS, e.getMessage() );
            }
        utilisateur.setMotDePasse( motDePasse );

            /* Initialisation du résultat global de la validation. */
            if ( erreurs.isEmpty() ) {
                resultat = "Succès de la connexion.";
            } else {
                resultat = "Échec de la connexion.";
            }

        return utilisateur;
        }

        /**
        * Valide l'adresse email saisie.
        */
        private void validationEmail( String email ) throws Exception {
            if ( email != null && !email.matches(
        "([^.@]+)(\\.[^.@]+)*@([^.@]+\\.)+([^.@]+)" ) ) {
                throw new Exception( "Merci de saisir une adresse mail
        valide." );
            }
        }

        /**
        * Valide le mot de passe saisi.
        */
        private void validationMotDePasse( String motDePasse ) throws
        Exception {
            if ( motDePasse != null ) {
                if ( motDePasse.length() < 3 ) {
                    throw new Exception( "Le mot de passe doit contenir
        au moins 3 caractères." );
                }
            } else {
                throw new Exception( "Merci de saisir votre mot de
        passe." );
            }
        }

        /*
        * Ajoute un message correspondant au champ spécifié à la map des
        erreurs.
        */
        private void setErreur( String champ, String message ) {
            erreurs.put( champ, message );
        }

        /*
        * Méthode utilitaire qui retourne null si un champ est vide, et son
```

```
        contenu
        * sinon.
        */
            private static String getValeurChamp( HttpServletRequest
        request, String nomChamp ) {
                String valeur = request.getParameter( nomChamp );
                if ( valeur == null || valeur.trim().length() == 0 ) {
                    return null;
                } else {
                    return valeur;
                }
            }
        }
```

Nous retrouvons ici la même architecture que dans l'objet **InscriptionForm** :

- des constantes d'identification des champs du formulaire ;
- des méthodes de validation des champs ;
- une méthode de gestion des erreurs ;
- une méthode centrale, `connecterUtilisateur()`, qui fait intervenir les méthodes suscitées et renvoie un bean **Utilisateur**.

Rien de nouveau n'entre en jeu pour l'instant. Si vous ne comprenez pas bien ce code, vous devez relire attentivement les deux chapitres précédents : tout y est expliqué !

La servlet

Afin de rendre tout ce petit monde opérationnel, nous devons mettre en place une servlet dédiée à la connexion. Une fois n'est pas coutume, nous allons grandement nous inspirer de ce que nous avons créé dans les chapitres précédents. Voici donc le code de la servlet nommée **Connexion**, placée tout comme sa grande sœur dans le package `com.sdzee.servlets` :

Code : Java - com.sdzee.servlets.Connexion

```
        package com.sdzee.servlets;

        import java.io.IOException;

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

        import com.sdzee.beans.Utilisateur;
        import com.sdzee.forms.ConnexionForm;

        public class Connexion extends HttpServlet {
            public static final String ATT_USER         = "utilisateur";
            public static final String ATT_FORM         = "form";
        public static final String ATT_SESSION_USER = "sessionUtilisateur";
            public static final String VUE              = "/WEB-
        INF/connexion.jsp";

            public void doGet( HttpServletRequest request,
        HttpServletResponse response ) throws ServletException, IOException
        {
                /* Affichage de la page de connexion */
                this.getServletContext().getRequestDispatcher( VUE
        ).forward( request, response );
            }

            public void doPost( HttpServletRequest request,
        HttpServletResponse response ) throws ServletException, IOException
        {
                /* Préparation de l'objet formulaire */
                ConnexionForm form = new ConnexionForm();
```

```
        /* Traitement de la requête et récupération du bean en
résultant */
        Utilisateur utilisateur = form.connecterUtilisateur( request
);

        /* Récupération de la session depuis la requête */
HttpSession session = request.getSession();

        /**
* Si aucune erreur de validation n'a eu lieu, alors ajout du bean
* Utilisateur à la session, sinon suppression du bean de la
session.
*/
if ( form.getErreurs().isEmpty() ) {
session.setAttribute( ATT_SESSION_USER, utilisateur );
} else {
session.setAttribute( ATT_SESSION_USER, null );
}

        /* Stockage du formulaire et du bean dans l'objet request
*/
        request.setAttribute( ATT_FORM, form );
        request.setAttribute( ATT_USER, utilisateur );

        this.getServletContext().getRequestDispatcher( VUE
).forward( request, response );
    }
}
```

Au niveau de la structure, rien de nouveau : la servlet joue bien le rôle d'aiguilleur, en appelant les traitements présents dans l'objet **ConnexionForm**, et en récupérant le bean **utilisateur**. Ce qui change cette fois, c'est bien entendu la gestion de la session, surlignée en jaune dans le code ci-dessus :

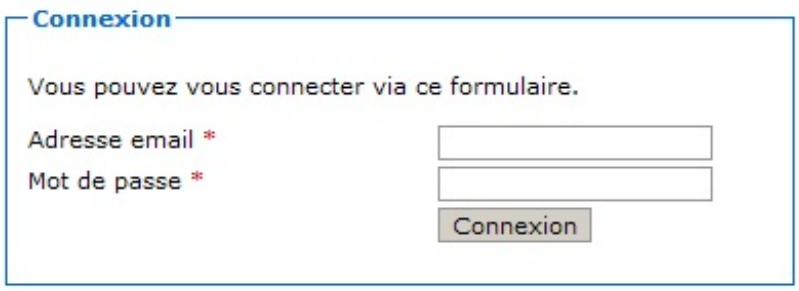
- à la ligne 33, nous appelons la méthode `request.getSession()` pour créer une session ou récupérer la session existante ;
- dans le bloc `if` lignes 39 à 43, nous enregistrons le bean **utilisateur** en tant qu'attribut de session uniquement si aucune erreur de validation n'a été envoyée lors de l'exécution de la méthode `connecterUtilisateur()`. À partir du moment où une seule erreur est détectée, c'est à dire si `form.getErreurs.isEmpty()` renvoie `false`, alors le bean **utilisateur** est supprimé de la session, via un passage de l'attribut à `null`.

Pour terminer, voici la configuration de cette servlet dans le fichier **web.xml** de l'application :

Code : XML - /WEB-INF/web.xml

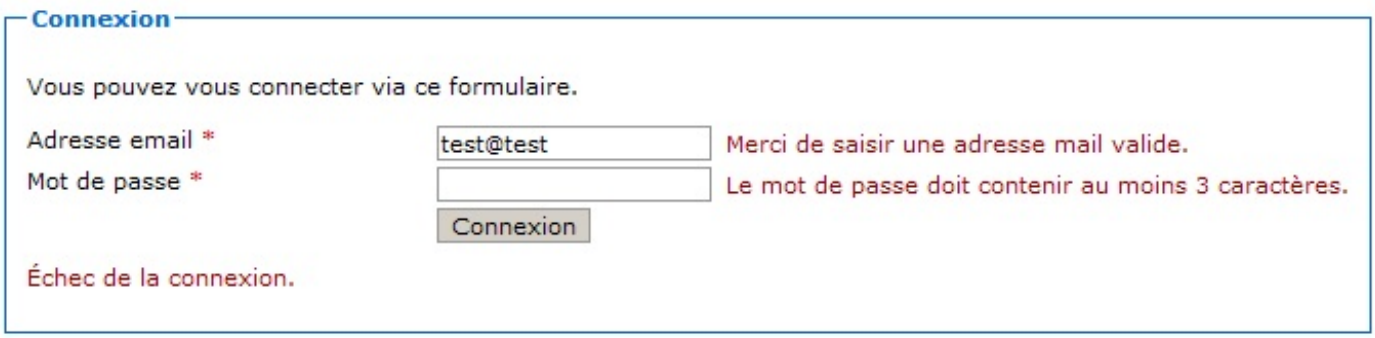
```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
 <servlet>
  <servlet-name>Inscription</servlet-name>
  <servlet-class>com.sdzee.servlets.Inscription</servlet-class>
 </servlet>
<servlet>
<servlet-name>Connexion</servlet-name>
<servlet-class>com.sdzee.servlets.Connexion</servlet-class>
</servlet>

 <servlet-mapping>
  <servlet-name>Inscription</servlet-name>
  <url-pattern>/inscription</url-pattern>
 </servlet-mapping>
<servlet-mapping>
<servlet-name>Connexion</servlet-name>
<url-pattern>/connexion</url-pattern>
</servlet-mapping>
</web-app>
```

Une fois tout ce code bien en place, vous pouvez accéder à la page de connexion via votre navigateur en vous rendant sur l'URL http://localhost:8080/pro/connexion. Ci-dessous, le résultat attendu :



Formulaire de connexion.

Oui mais voilà, nous n'avons pas encore de moyen de tester le bon fonctionnement de ce semblant de système de connexion ! Et pour cause, les seuls messages que nous affichons dans notre vue, ce sont les résultats des vérifications du contenu des champs du formulaire... Ci-dessous, le résultat actuel respectivement lors d'un échec et d'un succès de la validation :



Échec de la connexion.

Succès de la connexion.

Ce qu'il serait maintenant intéressant de vérifier dans notre vue, c'est le contenu de la session. Et comme le hasard fait très bien les choses, je vous ai justement rappelé en début de chapitre qu'il existe un objet implicite nommé **sessionScope**, dédié à l'accès au contenu de la session !

Les vérifications

Test du formulaire de connexion

Pour commencer, nous allons donc modifier notre page JSP afin d'afficher le contenu de la session :

Code : JSP - /WEB-INF/connexion.jsp

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

```
            <meta charset="utf-8" />
            <title>Connexion</title>
            <link type="text/css" rel="stylesheet" href="form.css" />
        </head>
        <body>
            <form method="post" action="connexion">
                <fieldset>
                    <legend>Connexion</legend>
                    <p>Vous pouvez vous connecter via ce formulaire.</p>

                    <label for="nom">Adresse email <span
class="requis">*</span></label>
                    <input type="email" id="email" name="email"
value="<c:out value="${utilisateur.email}"/>" size="20"
maxlength="60" />
                    <span class="erreur">${form.erreurs['email']}</span>


                    <label for="motdepasse">Mot de passe <span
class="requis">*</span></label>
                    <input type="password" id="motdepasse"
name="motdepasse" value="" size="20" maxlength="20" />
                    <span
class="erreur">${form.erreurs['motdepasse']}</span>


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


                    <p class="${empty form.erreurs ? 'succes' :
'erreur'}">${form.resultat}</p>

<%-- Vérification de la présence d'un objet utilisateur en session --%>
<c:if test="${!empty sessionScope.sessionUtilisateur}">
<%-- Si l'utilisateur existe en session, alors on affiche son
adresse email. --%>
<p class="succes">Vous êtes connecté(e) avec l'adresse :
${sessionScope.sessionUtilisateur.email}</p>
</c:if>
                </fieldset>
            </form>
        </body>
    </html>
```

Dans cette courte modification, vous pouvez remarquer :

- l'utilisation de l'objet implicite **sessionScope** pour cibler la portée session ;
- la mise en place d'un test conditionnel via la balise **<c:if/>**. Son contenu (les lignes 32 et 33) s'affichera uniquement si le test est validé ;
- le test de l'existence d'un objet via l'expression `${!empty ...}`. Si aucun objet n'est trouvé, ce test renvoie **false** ;
- l'accès au bean **sessionUtilisateur** de la session via l'expression `${sessionScope.sessionUtilisateur}` ;
- l'accès à la propriété **email** du bean **sessionUtilisateur** via l'expression `${sessionScope.sessionUtilisateur.email}`.

Rappelez-vous : nous pourrions très bien accéder à l'objet en écrivant simplement `${sessionUtilisateur}`, et l'expression EL chercherait alors d'elle-même un objet nommé **sessionUtilisateur** dans chaque portée. Mais je vous ai déjà dit que la bonne pratique était de réserver cette écriture à l'accès des objets de la portée **page**, et de toujours accéder aux objets des autres portées en précisant l'objet implicite correspondant (**requestScope**, **sessionScope** ou **applicationScope**).

Accédez maintenant à la page http://localhost:8080/pro/connexion, et entrez des données valides. Voici le résultat attendu après succès de la connexion :

Ensuite, ré-affichez la page http://localhost:8080/pro/connexion, mais attention pas en appuyant sur F5 ni en actualisant la page : cela renverrait les données de votre formulaire ! Non, ré-entrez simplement l'URL dans le même onglet de votre navigateur, ou bien ouvrez un nouvel onglet. Voici le résultat attendu :

Vous pouvez alors constater que la mémorisation de l'utilisateur a fonctionné ! Lorsqu'il a reçu la deuxième requête d'affichage du formulaire, le serveur vous a reconnu : il sait que vous êtes le client qui a effectué la requête de connexion auparavant, et a conservé vos informations dans la session. En outre, vous voyez également que les informations qui avaient été saisies dans les champs du formulaire lors de la première requête sont quant à elles bien évidemment perdues : elles n'avaient été gérées que via l'objet **request**, et ont donc été détruites après envoi de la première réponse au client.

Vous devez maintenant mieux comprendre cette histoire de portée des objets : l'objet qui a été enregistré en session reste accessible sur le serveur au fil des requêtes d'un même client, alors que l'objet qui a été enregistré dans la requête n'est accessible que lors d'une requête donnée, et disparaît du serveur dès que la réponse est envoyée au client.

Pour finir, testons l'effacement de l'objet de la session lorsqu'une erreur de validation survient. Remplissez le formulaire avec des données invalides, et constatez :

Le contenu du corps de la balise `<c:if/>` n'est ici pas affiché. Cela signifie que le test de présence de l'objet en session a retourné `false`, et donc que notre servlet a bien passé l'objet **utilisateur** à `null` dans la session. En conclusion, jusqu'à présent tout roule !

Test de la destruction de session

Je vous l'ai rappelé en début de chapitre, la session peut être détruite dans plusieurs circonstances :

- l'utilisateur ferme son navigateur ;
- la session expire après une période d'inactivité de l'utilisateur ;
- l'utilisateur se déconnecte.

L'utilisateur ferme son navigateur

Ce paragraphe va être très court. Faites le test vous-mêmes :

1. ouvrez un navigateur et affichez le formulaire de connexion ;
2. entrez des données valides et connectez-vous ;
3. fermez votre navigateur ;
4. rouvrez-le, et rendez-vous à nouveau sur le formulaire de connexion.

Vous constaterez alors que le serveur ne vous a pas reconnu : les informations vous concernant n'existent plus, et le serveur considère que vous êtes un nouveau client.

La session expire après une période d'inactivité de l'utilisateur

Par défaut avec Tomcat, la durée maximum de validité imposée après laquelle la session sera automatiquement détruite par le serveur est de 30 minutes. Vous vous doutez bien que nous n'allons pas poireauter une demi-heure devant notre écran pour vérifier si cela fonctionne bien : vous avez la possibilité via le fichier **web.xml** de votre application de personnaliser cette durée. Ouvrez-le dans Eclipse et modifiez-le comme suit :

Code : XML - /WEB-INF/web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
<session-config>
<session-timeout>1</session-timeout>
</session-config>

 <servlet>
  <servlet-name>Inscription</servlet-name>
  <servlet-class>com.sdzee.servlets.Inscription</servlet-class>
 </servlet>
 <servlet>
  <servlet-name>Connexion</servlet-name>
  <servlet-class>com.sdzee.servlets.Connexion</servlet-class>
 </servlet>

 <servlet-mapping>
  <servlet-name>Inscription</servlet-name>
  <url-pattern>/inscription</url-pattern>
 </servlet-mapping>
 <servlet-mapping>
  <servlet-name>Connexion</servlet-name>
  <url-pattern>/connexion</url-pattern>
 </servlet-mapping>
</web-app>
```

Le champ **<session-timeout>** permet de définir en minutes le temps d'inactivité de l'utilisateur après lequel sa session sera détruite. Je l'ai ici rabaissé à une minute, uniquement pour effectuer notre vérification. Redémarrez Tomcat afin que la modification apportée au fichier soit prise en compte, puis :

1. ouvrez un navigateur et affichez le formulaire de connexion ;
2. entrez des données valides et connectez-vous ;
3. attendez quelques minutes, puis affichez à nouveau le formulaire, dans la même page ou dans un nouvel onglet.

Vous constaterez alors que le serveur vous a oublié : les informations vous concernant n'existent plus, et le serveur considère

que vous êtes un nouveau client.

Une fois ce test effectué, éditez à nouveau votre fichier **web.xml** et supprimez la section fraîchement ajoutée : dans la suite de nos exemples, nous n'allons pas avoir besoin de cette limitation.

L'utilisateur se déconnecte

Cette dernière vérification va nécessiter un peu de développement. En effet, nous avons créé une servlet de connexion, mais nous n'avons pas encore mis en place de servlet de déconnexion. Par conséquent, il est pour le moment impossible pour le client de se déconnecter volontairement du site, il est obligé de fermer son navigateur ou d'attendre que la durée d'inactivité soit dépassée.

Comment détruire manuellement une session ?

Il faut regarder dans la documentation de l'objet `HttpSession` pour répondre à cette question : nous y trouvons une méthode `invalidate()`, qui supprime une session et les objets qu'elle contient, et envoie une exception si jamais elle est appliquée sur une session déjà détruite.

Créons sans plus attendre notre nouvelle servlet nommée **Deconnexion** :

Code : Java - com.sdzee.servlets.Deconnexion

```
package com.sdzee.servlets;

import java.io.IOException;

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

public class Deconnexion extends HttpServlet {
public static final String URL_REDIRECTION =
"http://www.siteduzero.com";

    public void doGet( HttpServletRequest request,
HttpServletResponse response ) throws ServletException, IOException
{
        /* Récupération et destruction de la session en cours */
        HttpSession session = request.getSession();
session.invalidate();

        /* Redirection vers le site du zéro ! */
response.sendRedirect( URL_REDIRECTION );
    }
}
```

Vous remarquez ici deux nouveautés :

- l'appel à la méthode `invalidate()` de l'objet `HttpSession` ;
- la redirection vers la page de connexion via la méthode `sendRedirect()` de l'objet `HttpServletResponse`, en lieu de place du *forwarding* que nous utilisions auparavant.

Quelle est la différence entre la redirection et le forwarding ?

En réalité, vous le savez déjà ! Eh oui, vous ne l'avez pas encore appliqué depuis une servlet, mais je vous ai déjà expliqué le principe lorsque nous avons découvert la balise **`<c:redirect/>`**, dans cette partie du chapitre portant sur la JSTL Core.

Pour rappel donc, une redirection HTTP implique l'envoi d'une réponse au client, alors que le *forwarding* s'effectue sur le serveur et le client n'en est pas tenu informé. Cela implique notamment que via un *forwarding* il est uniquement possible de cibler des pages internes à l'application, alors que via la redirection il est possible d'atteindre n'importe quelle URL publique ! En l'occurrence, j'ai dans notre servlet fait en sorte que lorsque vous vous déconnectez, vous êtes redirigé vers votre site web préféré.

Fin du rappel, nous allons de toute manière y revenir dans le prochain paragraphe. Pour le moment, concentrons-nous sur la destruction de notre session !

Déclarons notre servlet dans le fichier **web.xml** de notre application :

Code : XML - /WEB-INF/web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
 <servlet>
  <servlet-name>Inscription</servlet-name>
  <servlet-class>com.sdzee.servlets.Inscription</servlet-class>
 </servlet>
 <servlet>
  <servlet-name>Connexion</servlet-name>
  <servlet-class>com.sdzee.servlets.Connexion</servlet-class>
 </servlet>
<servlet>
<servlet-name>Deconnexion</servlet-name>
<servlet-class>com.sdzee.servlets.Deconnexion</servlet-class>
</servlet>

 <servlet-mapping>
  <servlet-name>Inscription</servlet-name>
  <url-pattern>/inscription</url-pattern>
 </servlet-mapping>
 <servlet-mapping>
  <servlet-name>Connexion</servlet-name>
  <url-pattern>/connexion</url-pattern>
 </servlet-mapping>
<servlet-mapping>
<servlet-name>Deconnexion</servlet-name>
<url-pattern>/deconnexion</url-pattern>
</servlet-mapping>
</web-app>
```

Redémarrez Tomcat pour que la modification du fichier **web.xml** soit prise en compte, et testez alors comme suit :

1. ouvrez un navigateur et affichez le formulaire de connexion ;
2. entrez des données valides et connectez-vous ;
3. entrez l'URL http://localhost:8080/pro/deconnexion ;
4. affichez à nouveau le formulaire de connexion.

Vous constaterez alors que lors de votre retour, le serveur ne vous reconnaît pas : la session a bien été détruite.

Différence entre forwarding et redirection

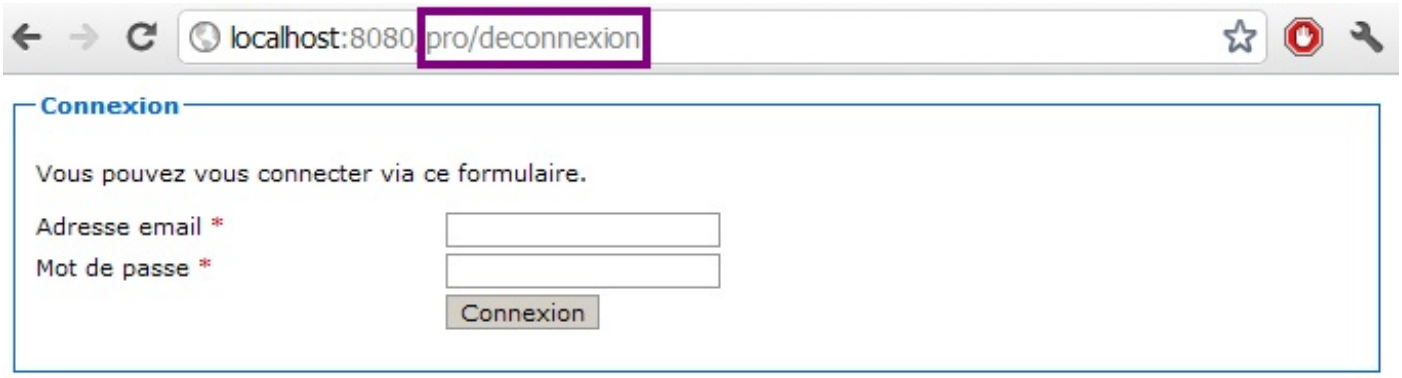
Avant de continuer, puisque nous y sommes, testons cette histoire de *forwarding* et de redirection. Modifiez le code de la servlet comme suit :

Code : Java - com.sdzee.servlets.Deconnexion

```
package com.sdzee.servlets;

import java.io.IOException;

import javax.servlet.ServletException;
```

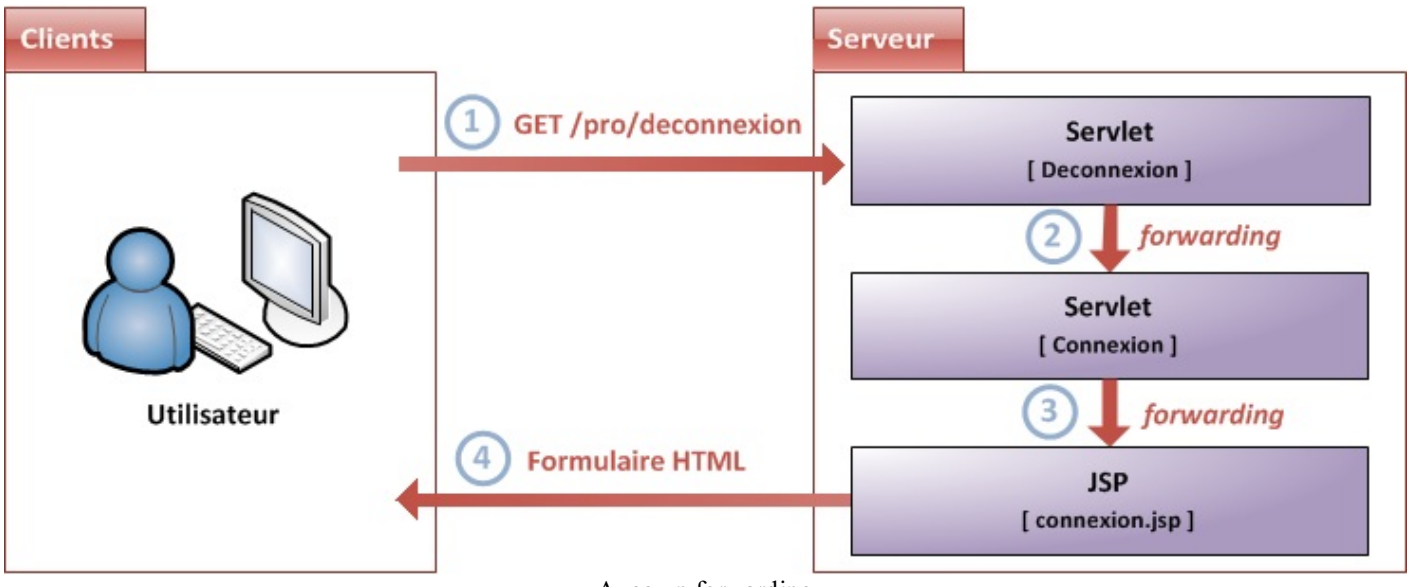
```
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

public class Deconnexion extends HttpServlet {
public static final String VUE = "/connexion";

    public void doGet( HttpServletRequest request,
HttpServletResponse response ) throws ServletException, IOException
{
        /* Récupération et destruction de la session en cours */
        HttpSession session = request.getSession();
        session.invalidate();

        /* Affichage de la page de connexion */
this.getServletContext().getRequestDispatcher( VUE ).forward(
request, response );
    }
}
```

Nous avons ici simplement mis en place un *forwarding* vers la servlet de connexion : une fois déconnecté, vous allez visualiser le formulaire de connexion dans votre navigateur. Oui, mais voyez plutôt :



Forwarding.

Vous comprenez ce qu'il s'est passé ? Comme je vous l'ai expliqué dans plusieurs chapitres, le client n'est pas au courant qu'un *forwarding* a été réalisé côté serveur. Pour lui, la page jointe est /pro/deconnexion, et c'est bien elle qui lui a renvoyé une réponse HTTP. Par conséquent, l'URL dans la barre d'adresse de votre navigateur n'a pas changé ! Pourtant, côté serveur a été effectué un petit enchaînement de *forwardings* :

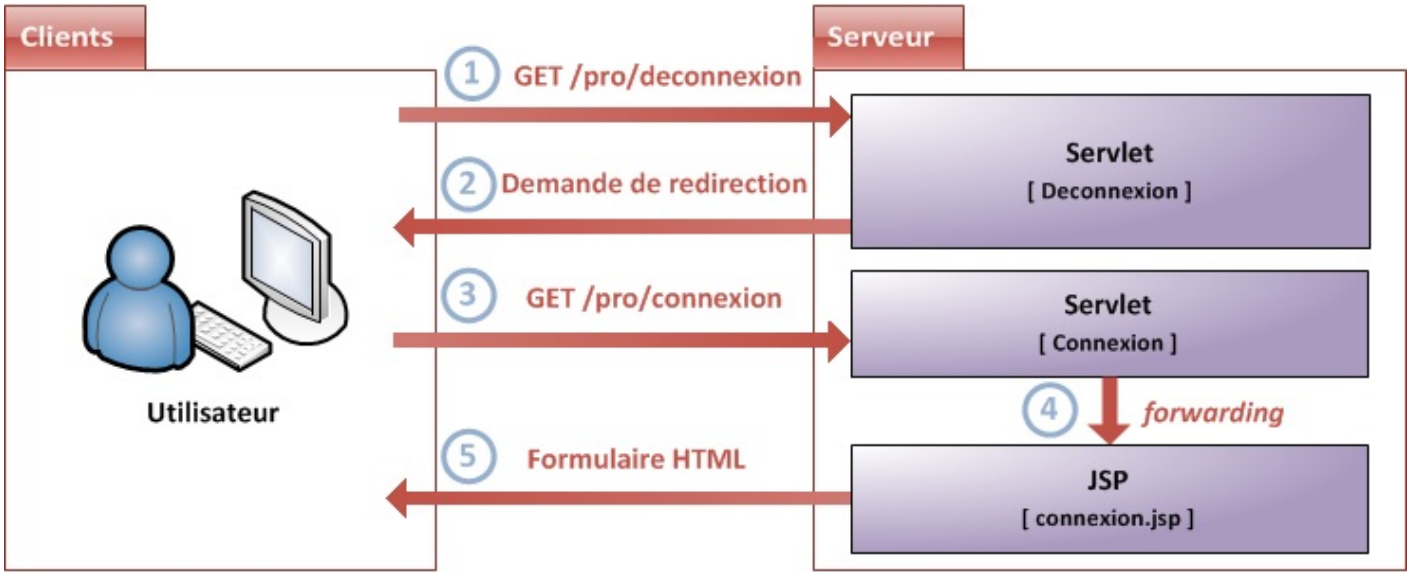


Avec un forwarding

1. l'utilisateur accède à la page de déconnexion depuis son navigateur ;
2. la servlet de déconnexion transfère la requête vers la servlet de connexion via un *forwarding* ;
3. la servlet de connexion transfère la requête vers la JSP du formulaire de connexion via un *forwarding* ;
4. la JSP renvoie le formulaire à l'utilisateur.

Ce que vous devez comprendre avec ce schéma, c'est que du point de vue du client, pour qui le serveur est comme une grosse boîte noire, la réponse envoyée par la JSP finale correspond à la requête vers la servlet de déconnexion qu'il a effectuée. C'est donc pour cette raison que l'utilisateur croit que la réponse est issue de la servlet de déconnexion, et que son navigateur lui affiche toujours l'URL de la page de déconnexion dans la barre d'adresse : il ne voit pas ce qu'il se passe côté serveur, et ne sait pas qu'en réalité sa requête a été baladée de servlet en servlet.

Voyons maintenant ce qui se passerait si nous utilisions une redirection vers la page de connexion à la place du *forwarding* dans la servlet de déconnexion :



Avec une redirection

1. l'utilisateur accède à la page de déconnexion depuis son navigateur ;
2. la servlet de déconnexion envoie une demande de redirection au navigateur vers la servlet de connexion, via un `sendRedirect( "/pro/connexion" )` ;
3. le navigateur de l'utilisateur exécute alors la redirection et effectue alors une nouvelle requête vers la servlet de connexion ;
4. la servlet de connexion transfère la requête vers la JSP du formulaire de connexion via un *forwarding* ;
5. la JSP renvoie le formulaire à l'utilisateur.

Cette fois, vous voyez bien que la réponse envoyée par la JSP finale correspond à la seconde requête effectuée par le navigateur, à savoir celle vers la servlet de connexion. Ainsi, l'URL affichée dans la barre d'adresse du navigateur est bien celle de la page de connexion, et l'utilisateur n'est pas dérouté.

Alors certes, dans le cas de notre page de déconnexion et de notre *forwarding*, le fait que le client ne soit pas au courant du cheminement de sa requête au sein du serveur n'a rien de troublant, seule l'URL n'est pas en accord avec l'affichage final. En effet, si le client appuie sur F5 et actualise la page, cela va appeler à nouveau la servlet de déconnexion, qui va supprimer sa session si elle existe, puis à nouveau faire un *forwarding*, etc. puis finir par afficher le formulaire de connexion à nouveau.

Seulement imaginez maintenant que nous n'avons plus affaire à un système de déconnexion, mais à un système de gestion de compte en banque, dans lequel la servlet de déconnexion deviendrait une servlet de transfert d'argent, et la servlet de connexion deviendrait une servlet d'affichage du solde du compte. Si nous gardons ce système de *forwarding*, alors après que le client effectue un transfert d'argent, il est redirigé de manière transparente vers l'affichage du solde de son compte. Et là, ça devient problématique : si le client ne fait pas attention, et qu'il actualise la page en pensant simplement actualiser l'affichage de son solde, il va en réalité à nouveau effectuer un transfert d'argent, puisque l'URL de son navigateur est restée figée sur la première servlet contactée...

Vous comprenez mieux maintenant pourquoi je vous avais conseillé d'utiliser **`<c:redirect/>`** plutôt que **`<jsp:forward/>`** dans le chapitre sur la JSTL Core, et pourquoi dans notre exemple j'ai ici mis en place une redirection HTTP via

`sendRedirect()` plutôt qu'un *forwarding* ?

Avant de poursuivre, éditez le code de votre servlet de déconnexion et remettez en place la redirection HTTP vers votre site préféré, comme je vous l'ai montré avant de faire cet aparté sur le *forwarding*.

Derrière les rideaux

La théorie : principe de fonctionnement

C'est bien joli tout ça, mais nous n'avons toujours pas abordé la question fatidique :

Comment fonctionnent les sessions ?

Jusqu'à présent, nous ne sommes pas inquiétés de ce qui se passe derrière les rideaux. Et pourtant croyez-moi, il y a de quoi faire !

La chose la plus importante à retenir, c'est que c'est vous qui contrôlez l'existence d'une session dans votre application. Un objet `HttpSession` dédié à un utilisateur sera créé ou récupéré **uniquement lorsque la page qu'il visite implique un appel à** `request.getSession()`, en d'autres termes uniquement lorsque vous aurez placé un tel appel dans votre code. En ce qui concerne la gestion de l'objet, c'est le conteneur de servlets qui va s'en charger, en le créant et le stockant en mémoire. Au passage, le serveur dispose d'un moyen pour identifier chaque session qu'il crée : il leur attribue un identifiant unique, que nous pouvons d'ailleurs retrouver via la méthode `session.getId()`.

Ensuite, le conteneur va mettre en place un élément particulier dans la réponse HTTP qu'il va renvoyer au client : un `Cookie`. Nous reviendrons plus tard sur ce ce que sont exactement ces cookies, et sur comment les manipuler. Pour le moment, voyez simplement un cookie comme un simple marqueur, un petit fichier texte qui :

- contient des informations envoyées par le serveur ;
- est stocké par votre navigateur, directement sur votre poste ;
- a obligatoirement un nom et une valeur.

En l'occurrence, le cookie mis en place lors de la gestion d'une session utilisateur par le serveur se nomme **JSESSIONID**, et contient l'identifiant de session unique en tant que valeur.

Pour résumer, le serveur va placer directement chez le client son identifiant de session. Et donc à chaque nouveau client, lorsqu'il crée une session, le serveur va envoyer son identifiant au navigateur du client.

Comment est géré ce cookie ?

Je vous l'ai déjà dit, nous allons y revenir plus en détails dans un prochain chapitre. Toutefois, nous pouvons déjà esquisser brièvement ce qui se passe dans les coulisses. La spécification du cookie HTTP, qui constitue un contrat auquel tout navigateur web décent ainsi que tout serveur web doit adhérer, est très claire : elle demande au navigateur de renvoyer ce cookie dans les requêtes suivantes tant que le cookie reste valide.

Voilà donc la clé du système : le conteneur de servlets va analyser chaque requête HTTP entrante, y chercher le cookie ayant pour nom **JSESSIONID** et utiliser sa valeur, c'est-à-dire l'identifiant de session, afin de récupérer l'objet `HttpSession` associé dans la mémoire du serveur.

Quand les données ainsi stockées deviennent-elles obsolètes ?

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

Côté client, le cookie de session a une durée de vie également, qui par défaut est limitée au temps durant lequel le navigateur reste ouvert. Lorsque le client ferme son navigateur, le cookie est donc détruit côté client. Si le client ouvre à nouveau son navigateur, le cookie associé à la précédente session ne sera alors plus envoyé. Nous revenons alors au principe général que je

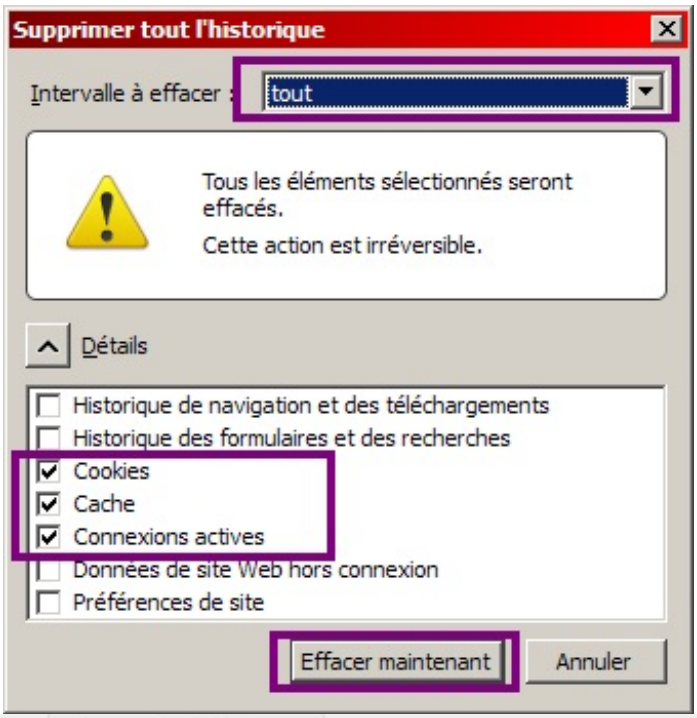
vous ai énoncé quelques lignes plus tôt : un appel à `request.getSession()` retournerait alors un nouvel objet `HttpSession`, et mettrait ainsi en place un nouveau cookie contenant un nouvel identifiant de session.

Plutôt que de vous ressortir les schémas précédents en modifiant et complétant leurs légende et explications pour y faire apparaître la gestion du cookie, je vais vous faire pratiquer ! Nous allons directement tester notre petit système de connexion, et analyser ce qui se trame dans les entrailles des échanges HTTP...

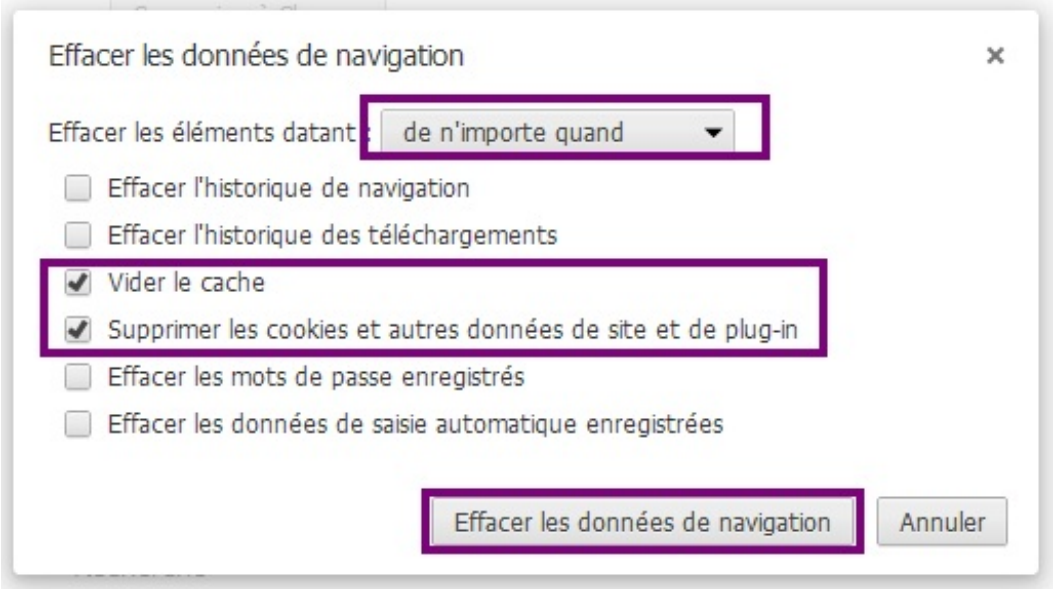
La pratique : scrutons nos requêtes et réponses

Pour commencer, nous allons reprendre notre exemple de connexion et analyser les échanges qu'il engendre :

1. redémarrez votre serveur Tomcat ;
2. fermez votre navigateur, puis ouvrez-le à nouveau ;
3. supprimez toutes les données qui y sont automatiquement enregistrées. Depuis Firefox ou Chrome, il suffit d'appuyer simultanément sur Ctrl + Maj + Suppr pour qu'un menu de suppression du cache, des cookies et autres données diverses apparaisse :



Suppression des données sous firefox

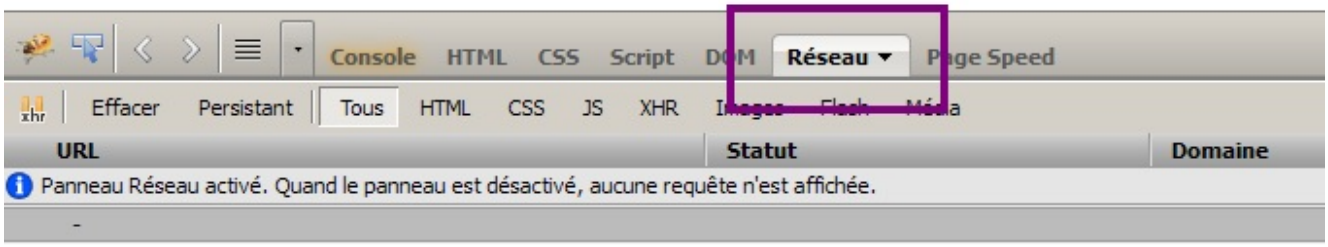


Suppression des données sous Chrome

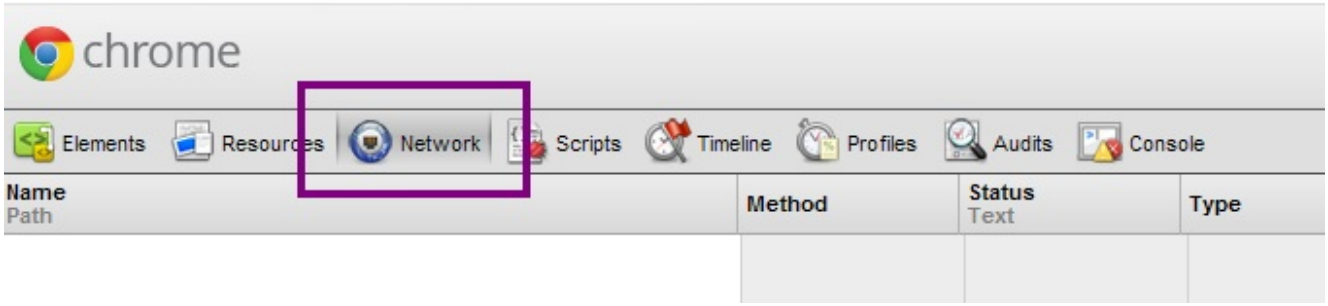
4. ouvrez un nouvel onglet vide, et appuyez alors sur F12 pour lancer Firebug depuis Firefox, ou l'outil équivalent intégré

depuis Chrome ;

5. cliquez alors sur l'onglet **Réseau** de Firebug, ou sur l'onglet **Network** de l'outil intégré à Chrome :



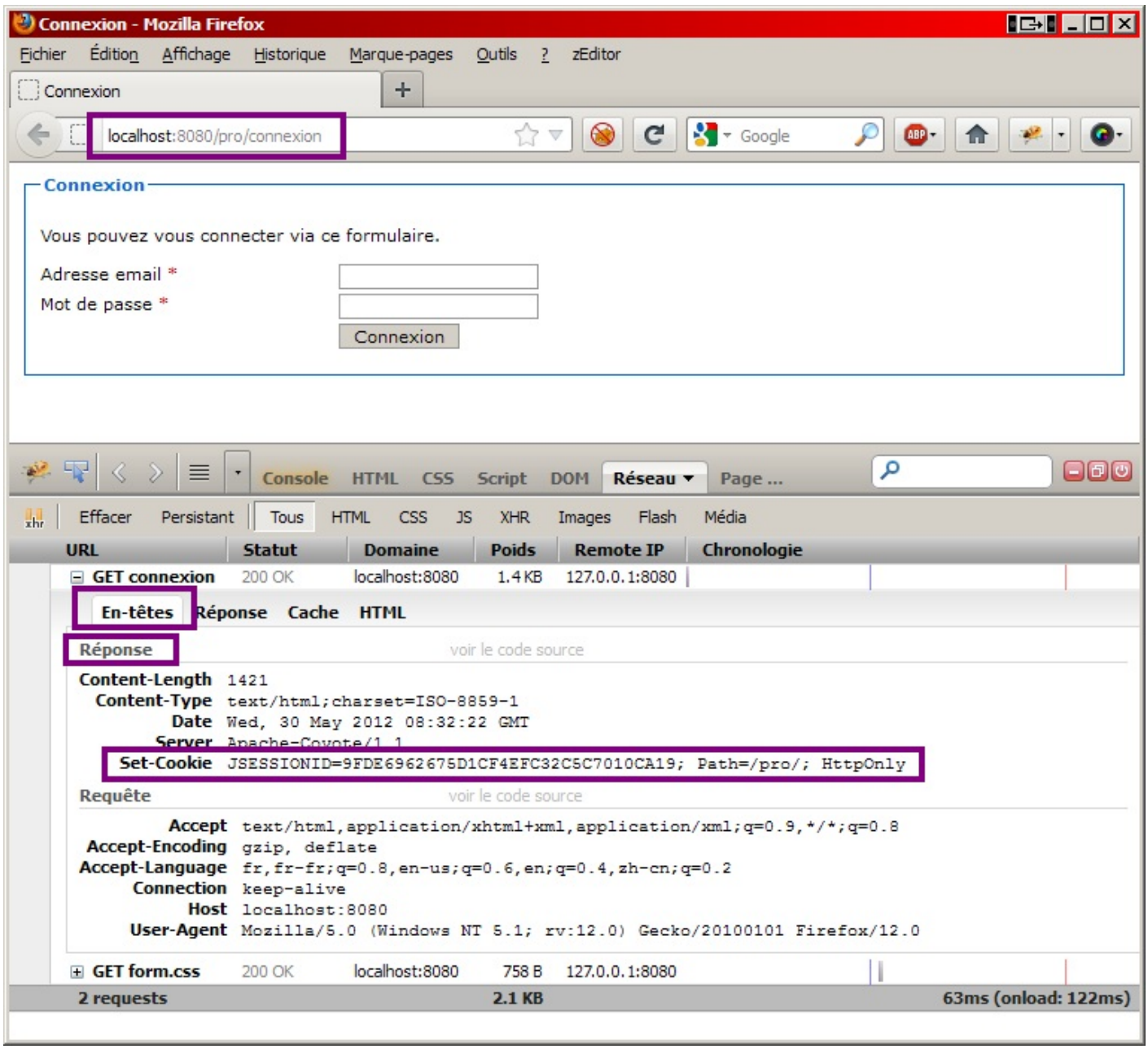
Onglet réseau sous Firebug



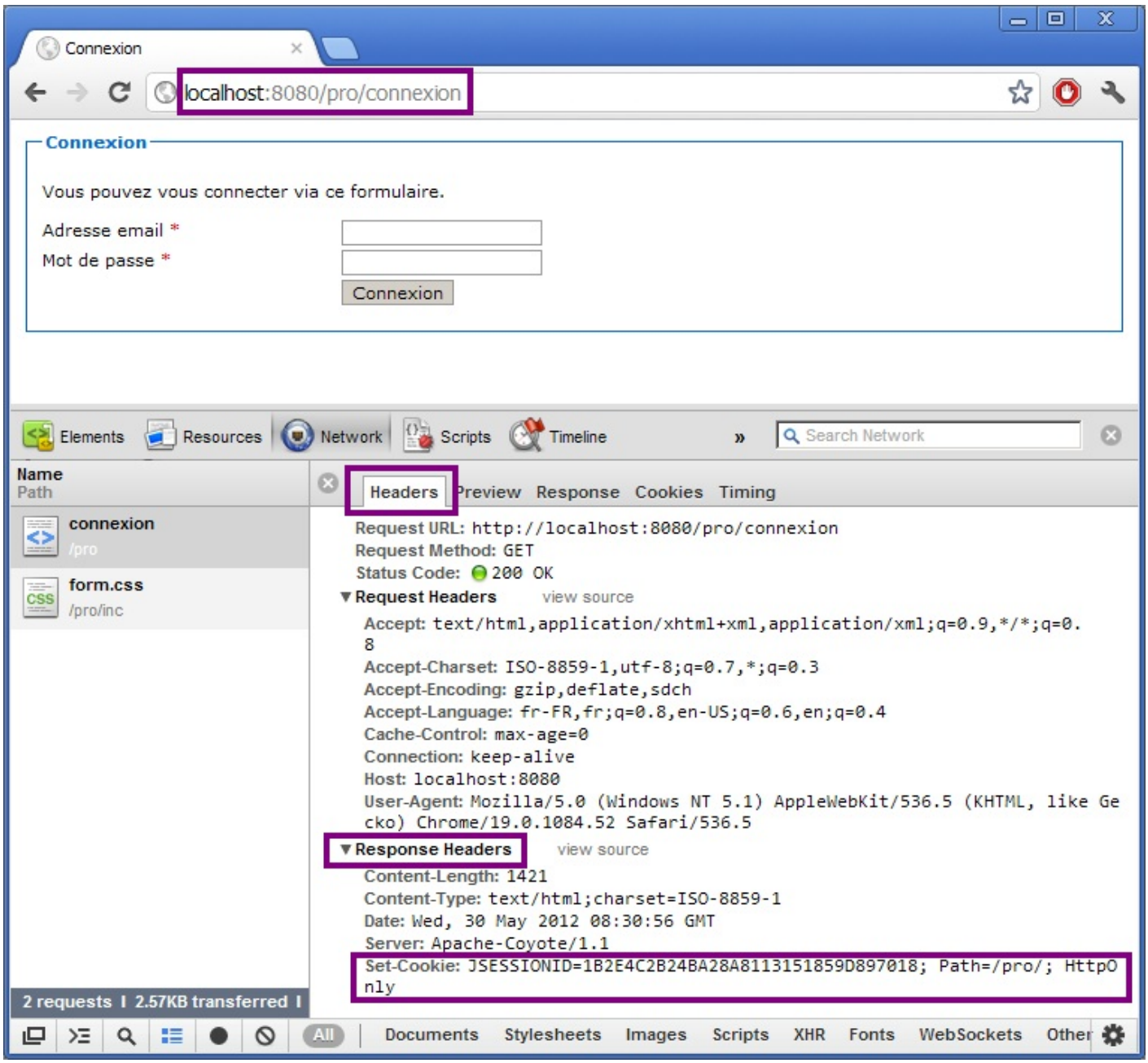
Onglet network sous Chrome

Le tout premier accès

Rendez-vous ensuite sur la page http://localhost:8080/pro/connexion. Les données enregistrées côté client ont été effacées, et le serveur a été redémarré, il s'agit donc ici de notre toute première visite sur une page du site. En outre, nous savons que la servlet **Connexion** associée à cette page contient un appel à `request.getSession()`. Observez alors ce qui s'affiche dans votre outil :



Cookie dans la réponse avec l'outil Firebug.



Cookie dans la réponse avec l'outil de Chrome.

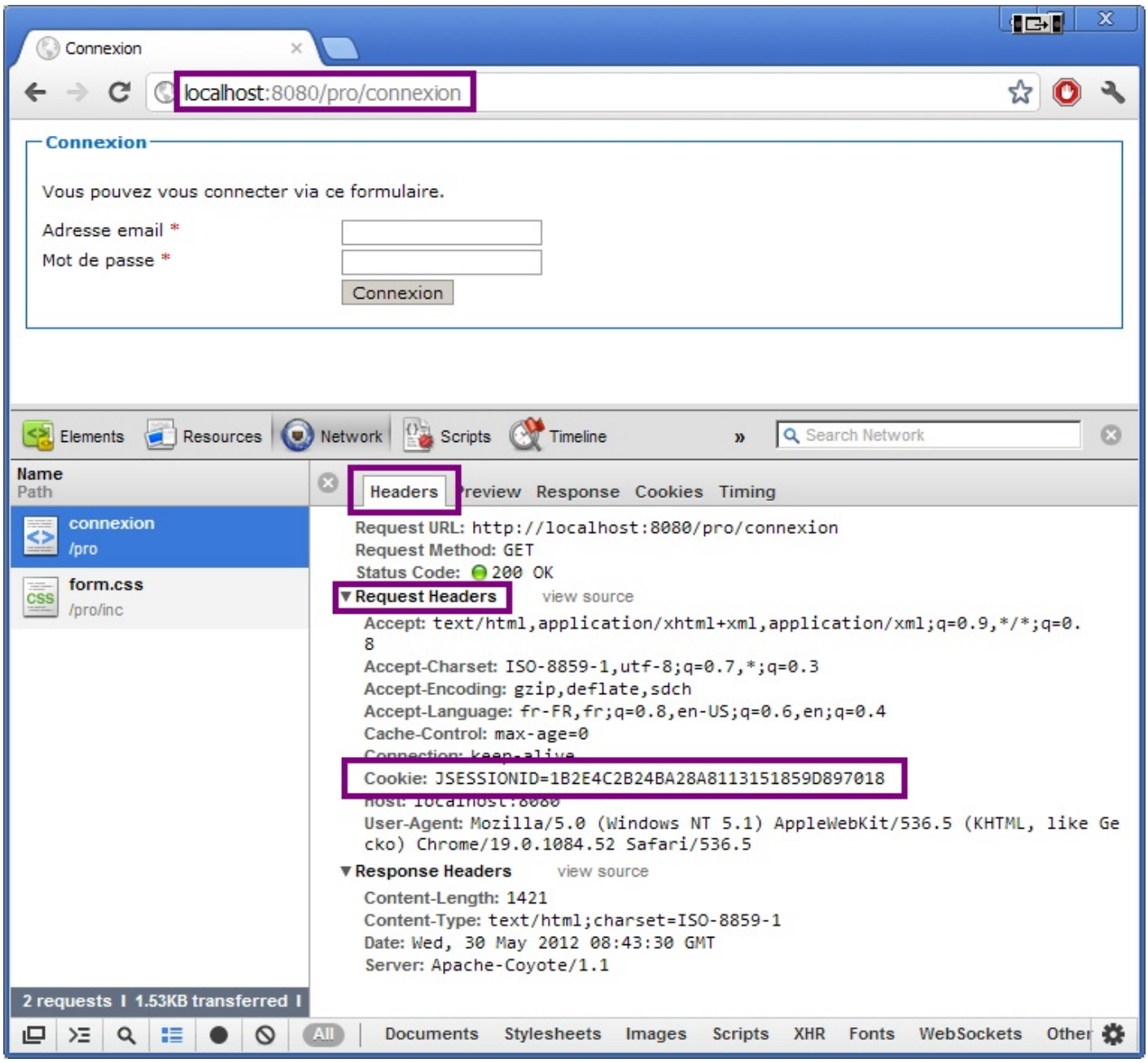
Vous pouvez ici remarquer plusieurs choses importantes :

- la réponse renvoyée par le serveur contient une instruction **Set-Cookie**, destinée à mettre en place le cookie de session dans le navigateur du client ;
- le nom du cookie est bien **JSESSIONID**, et sa valeur est bien un long identifiant unique ;
- bien que je sois le seul réel client qui accède au site, le serveur considère mes visites depuis Firefox et Chrome comme étant issues de deux clients distincts, et génère donc deux sessions différentes. Vérifiez les identifiants, ils sont bien différents d'un écran à l'autre.

Dorénavant, je vous afficherai uniquement des captures d'écran réalisées avec l'outil de Chrome, pour ne pas surcharger d'images ce chapitre. Si vous utilisez Firebug, reportez vous à la capture précédente si jamais vous ne vous souvenez plus où regarder les informations relatives aux échanges HTTP.

L'accès suivant, avec la même session

Dans la foulée, rendez-vous à nouveau sur cette même page de connexion (actualisez la page via un appui sur F5 par exemple). Observez alors :



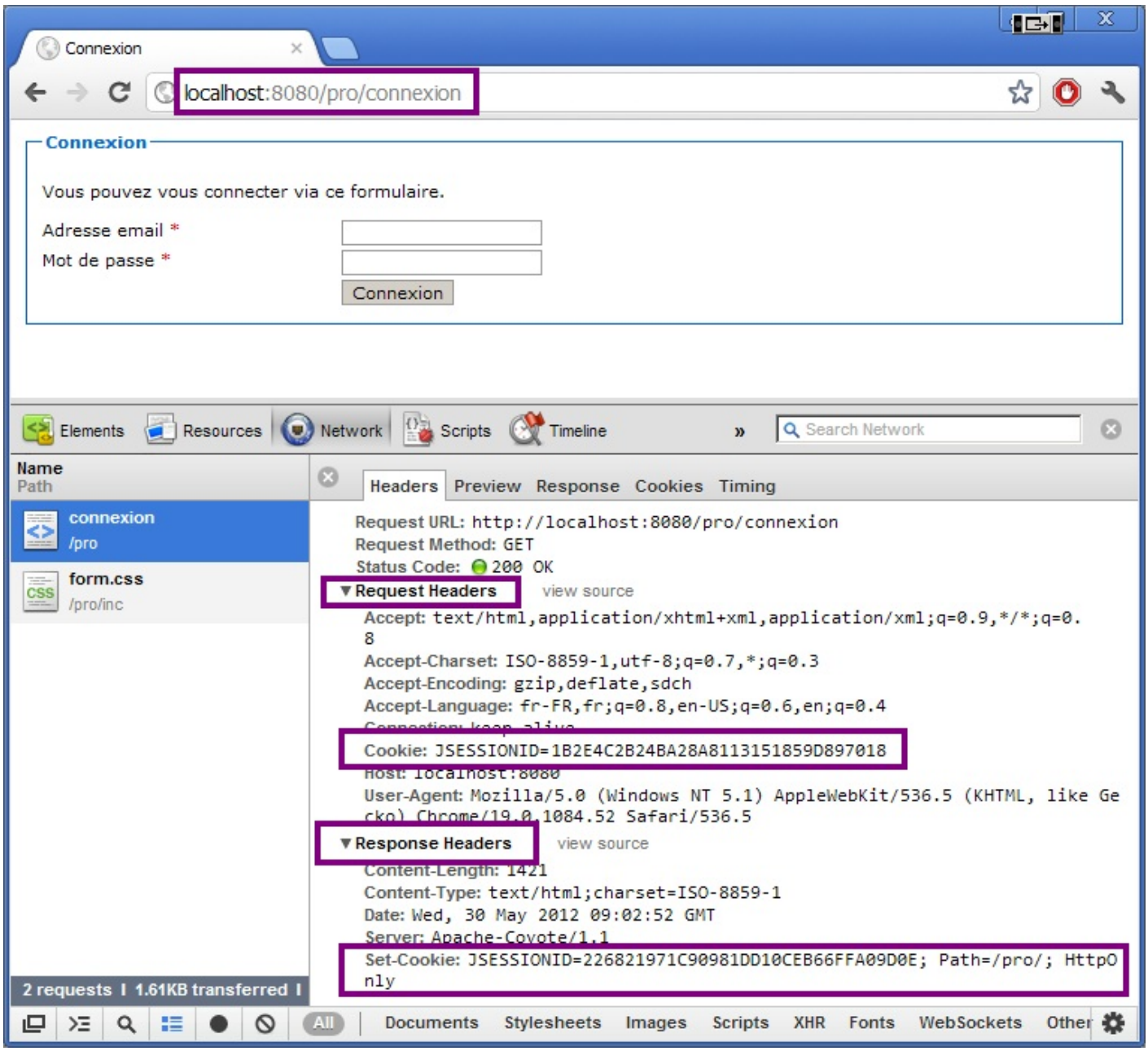
Cookie dans la requête.

Là encore, vous pouvez remarquer plusieurs choses importantes :

- un cookie est cette fois envoyé par le navigateur au serveur, dans le paramètre **Cookie** de l'en-tête de la requête HTTP effectuée ;
- sa valeur correspond à celle contenue dans le cookie envoyé par le serveur dans la réponse précédente ;
- après réception de la première réponse contenant l'instruction **Set-Cookie**, le navigateur avait donc bien sauvegardé le cookie généré par le serveur, et le renvoie automatiquement lors des requêtes suivantes ;
- dans la deuxième réponse du serveur, il n'y a cette fois plus d'instruction **Set-Cookie** : le serveur ayant reçu un cookie nommé **JSESSIONID** depuis le client, et ayant trouvé dans sa mémoire une session correspondant à l'identifiant contenu dans la valeur du cookie, il sait que le client a déjà enregistré la session en cours et qu'il n'est pas nécessaire de demander à nouveau la mise en place d'un cookie !

L'accès suivant, après une déconnexion !

Rendez-vous maintenant sur la page http://localhost:8080/pro/deconnexion, puis retournez ensuite sur http://localhost:8080/pro/connexion. Observez alors :



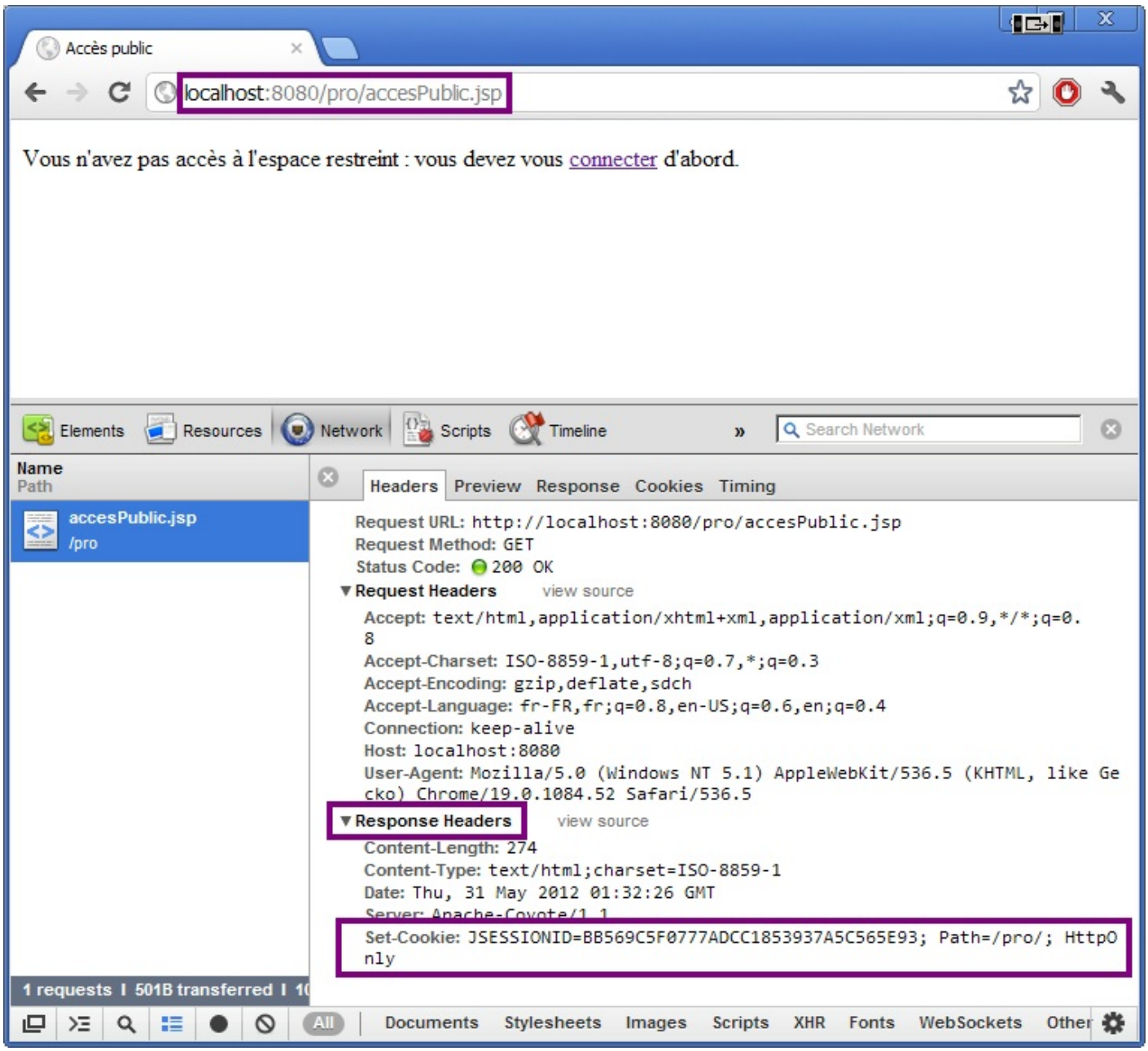
Cookie dans la requête et la réponse.

Cette fois encore, vous pouvez remarquer plusieurs choses importantes :

- deux cookies nommés **JSESSIONID** interviennent : un dans la requête et un dans la réponse ;
- la valeur de celui présent dans la requête contient l'identifiant de notre précédente session. Puisque nous n'avons pas fermé notre navigateur ni supprimé les cookies enregistrés, le navigateur considère que la session est toujours ouverte côté serveur, et envoie donc par défaut le cookie qu'il avait enregistré lors de l'échange précédent !
- la valeur de celui présent dans la réponse contient un nouvel identifiant de session. Le serveur ayant supprimé la session de sa mémoire lors de la déconnexion du client (souvenez-vous du code de notre servlet de déconnexion), il ne trouve aucune session qui corresponde à l'identifiant envoyé par le navigateur dans le cookie de la requête. Il crée donc une nouvelle session, et demande aussitôt au navigateur de remplacer le cookie existant par celui contenant le nouveau numéro de session, toujours via l'instruction **Set-Cookie** de la réponse renvoyée !

L'accès à une page sans session

Nous allons cette fois accéder à une page qui n'implique aucun appel à `request.getSession()`. Redémarrez Tomcat, effacez les données de votre navigateur via un Ctrl + Maj + Suppr, et rendez-vous alors sur la page http://localhost:8080/pro/accesPublic.jsp. Observez alors :



Création de session par la JSP.

Pourquoi le serveur demande-t-il la mise en place d'un cookie de session dans le navigateur ?!

En effet, c'est un comportement troublant ! Je vous ai annoncé qu'une session n'existait que lorsqu'un appel à `request.getSession()` était effectué. Or, le contenu de notre page **accesPublic.jsp** ne fait pas intervenir de session, et aucune servlet ne lui est associée : d'où sort cette session ? Eh bien rassurez-vous, je ne vous ai pas menti : c'est bien vous qui contrôlez la création de la session. Seulement voilà, il existe un comportement qui vous est encore inconnu, celui d'une page JSP : **par défaut, une page JSP va toujours tenter de créer ou récupérer une session**.

Nous pouvons d'ailleurs le vérifier en jetant un oeil au code de la servlet auto-générée par Tomcat. Nous l'avions déjà fait lorsque nous avions découvert les JSP pour la première fois, et je vous avais alors fait remarquer que le répertoire contenant ces fichiers pouvait varier selon votre installation et votre système. Voici un extrait du code du fichier **accesPublic_jsp.java** généré :

Code : Java - C:\eclipse\workspace\.metadata\.plugins\org.eclipse.wst.server.core\tmp0\work\Catalina\localhost\pro\org\apache\jsp\accesPublic_jsp.java

```
javax.servlet.http.HttpSession session = null;

...
```

```
session = pageContext.getSession();
```

Voilà donc l'explication de l'existence d'une session lors de l'accès à notre page JSP : dans le code auto-généré, il existe un appel à la méthode `getSession()` !

Comment éviter la création automatique d'une session depuis une page JSP ?

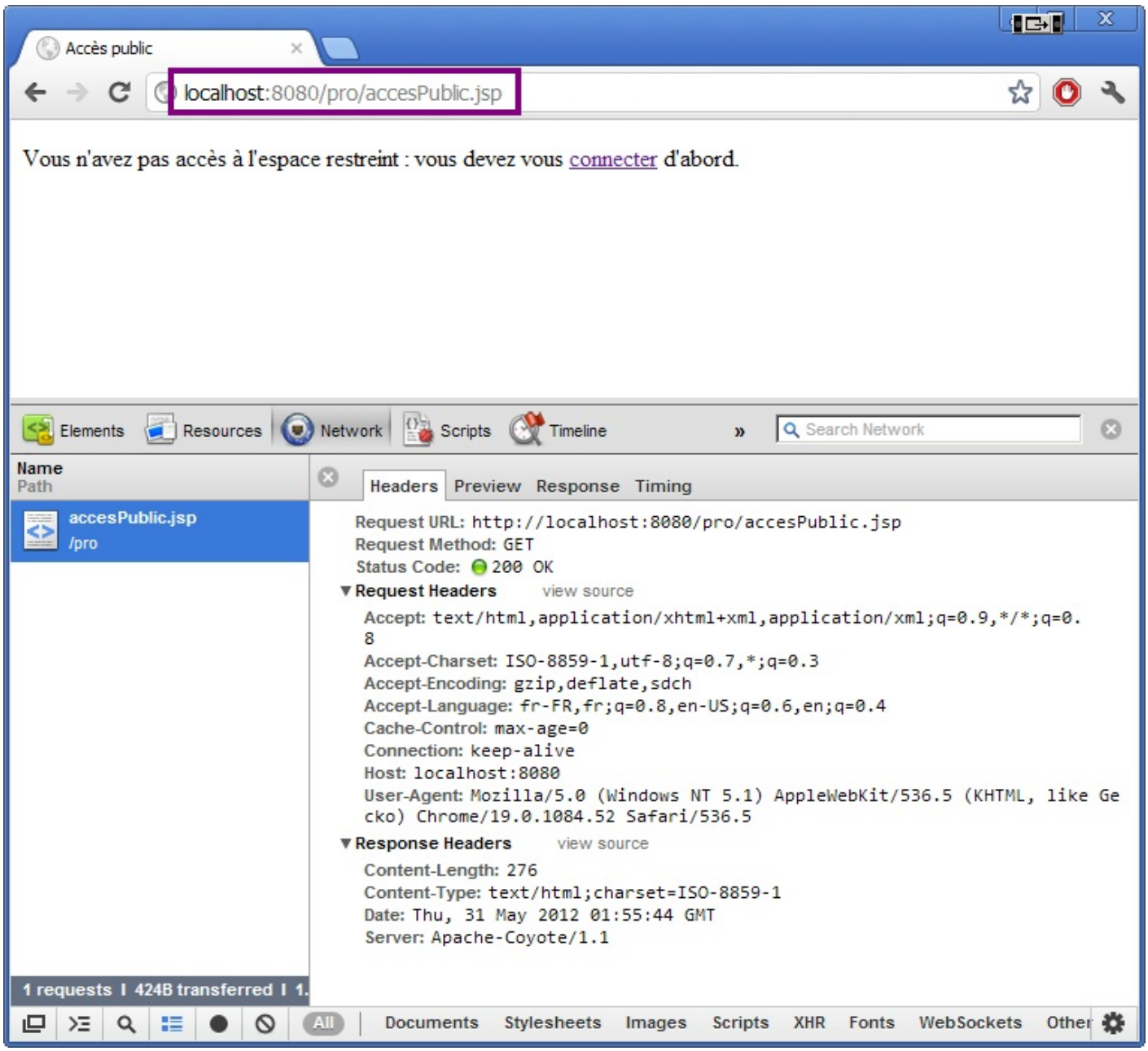
La solution qui s'offre à nous est l'utilisation de la directive **page**. Voici la ligne à ajouter en début de page pour empêcher la création d'une session :

Code : JSP

```
<%@ page session="false" %>
```

Toutefois, comprenez bien : cette directive désactive la session sur tout la page JSP. Autrement dit, en procédant ainsi vous interdisez la manipulation de sessions depuis votre page JSP. Dans ce cas précis, tout va bien, notre page n'en fait pas intervenir. Mais dans une page qui accède à des objets présents en session, vous ne devez bien évidemment pas mettre en place cette directive !

Éditez-donc votre page **accesPublic.jsp** et ajoutez-y cette directive en début de code. Redémarrez alors Tomcat, effacez les données de votre navigateur via un Ctrl + Maj + Suppr, et rendez-vous à nouveau sur la page http://localhost:8080/pro/accesPublic.jsp. Observez cette fois :



Désactivation de la session dans la JSP.

Vous pouvez cette fois remarquer qu'aucun cookie n'intervient dans l'échange HTTP ! Le serveur ne cherche pas à créer ni récupérer de session, et par conséquent il ne demande pas la mise en place d'un cookie dans le navigateur de l'utilisateur.

Dans un réel projet, ce genre d'optimisations n'a absolument aucun impact sur une application de petite envergure, vous pouvez très bien vous en passer. Mais sur une application à très forte fréquentation ou simplement sur une page à très fort trafic, en désactivant l'accès à la session lorsqu'elle n'est pas utilisée, vous pouvez gagner en performances et en espace mémoire disponibles : vous empêcherez ainsi la création d'objets inutilisés par le serveur, qui occuperont de la place jusqu'à ce qu'ils soient détruits après la période d'inactivité dépassée.

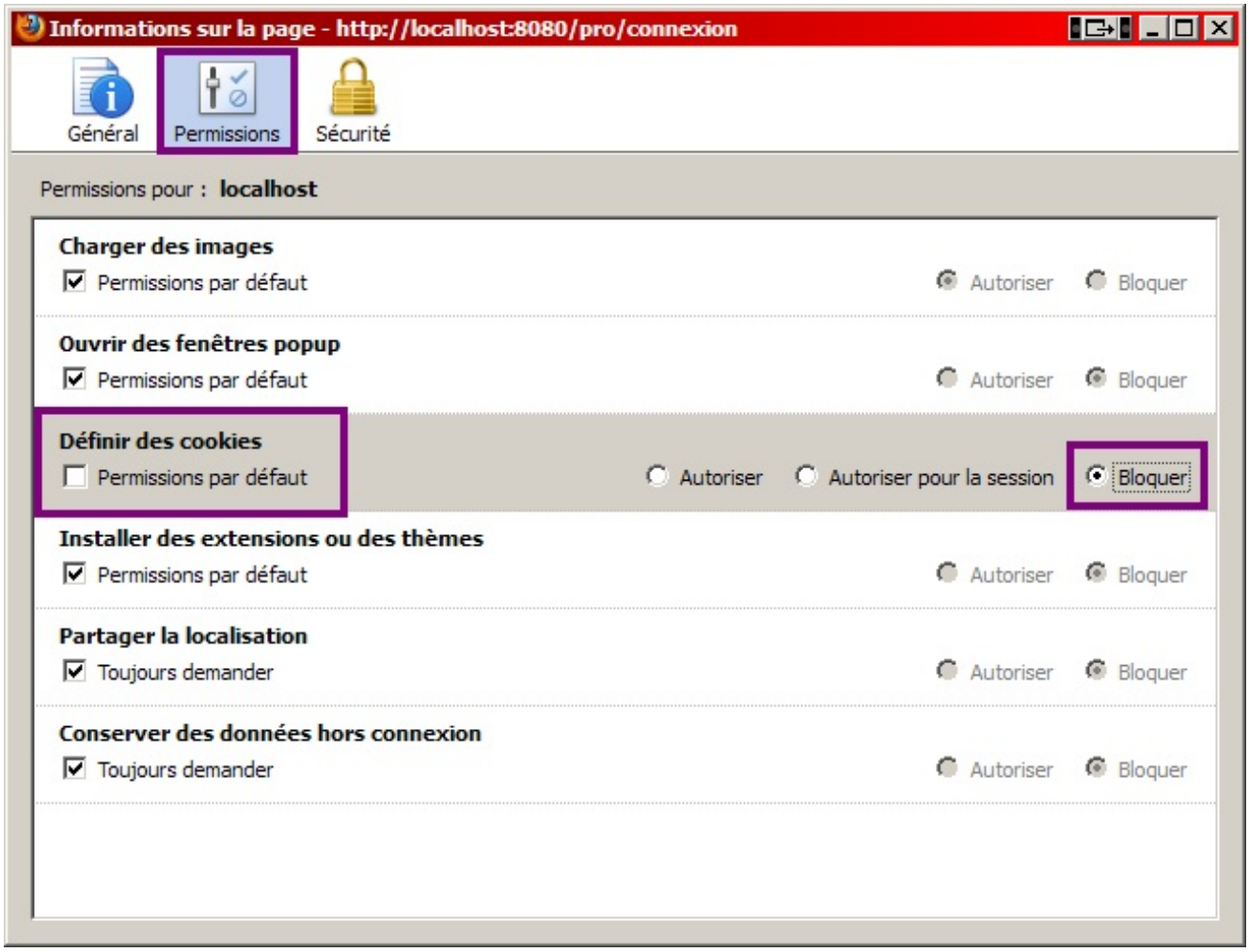
L'accès à une page sans cookie

Dernier scenario et non des moindres, l'accès à une page faisant intervenir un appel à `request.getSession()` depuis un navigateur qui n'accepte pas les cookies ! Eh oui, tous les navigateurs ne gardent pas leurs portes ouvertes, et certains refusent la sauvegarde de données sous forme de cookies. Procédez comme suit pour bloquer les cookies depuis votre navigateur.

Depuis Firefox :

1. Allez sur la page http://localhost:8080/pro/connexion.
2. Faites un clic droit dans la page et sélectionnez **Informations sur la page**.
3. Sélectionnez alors le panneau **Permissions**.

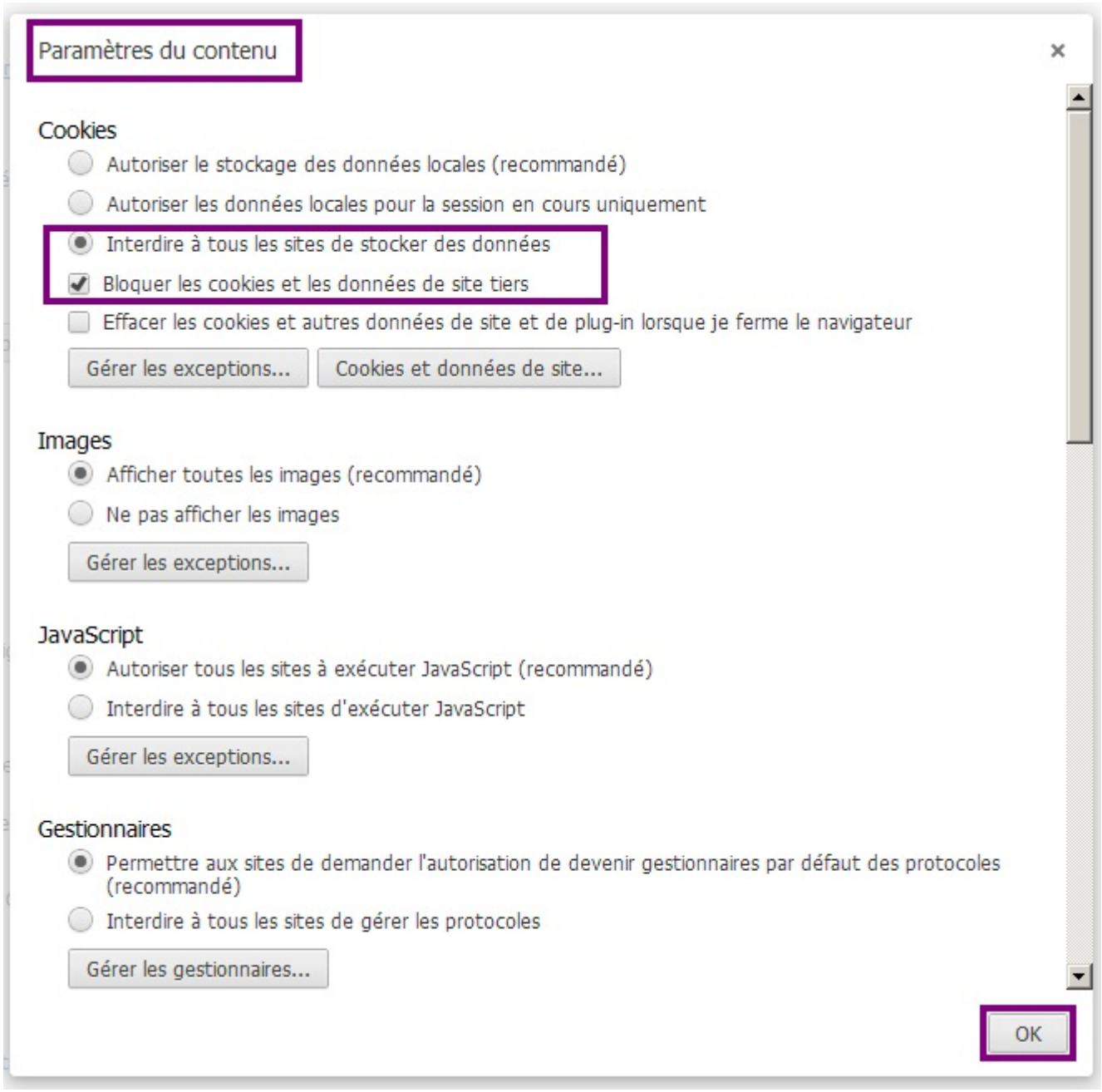
4. Sous **Définir des cookies**, décochez **Permissions par défaut** et cochez **Bloquer** :



5. Fermer la fenêtre "Informations sur la page".

Depuis Chrome :

1. Cliquez sur l'icône représentant une clé à molette qui est située dans la barre d'outils du navigateur.
2. Sélectionnez **Paramètres**.
3. Cliquez sur **Afficher les paramètres avancés**.
4. Dans la section "Confidentialité", cliquez sur le bouton **Paramètres de contenu**.
5. Dans la section "Cookies", modifiez les paramètres suivants :



Redémarrez ensuite Tomcat, effacez les données de votre navigateur via un Ctrl + Maj + Suppr, et rendez-vous sur la page http://localhost:8080/pro/connexion. Vous observerez alors que la réponse du serveur contient une instruction **Set-Cookie**. Actualisez maintenant la page en appuyant sur F5, et vous constaterez cette fois que la requête envoyée par votre navigateur ne contient pas de cookie, et que la réponse du serveur contient à nouveau une instruction **Set-Cookie** présentant un identifiant de session différent ! Et c'est tout à fait logique :

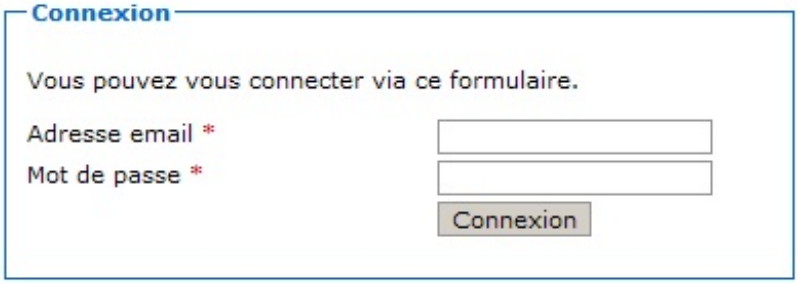
- le navigateur n'accepte plus les cookies, il n'a donc pas enregistré le premier identifiant envoyé par le serveur dans la première réponse. Par conséquent, il n'a pas envoyé d'identifiant dans la requête suivante ;
- le serveur ne trouvant aucune information de session dans la seconde requête envoyée par le navigateur du client, il le considère comme un nouveau visiteur, crée une nouvelle session et lui demande d'en enregistrer le nouvel identifiant dans un cookie.

Ok, c'est logique. Mais dans ce cas, comment le serveur peut-il associer une session à un utilisateur ?

Voilà en effet une excellente question : comment le serveur va-t-il être capable de retrouver des informations en session s'il n'est pas capable de reconnaître un visiteur d'une requête à l'autre ? Étant donné l'état actuel de notre code, la réponse est simple : il ne peut pas ! D'ailleurs, vous pouvez vous en rendre compte simplement :

1. rendez-vous sur la page de connexion, saisissez des données correctes et validez le formulaire. Observez alors :

2. ouvrez alors un nouvel onglet, et rendez-vous à nouveau sur la page de connexion. Observez cette fois :



Lors de nos précédents tests, dans la partie sur les vérifications, le formulaire ré-affichait l'adresse mail avec laquelle vous vous étiez connecté auparavant. Cette fois, aucune information n'est ré-affichée et le formulaire de connexion apparaît à nouveau vierge. Vous constatez donc bien l'incapacité du serveur à vous reconnaître !

Pas de panique, nous allons y remédier très simplement. Dans notre page **connexion.jsp**, nous allons modifier une ligne de code :

Code : JSP - /WEB-INF/connexion.jsp

```
<!-- Dans la page connexion.jsp, remplacez la ligne suivante : -->
<form method="post" action="connexion">

<!-- Par cette ligne : -->
<form method="post" action="<c:url value="/connexion" />">
```

Si vous reconnaissez ici la balise **<c:url/>** de la JSTL Core, vous devez également vous souvenir qu'elle est équipée pour la gestion automatique des sessions. Je vous avais en effet déjà expliqué que cette balise avait l'effet suivant :

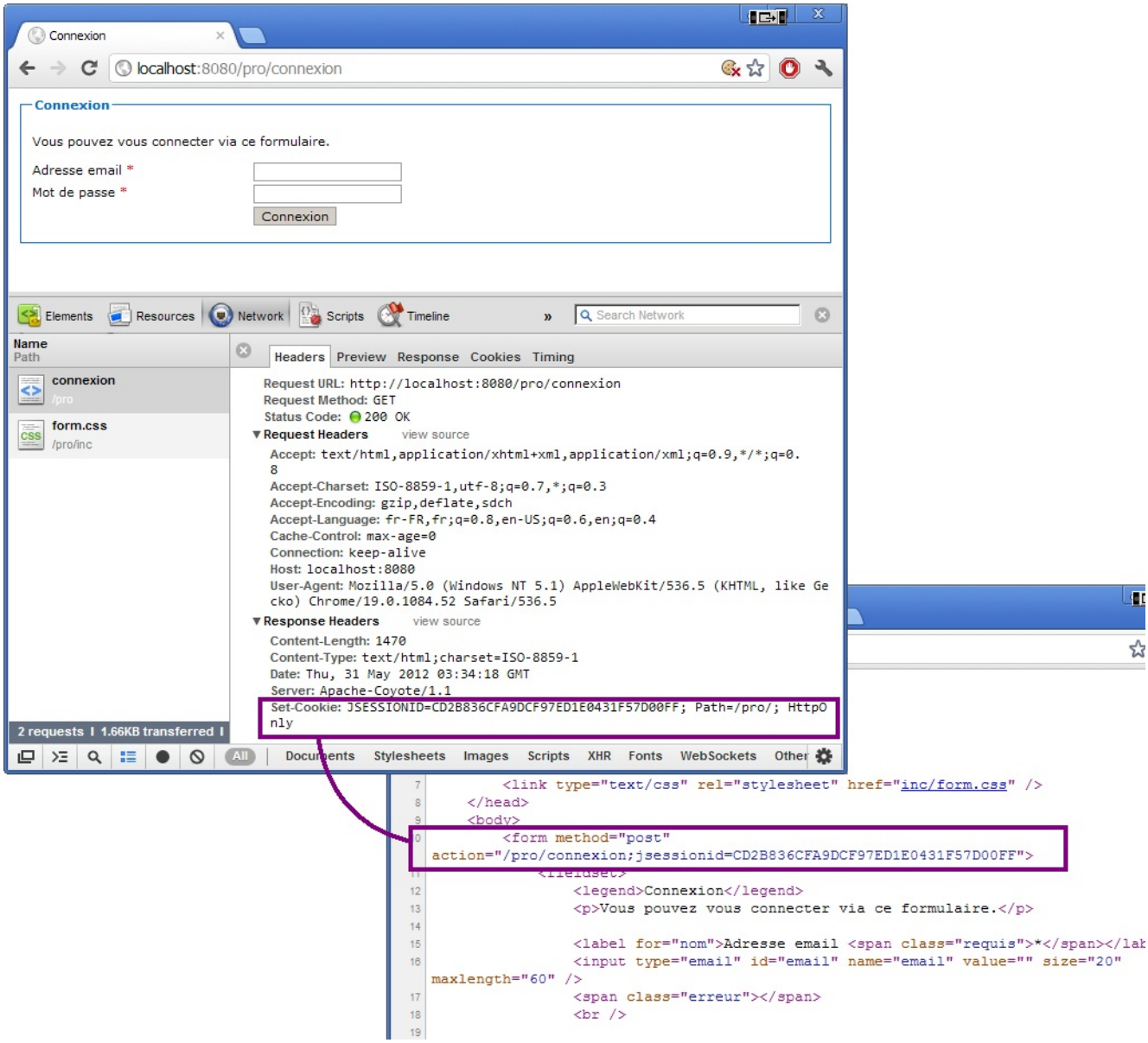
Code : JSP

```
<%-- L'url ainsi générée --%>
<c:url value="test.jsp" />

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

<%-- Et sera rendue sous cette forme si le cookie est absent --%>
test.jsp;jsessionid=BB569C7F07C5E887A4D
```

Et ça, c'est exactement ce dont nous avons besoin ! Puisque notre navigateur n'accepte plus les cookies, nous n'avons pas d'autre choix que de faire passer l'identifiant de session directement au sein de l'URL.

Une fois la modification sur la page **connexion.jsp** effectuée, suivez ce scénario de tests :

- rendez-vous à nouveau sur la page http://localhost:8080/pro/connexion, et regardez à la fois la réponse envoyée par le serveur et le code source de la page de connexion. Vous constaterez alors que puisque le serveur ne détecte aucun cookie présent chez le client, il va d'un côté tenter de passer l'identifiant de session via l'instruction **Set-Cookie**, et de l'autre générer une URL précisant l'identifiant de session. Voyez plutôt :



- connectez-vous alors avec des données valides. Vous retrouverez alors dans la barre d'adresse de votre navigateur l'URL modifiée par la balise **`<c:url/>`**, contenant l'identifiant de session passé par le serveur :

- ouvrez un nouvel onglet, et copiez/collez l'URL dans la barre d'adresse pour y ouvrir à nouveau la page de connexion en conservant le **JSESSIONID**. Vous constaterez cette fois que le serveur vous a bien reconnu en se basant sur l'identifiant contenu dans l'URL que vous lui transmettez, et qu'il est capable de retrouver l'adresse mail avec laquelle vous vous êtes

connectés :

- accédez maintenant à la page http://localhost:8080/pro/connexion sans préciser le **JSESSIONID** dans l'URL, et constatez que le serveur est à nouveau incapable de vous reconnaître et vous affiche un formulaire vierge !

Nous en avons enfin terminé avec notre batterie de tests, et avec tout ce que vous avez découvert, les sessions n'ont maintenant presque plus aucun secret pour vous !

Le bilan

Voici un récapitulatif des points importants que vous devez bien comprendre et assimiler :

- une session est un espace mémoire alloué sur le serveur, et dont le contenu n'est accessible que depuis le serveur ;
- afin de savoir quel client est associé à telle session créée, le serveur transmet au client l'identifiant de la session qui lui est dédiée, le **JSESSIONID**, dans les en-têtes de la réponse HTTP sous forme d'un cookie ;
- si le navigateur accepte les cookies, il stocke alors ce cookie contenant l'identifiant de session, et le retransmet au serveur dans les en-têtes de chaque requête HTTP qui va suivre ;
- si le serveur lui renvoie un nouveau numéro, autrement dit si le serveur a fermé l'ancienne session et en a ouvert une nouvelle, alors le navigateur remplace l'ancien numéro stocké par ce nouveau numéro, en écrasant l'ancien cookie par le nouveau ;
- si le navigateur n'accepte pas les cookies, alors le serveur dispose d'un autre moyen pour identifier le client : il est capable de chercher l'identifiant directement dans l'URL de la requête, et pas uniquement dans ses en-têtes ;
- il suffit au développeur de manipuler correctement les URL qu'il met en place dans son code pour permettre une gestion continue des sessions, indépendante de l'acceptation ou non des cookies côté client.

Nous en avons terminé avec la découverte des sessions, et j'espère que cet exemple pratique vous aura fait comprendre simplement les concepts clés.

Vous avez maintenant un outil puissant à votre disposition, et ne vous fiez pas uniquement au titre du chapitre : s'il est vrai que les sessions peuvent être utilisées pour réaliser un système de connexion, ce n'est absolument pas le seul usage qu'il est possible d'en faire ! Nous nous en sommes ici servis pour stocker les informations relatives à l'identité de nos visiteurs, mais elles peuvent bien entendu vous servir pour stocker n'importe quoi : les commandes en cours des clients dans le cas d'un site d'e-commerce, l'historique des transactions bancaires des clients dans le cas d'un site de finance, la liste des dernières interventions dans le cas du site du zéro...

Le filtre : créez un espace membre

Maintenant que nous savons manipuler les sessions et connecter nos utilisateurs, il serait intéressant de pouvoir mettre en place un espace membre dans notre application : c'est un ensemble de pages web qui est uniquement accessible aux utilisateurs connectés.

Pour ce faire, nous allons commencer par étudier le principe sur une seule page, via une servlet classique. Puis nous allons étendre ce système à tout un ensemble de page, et découvrir un nouveau composant cousin de la servlet : **le filtre** !

Restreindre l'accès à une page

Ce principe est massivement utilisé dans la plupart des applications web : les utilisateurs enregistrés et connectés à un site ont bien souvent accès à plus de contenu et de fonctionnalités que les simples visiteurs.

Comment mettre en place une telle restriction d'accès ?

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

Les pages d'exemple

Mettons en place pour commencer deux pages JSP :

- une dont nous allons plus tard restreindre l'accès aux utilisateurs connectés uniquement, nommée **accesRestreint.jsp** et placée sous **/WEB-INF** ;
- une qui sera accessible à tous les visiteurs, nommée **accesPublic.jsp** et placée sous la racine du projet (symbolisée par le dossier **WebContent** sous Eclipse).

Le contenu de ces pages importe peu, voici les exemples basiques que je vous propose :

Code : JSP - /WEB-INF/accesRestreint.jsp

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Accès restreint</title>
    </head>
    <body>
<p>Vous êtes connecté(e) avec l'adresse
${sessionScope.sessionUtilisateur.email}, vous avez bien accès à
l'espace restreint.</p>
    </body>
</html>
```

Code : JSP - /accesPublic.jsp

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Accès public</title>
    </head>
    <body>
        <p>Vous n'avez pas accès à l'espace restreint : vous devez
vous <a href="connexion">connecter</a> d'abord. </p>
    </body>
</html>
```

Rien de particulier à signaler ici, si ce n'est l'utilisation d'une expression EL dans la page restreinte, afin d'accéder à l'adresse mail de l'utilisateur enregistré en session, à travers l'objet implicite **sessionScope**.

La servlet de contrôle

Ce qu'il nous faut réaliser maintenant, c'est ce fameux contrôle sur le contenu de la session avant d'autoriser l'accès à la page **accesRestreint.jsp**. Voyez plutôt :

Code : Java - com.sdzee.servlets.Restriction

```
package com.sdzee.servlets;

import java.io.IOException;

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

public class Restriction extends HttpServlet {
    public static final String ACCES_PUBLIC     =
"/accesPublic.jsp";
    public static final String ACCES_RESTREINT  = "/WEB-
INF/accesRestreint.jsp";
    public static final String ATT_SESSION_USER =
"sessionUtilisateur";

    public void doGet( HttpServletRequest request,
HttpServletResponse response ) throws ServletException, IOException
{
        /* Récupération de la session depuis la requête */
        HttpSession session = request.getSession();

        /**
* Si l'objet utilisateur n'existe pas dans la session en cours,
alors
* l'utilisateur n'est pas connecté.
*/
if ( session.getAttribute( ATT_SESSION_USER ) == null ) {
            /* Redirection vers la page publique */
            response.sendRedirect( request.getContextPath() +
ACCES_PUBLIC );
} else {
            /* Affichage de la page restreinte */
            this.getServletContext().getRequestDispatcher(
ACCES_RESTREINT ).forward( request, response );
        }
    }
}
```

Comme vous le voyez, le procédé est très simple : il suffit de récupérer la session, de tester si l'attribut **sessionUtilisateur** y existe déjà, et de rediriger vers la page restreinte ou publique selon le résultat du test.

Remarquez au passage l'emploi d'une redirection dans le cas de la page publique, et d'un *forwarding* dans le cas de la page privée. Vous devez maintenant être familiers avec le principe, je vous l'ai expliqué dans le chapitre précédent. Ici, si j'avais mis en place un *forwarding* en lieu et place de la redirection HTTP pour la page publique, alors l'URL dans le navigateur de l'utilisateur n'aurait pas été mise à jour en cas d'échec de l'accès à la page restreinte.

Par ailleurs, vous devez également prêter attention à la manière dont j'ai construit l'URL utilisée pour la redirection, à la ligne 26. Vous êtes déjà au courant que contrairement au *forwarding* qui est limité aux pages internes, la redirection HTTP permet quant à elle d'envoyer la requête à n'importe quelle page, y compris des pages provenant d'autres sites. Ce que vous ne savez pas encore, à moins d'avoir lu attentivement les documentations respectivement des méthodes `getRequestDispatcher()` et

`sendRedirect()`, c'est que l'URL prise en argument par la méthode de *forwarding* est relative au **contexte de l'application**, alors que l'URL prise en argument par la méthode de redirection est relative à la **racine de l'application** !

Concrètement, qu'est-ce que ça implique ?

Cette différence est très importante :

- l'URL passée à la méthode `getRequestDispatcher()` doit être interne à l'application. En l'occurrence, dans notre projet cela signifie qu'il est impossible de préciser une URL qui cible une page en dehors du projet **pro**. Ainsi, un appel à `getRequestDispatcher( "/accesPublic.jsp" )` ciblera automatiquement la page /pro/accesPublic.jsp, vous n'avez pas à préciser vous-même le contexte /pro ;

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

Bref, vous l'aurez compris, vous devez être attentif aux méthodes que vous employez et à la manière dont elles vont gérer les URL que vous leur transmettez. Entre les URL absolues, les URL relatives à la racine de l'application, les URL relatives au contexte de l'application et les URL relatives au répertoire courant, il est parfois difficile de ne pas s'emmêler les crayons !

Pour terminer, voici sa configuration dans le fichier **web.xml** de notre application :

Code : XML - /WEB-INF/web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
 <servlet>
  <servlet-name>Inscription</servlet-name>
  <servlet-class>com.sdzee.servlets.Inscription</servlet-class>
 </servlet>
 <servlet>
  <servlet-name>Connexion</servlet-name>
  <servlet-class>com.sdzee.servlets.Connexion</servlet-class>
 </servlet>
 <servlet>
  <servlet-name>Deconnexion</servlet-name>
  <servlet-class>com.sdzee.servlets.Deconnexion</servlet-class>
 </servlet>
<servlet>
<servlet-name>Restriction</servlet-name>
<servlet-class>com.sdzee.servlets.Restriction</servlet-class>
</servlet>

 <servlet-mapping>
  <servlet-name>Inscription</servlet-name>
  <url-pattern>/inscription</url-pattern>
 </servlet-mapping>
 <servlet-mapping>
  <servlet-name>Connexion</servlet-name>
  <url-pattern>/connexion</url-pattern>
 </servlet-mapping>
 <servlet-mapping>
  <servlet-name>Deconnexion</servlet-name>
  <url-pattern>/deconnexion</url-pattern>
 </servlet-mapping>
<servlet-mapping>
<servlet-name>Restriction</servlet-name>
<url-pattern>/restriction</url-pattern>
```

```
    </servlet-mapping>
    </web-app>
```

N'oubliez pas de redémarrer Tomcat pour que ces modifications soient prises en compte.

Test du système

Pour vérifier le bon fonctionnement de cet exemple d'accès restreint, suivez le scénario suivant :

1. redémarrez Tomcat, afin de faire disparaître toute session qui serait encore active ;
2. rendez-vous sur la page http://localhost:8080/pro/restriction, et constatez au passage la redirection (changement d'URL) :

Vous n'avez pas accès à l'espace restreint : vous devez vous connecter d'abord.

Acces restreint.

3. cliquez alors sur le lien vers la page de connexion, entrez des informations valides et connectez-vous :

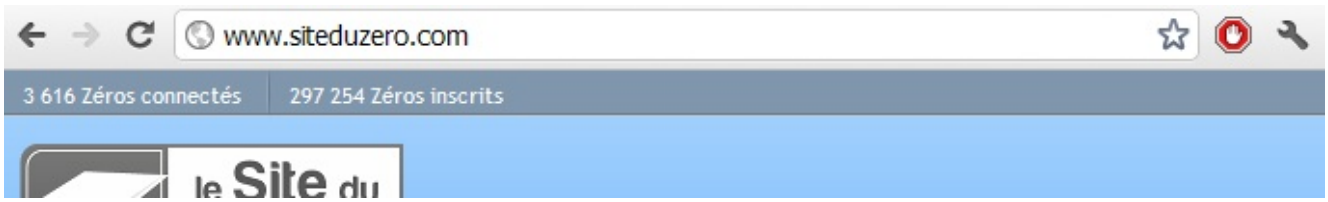
Connexion.

4. rendez-vous à nouveau sur la page http://localhost:8080/pro/restriction, et constatez au passage l'absence de redirection (l'URL ne change pas) :

Vous êtes connecté(e) avec l'adresse test@test.com, vous avez bien accès à l'espace restreint.

Acces restreint.

5. allez maintenant sur la page http://localhost:8080/pro/deconnexion :



Deconnexion.

6. retournez une dernière fois sur la page http://localhost:8080/pro/restriction.

Sans grande surprise, le système fonctionne bien : nous devons être connectés pour accéder à la page dont l'accès est restreint, sinon nous sommes redirigés vers la page publique.

Le problème

Oui, parce qu'il y a un léger problème ! Dans cet exemple, nous nous sommes occupés de deux pages : une page privée, une page publique. C'était rapide, simple et efficace. Maintenant si je vous demande d'étendre la restriction à 100 pages privées, comment comptez-vous vous y prendre ?

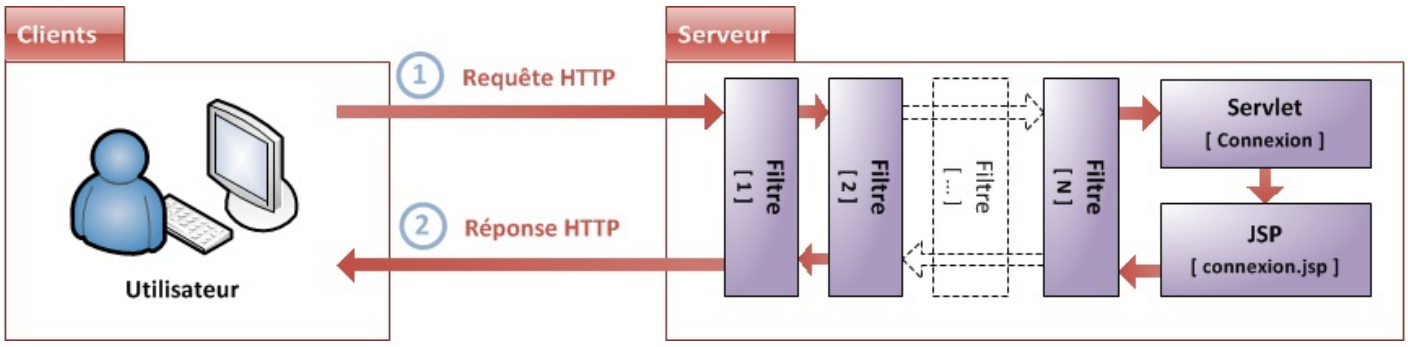
En l'état actuel de vos connaissances, vous n'avez pas d'autres moyens que de mettre en place un test sur le contenu de la session dans chacune des 100 servlets contrôlant l'accès aux 100 pages privées. Vous vous doutez bien que ce n'est absolument pas viable, et qu'il nous faut apprendre une autre méthode. Le remède à nos soucis s'appelle le **filtre**, et nous allons le découvrir dans le paragraphe suivant.

Le principe du filtre

Généralités

Qu'est-ce qu'un filtre ?

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



Vous pouvez d'ores et déjà remarquer sur cette illustration que les filtres peuvent intervenir à la fois sur la requête entrante et sur la réponse émise, et qu'ils s'appliquent dans un ordre précis et en cascade.

Quelle est la différence entre un filtre et une servlet ?

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

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

Quel est l'intérêt d'un filtre ?

Le filtres offre trois avantages majeurs, qui sont interdépendants :

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

C'est la combinaison de ces trois propriétés qui fait du filtre un composant parfaitement adapté à tous les traitements de masse, nécessitant d'être appliqués systématiquement à tout ou partie des pages d'une application. À titre d'exemple, on peut citer les usages suivants : l'authentification des visiteurs, la génération de logs, la conversion d'images, la compression de données ou encore le chiffrement de données.

Fonctionnement

Regardons maintenant comment est construit un filtre. À l'instar de sa cousine la servlet, qui doit obligatoirement implémenter l'interface `Servlet`, le filtre doit implémenter l'interface `Filter`. Mais cette fois, contrairement au cas de la servlet qui peut par exemple hériter de `HttpServlet`, il n'existe ici pas de classe fille. Lorsque nous étudions la documentation de l'interface, nous remarquons qu'elle est plutôt succincte, elle ne contient que trois définitions de méthodes : `init()`, `doFilter()` et `destroy()`.

Vous le savez, lorsqu'une classe Java implémente une interface, elle doit redéfinir chaque méthode présente dans cette interface. Ainsi, voici le code de la structure à vide d'un filtre :

Code : Java - Exemple d'un filtre

```
import java.io.IOException;

import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;

public class ExempleFilter implements Filter {
    public void init( FilterConfig config ) throws ServletException
{
        // ...
    }

    public void doFilter( ServletRequest request, ServletResponse
response, FilterChain chain ) throws IOException,
            ServletException {
        // ...
    }

    public void destroy() {
        // ...
    }
}
```

Les méthodes `init()` et `destroy()` concernent le cycle de vie du filtre dans l'application. Nous allons y revenir en aparté dans le paragraphe suivant. La méthode qui va contenir les traitements effectués par le filtre est donc `doFilter()`. Vous pouvez d'ailleurs le deviner en regardant les arguments qui lui sont transmis : elle reçoit en effet la requête et la réponse, ainsi qu'un troisième élément, **la chaîne des filtres**.

À quoi sert cette chaîne ?

Elle vous est encore inconnue, mais elle est en réalité un objet relativement simple : je vous laisse jeter un oeil à sa courte documentation. Je vous ai annoncé un peu plus tôt que plusieurs filtres pouvaient être appliqués à la même requête. Eh bien c'est à travers cette chaîne qu'un ordre va pouvoir être établi : chaque filtre qui doit être appliqué à la requête va être inclus à la chaîne, qui ressemble en fin de compte à une file d'invocations.

Cette chaîne est entièrement gérée par le conteneur, vous n'avez pas à vous en soucier. La seule chose que vous allez contrôler, c'est le passage d'un filtre à l'autre dans cette chaîne via l'appel de sa seule et unique méthode, elle aussi nommée `doFilter()`.

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

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

Code : XML - Exemple de déclaration de filtres

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
    ...

    <filter>
        <filter-name>Exemple</filter-name>
        <filter-class>package.ExempleFilter</filter-class>
    </filter>
    <filter>
        <filter-name>SecondExemple</filter-name>
        <filter-class>package.SecondExempleFilter</filter-class>
    </filter>

    <filter-mapping>
        <filter-name>Exemple</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
    <filter-mapping>
        <filter-name>SecondExemple</filter-name>
        <url-pattern>/page</url-pattern>
    </filter-mapping>

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

Vous reconnaissez ici la structure des blocs utilisés pour déclarer une servlet, la seule différence réside dans le nommage des champs : **`<servlet>`** devient **`<filter>`**, **`<servlet-name>`** devient **`<filter-name>`**, etc.

Eh bien là encore, de la même manière que pour les servlets, l'ordre des déclarations des mappings des filtres dans le fichier est important : c'est cet ordre qui va être suivi lors de l'invocation de plusieurs filtres appliqués à une même requête. En d'autres termes, c'est dans cet ordre que la chaîne des filtres va être automatiquement initialisée par le conteneur. Ainsi, si vous souhaitez qu'un filtre soit appliqué avant un autre, placez son mapping avant le mapping du second dans le fichier web.xml de votre application.

Cycle de vie

Avant de passer à l'application pratique et à la mise en place d'un filtre, penchons-nous un instant sur la manière dont le conteneur le gère. Une fois n'est pas coutume, il y a là encore de fortes similitudes avec une servlet. Lorsque l'application web démarre, le conteneur de servlets va créer une instance du filtre et la garder en mémoire durant toute l'existence de l'application. La même instance va être réutilisée pour chaque requête entrante dont l'URL correspond au contenu du champ **`<url-pattern>`** du mapping du filtre. Lors de l'instanciation, la méthode `init()` est appelée par le conteneur : si vous souhaitez passer des paramètres d'initialisation au filtre, vous pouvez alors les récupérer depuis l'objet `FilterConfig` passé en argument à la méthode.

Pour chacune de ces requêtes, la méthode `doFilter()` va être appelée. Ensuite c'est évidemment au développeur, à vous donc, de décider quoi faire dans cette méthode : une fois vos traitements appliqués, soit vous appelez la méthode `doFilter()` de l'objet `FilterChain` pour passer au filtre suivant dans la liste, soit vous effectuez une redirection ou un *forwarding* pour

changer la destination d'origine de la requête.

Enfin, je me répète mais il est possible de faire en sorte que plusieurs filtres s'appliquent à la même URL. Ils seront alors appelés dans le même ordre que celui de leurs déclarations de mapping dans le fichier web.xml de l'application.

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

Après cette longue introduction plutôt abstraite, lançons-nous et essayons d'utiliser un filtre pour répondre à notre problème : mettre en place une restriction d'accès sur un groupe de pages. C'est probablement l'utilisation la plus classique du filtre dans une application web !

Dans notre cas, nous allons nous en servir pour vérifier la présence d'un utilisateur dans la session :

- s'il est présent, notre filtre laissera la requête poursuivre son cheminement jusqu'à la page souhaitée ;
- s'il n'existe pas, notre filtre redirigera l'utilisateur vers la page publique.

Pour cela, nous allons commencer par créer un répertoire que nous allons placer à la racine de notre projet et nommer **restreint**, dans lequel nous allons déplacer le fichier **accesRestreint.jsp** et y placer les deux fichiers suivants :

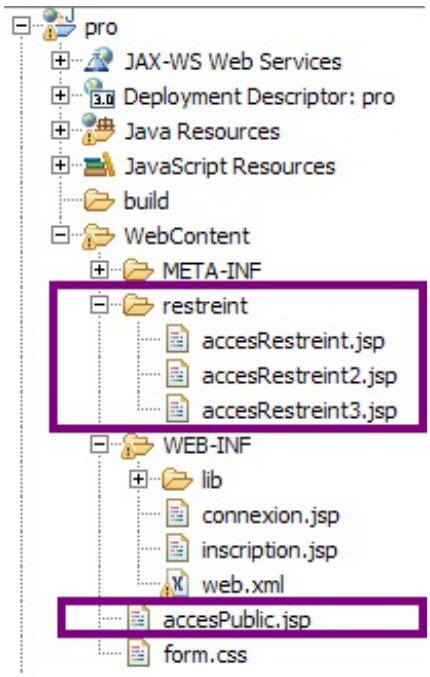
Code : JSP - /restreint/accesRestreint2.jsp

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Accès restreint 2</title>
    </head>
    <body>
        <p>Vous êtes connecté(e) avec l'adresse
${sessionScope.sessionUtilisateur.email}, vous avez bien accès à
l'espace restreint numéro 2.</p>
    </body>
</html>
```

Code : JSP - /restreint/accesRestreint3.jsp

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Accès restreint 3</title>
    </head>
    <body>
        <p>Vous êtes connecté(e) avec l'adresse
${sessionScope.sessionUtilisateur.email}, vous avez bien accès à
l'espace restreint numéro 3.</p>
    </body>
</html>
```

Voici un aperçu de l'arborescence que vous devez alors obtenir :

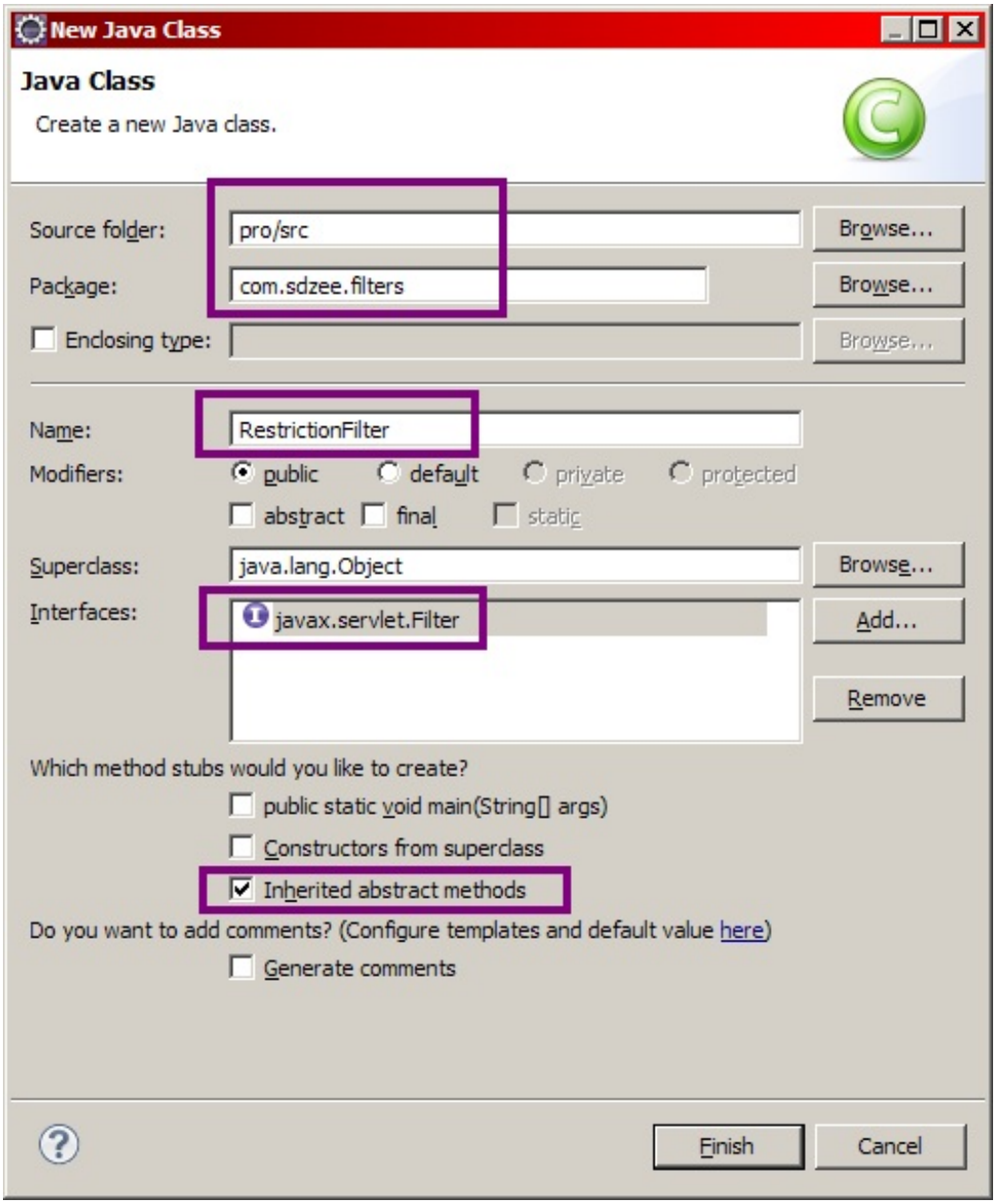


Arborescence sous Eclipse.

C'est de ce répertoire **restreint** que nous allons limiter l'accès aux utilisateurs connectés. Souvenez-vous bien du point suivant : pour le moment, nos pages JSP n'étant pas situées sous le répertoire **/WEB-INF**, elles sont accessibles au public directement depuis leur URL respective. Par exemple, vous pouvez vous rendre sur http://localhost:8080/pro/restreint/accesRestreint.jsp même sans être connecté, le seul problème que vous rencontrerez est l'absence de l'adresse email dans le message affiché.

Supprimez ensuite la servlet **Restriction** que nous avions développée en début de chapitre, ainsi que sa déclaration dans le fichier web.xml : elle nous est dorénavant inutile.

Nous pouvons maintenant créer notre filtre. Je vous propose de le placer dans un nouveau package `com.sdzee.filters`, et de le nommer **RestrictionFilter**. Voyez ci-dessous comment procéder après un Ctrl + N sous Eclipse :



Création d'un filtre.

Et remplacez alors le code généré automatiquement par Eclipse par le code suivant :

Code : Java - com.sdzee.filters.RestrictionFilter

```
package com.sdzee.filters;

import java.io.IOException;

import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;

public class RestrictionFilter implements Filter {
    public void init( FilterConfig config ) throws ServletException
{
    }

    public void doFilter( ServletRequest req, ServletResponse resp,
FilterChain chain ) throws IOException,
            ServletException {
    }
```

```
    public void destroy() {
    }
}
```

Rien de fondamental n'a changé par rapport à la version générée par Eclipse, j'ai simplement retiré les commentaires et renommé les arguments des méthodes pour que le code de notre filtre soit plus lisible par la suite.

Comme vous le savez, c'est dans la méthode `doFilter()` que nous allons réaliser notre vérification. Puisque nous avons déjà développé cette fonctionnalité dans une servlet en début de chapitre, il nous suffit de reprendre son code et de l'adapter un peu :

Code : Java - com.sdzee.filters.RestrictionFilter

```
package com.sdzee.filters;

import java.io.IOException;

import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

public class RestrictionFilter implements Filter {
    public static final String ACCES_PUBLIC     =
"/accesPublic.jsp";
    public static final String ATT_SESSION_USER =
"sessionUtilisateur";

    public void init( FilterConfig config ) throws ServletException
{
    }

    public void doFilter( ServletRequest req, ServletResponse res,
FilterChain chain ) throws IOException,
            ServletException {
        /* Cast des objets request et response */
HttpServletRequest request = (HttpServletRequest) req;
HttpServletResponse response = (HttpServletResponse) res;

        /* Récupération de la session depuis la requête */
        HttpSession session = request.getSession();

        /**
* Si l'objet utilisateur n'existe pas dans la session en cours,
alors
* l'utilisateur n'est pas connecté.
*/
        if ( session.getAttribute( ATT_SESSION_USER ) == null ) {
            /* Redirection vers la page publique */
            response.sendRedirect( request.getContextPath() +
ACCES_PUBLIC );
        } else {
            /* Affichage de la page restreinte */
chain.doFilter( request, response );
        }
    }

    public void destroy() {
    }
}
```

Quelques explications s'imposent :

- aux lignes 25 et 26, vous constatez que nous convertissons les objets transmis en argument à notre méthode `doFilter()`. La raison est simple : comme je vous l'ai déjà dit, il n'existe pas de classe fille implémentant l'interface `Filter`, alors que côté servlet nous avons bien `HttpServlet` qui implémente `Servlet`. Ce qui signifie que notre filtre n'est pas spécialisé, il implémente uniquement `Filter` et peut traiter n'importe quel type de requête et pas seulement les requêtes HTTP. C'est donc pour cela que nous devons manuellement spécialiser nos objets, en effectuant un *cast* vers les objets dédiés aux requêtes et réponses HTTP : c'est seulement en procédant à cette conversion que nous aurons accès ensuite à la session, qui est propre à l'objet `HttpServletRequest`, et n'existe pas dans l'objet `ServletRequest`.

- à la ligne 40, nous avons remplacé le *forwarding* auparavant en place dans notre servlet par un appel à la méthode `doFilter()` de l'objet `FilterChain`. Celle-ci a en effet une particularité intéressante : si un autre filtre existe après le filtre courant dans la chaîne, alors c'est vers ce filtre que la requête va être transmise. Par contre, si aucun autre filtre n'est présent ou si le filtre courant est le dernier de la chaîne, alors c'est vers la ressource initialement demandée que la requête va être acheminée. En l'occurrence, nous n'avons qu'un seul filtre en place, notre requête sera donc logiquement transmise à la page demandée.

Pour mettre en scène notre filtre, il nous faut enfin le déclarer dans le fichier web.xml de notre application :

Code : XML - /WEB-INF/web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
<filter>
<filter-name>RestrictionFilter</filter-name>
<filter-class>com.sdzee.filters.RestrictionFilter</filter-class>
</filter>
<filter-mapping>
<filter-name>RestrictionFilter</filter-name>
<url-pattern>/restreint/*</url-pattern>
</filter-mapping>

 <servlet>
  <servlet-name>Inscription</servlet-name>
  <servlet-class>com.sdzee.servlets.Inscription</servlet-class>
 </servlet>
 <servlet>
  <servlet-name>Connexion</servlet-name>
  <servlet-class>com.sdzee.servlets.Connexion</servlet-class>
 </servlet>
 <servlet>
  <servlet-name>Deconnexion</servlet-name>
  <servlet-class>com.sdzee.servlets.Deconnexion</servlet-class>
 </servlet>

 <servlet-mapping>
  <servlet-name>Inscription</servlet-name>
  <url-pattern>/inscription</url-pattern>
 </servlet-mapping>
 <servlet-mapping>
  <servlet-name>Connexion</servlet-name>
  <url-pattern>/connexion</url-pattern>
 </servlet-mapping>
 <servlet-mapping>
  <servlet-name>Deconnexion</servlet-name>
  <url-pattern>/deconnexion</url-pattern>
 </servlet-mapping>
</web-app>
```

À la ligne 9, vous pouvez remarquer l'url-pattern précisé : le caractère * signifie que notre filtre va être appliqué à toutes les pages présentes sous le répertoire **/restreint**.

Redémarrez ensuite Tomcat pour que les modifications effectuées soient prises en compte, puis suivez ce scénario de tests :

1. essayez d'accéder à la page http://localhost:8080/pro/restreint/accesRestreint.jsp, et constatez la redirection vers la page

publique :

Vous n'avez pas accès à l'espace restreint : vous devez vous connecter d'abord.

Acces restreint.

2. rendez-vous sur la page de connexion et connectez-vous avec des informations valides :

Connexion.

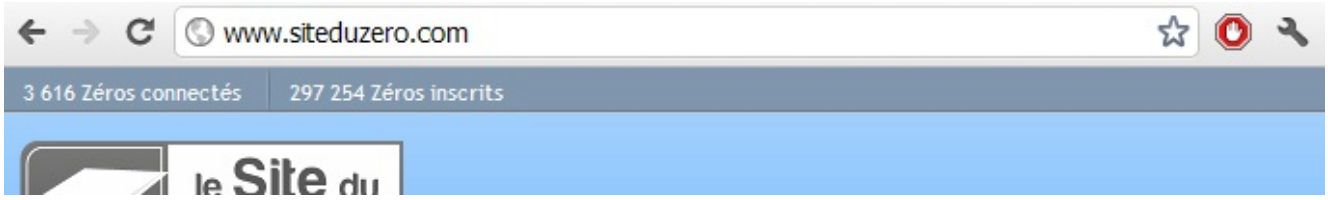
3. essayez à nouveau d'accéder à la page http://localhost:8080/pro/restreint/accesRestreint.jsp, et constatez le succès de l'opération :

Vous êtes connecté(e) avec l'adresse test@test.com, vous avez bien accès à l'espace restreint.

Acces restreint.

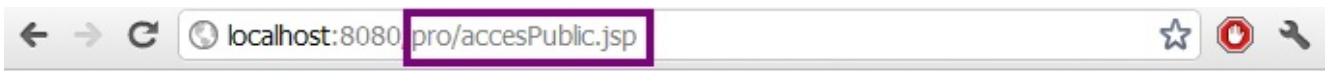
4. essayez alors d'accéder aux pages **accesRestreint2.jsp** et **accesRestreint3.jsp**, et constatez là encore le succès de l'opération ;

5. rendez-vous sur la page de déconnexion :



Deconnexion.

6. puis tentez alors d'accéder à la page http://localhost:8080/pro/restreint/accesRestreint.jsp et constatez cette fois l'échec de l'opération :



Vous n'avez pas accès à l'espace restreint : vous devez vous connecter d'abord.

Acces restreint.

Notre problème est bel et bien réglé ! Nous sommes maintenant capables de bloquer l'accès à un ensemble de pages avec une simple vérification dans un unique filtre : nous n'avons pas besoin de dupliquer le contrôle effectué dans des servlets appliquées à chacune des pages !

Restreindre l'application entière

Avant de nous quitter, regardons brièvement comment forcer l'utilisateur à se connecter pour accéder à notre application. Ce principe est par exemple souvent utilisé sur les intranets d'entreprise, où la connexion est généralement obligatoire dès l'entrée sur le site.

La première chose à faire, c'est de modifier la portée d'application du filtre. Puisque nous souhaitons couvrir l'intégralité des requêtes entrantes, il suffit d'utiliser le caractère * appliqué à la racine. La déclaration de notre filtre devient donc :

Code : XML - /WEB-INF/web.xml

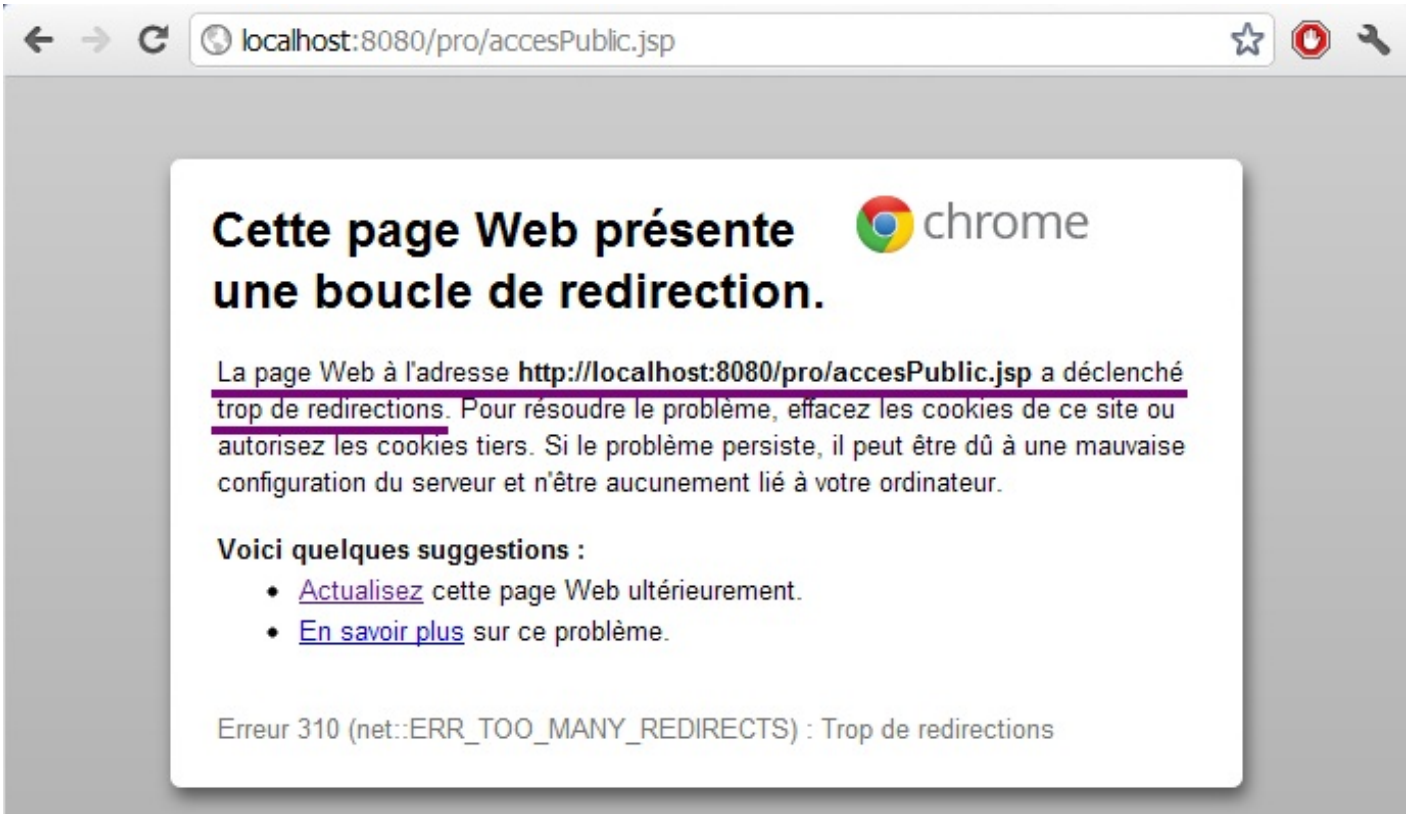
```
<filter>
        <filter-name>RestrictionFilter</filter-name>
        <filter-class>com.sdzee.filters.RestrictionFilter</filter-
class>
</filter>
<filter-mapping>
        <filter-name>RestrictionFilter</filter-name>
<url-pattern>/*</url-pattern>
</filter-mapping>
```

Redémarrez Tomcat pour que la modification soit prise en compte.

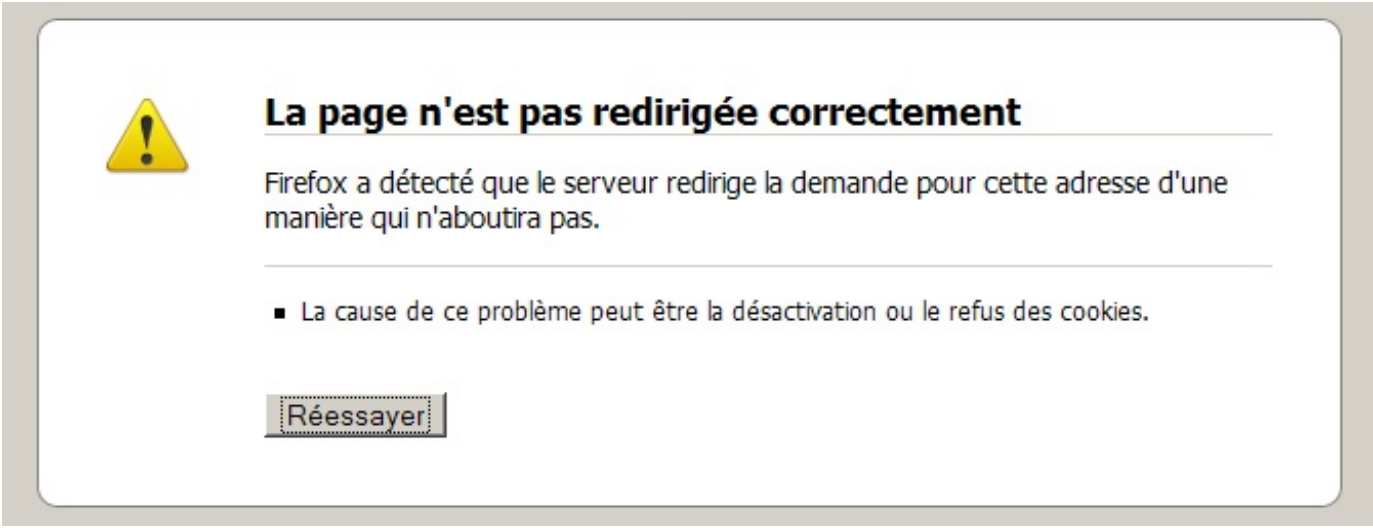
Maintenant, vous devez réfléchir à ce que nous venons de mettre en place : nous avons ordonné à notre filtre de bloquer toutes les requêtes entrantes si l'utilisateur n'est pas connecté. Le problème, c'est que si nous ne changeons pas le code de notre filtre, alors l'utilisateur ne pourra jamais accéder à notre site !

Pourquoi ? Notre filtre le redirigera vers la page **accesPublic.jsp** comme il le faisait dans le cas de la restriction d'accès au répertoire **restreint**, non ?

Eh bien non, plus maintenant ! La méthode de redirection que nous avons mise en place va bien être appelée, mais comme vous le savez elle va déclencher un échange HTTP, un aller-retour avec le navigateur du client. Et donc le client va renvoyer automatiquement une requête, qui va à son tour être interceptée par notre filtre. Le client n'étant toujours pas connecté, le même phénomène va se reproduire, etc. Si vous y tenez, vous pouvez essayer : vous verrez alors votre navigateur vous avertir que la page que vous essayez de contacter pose problème. Voici les messages affichés respectivement par Chrome et Firefox :



Echec de la restriction.



Echec de la restriction.

La solution est simple :

1. il faut envoyer l'utilisateur vers la page de connexion, et non plus vers la page **accesPublic.jsp** ;
2. il faut effectuer non plus une redirection HTTP mais un *forwarding*, afin qu'aucun nouvel échange HTTP n'ait lieu et que la demande aboutisse.

Voici ce que devient donc le code de notre filtre :

Code : Java - com.sdzee.filters.RestrictionFilter

```
package com.sdzee.filters;

import java.io.IOException;

import javax.servlet.Filter;
import javax.servlet.FilterChain;
```

```
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

public class RestrictionFilter implements Filter {
public static final String ACCES_CONNEXION = "/connexion";
    public static final String ATT_SESSION_USER =
"sessionUtilisateur";

    public void init( FilterConfig config ) throws ServletException
{
    }

    public void doFilter( ServletRequest req, ServletResponse res,
FilterChain chain ) throws IOException,
            ServletException {
        /* Cast des objets request et response */
        HttpServletRequest request = (HttpServletRequest) req;
        HttpServletResponse response = (HttpServletResponse) res;

        /* Récupération de la session depuis la requête */
        HttpSession session = request.getSession();

        /**
* Si l'objet utilisateur n'existe pas dans la session en cours,
alors
* l'utilisateur n'est pas connecté.
*/
        if ( session.getAttribute( ATT_SESSION_USER ) == null ) {
            /* Redirection vers la page publique */
request.getRequestDispatcher( ACCES_CONNEXION ).forward( request,
response );
        } else {
            /* Affichage de la page restreinte */
            chain.doFilter( request, response );
        }
    }

    public void destroy() {
    }
}
```

C'est tout pour le moment. Testez alors d'accéder à la page http://localhost:8080/pro/restreint/accesRestreint.jsp, vous obtiendrez :

Ratage du CSS.

Vous pouvez alors constater que notre solution fonctionne, l'utilisateur est maintenant bien redirigé vers la page de connexion. Oui, mais...

Où est passé le design de notre page ?!

Eh bien la réponse est simple : il a été bloqué ! En réalité lorsque vous accédez à une page web sur laquelle est attachée une feuille de style CSS, votre navigateur va dans les coulisses envoyer une deuxième requête au serveur pour récupérer silencieusement cette feuille et ensuite appliquer les styles au contenu HTML. Et vous pouvez le deviner, cette seconde requête a bien évidemment été bloquée par notre super-filtre !

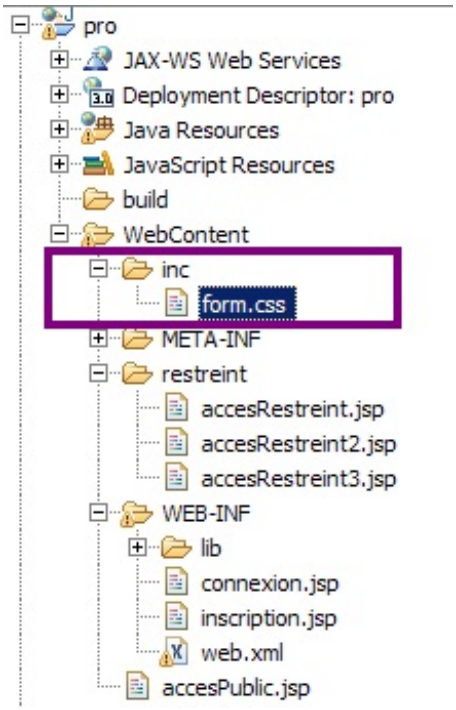
Comment régler ce problème ?

Il y a plusieurs solutions envisageables. Voici les deux plus courantes :

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

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

Pour commencer, créez donc un répertoire nommé **inc** sous la racine de votre application et placez-y notre fichier CSS :



Inc.

Puisque nous venons de déplacer le fichier, nous devons également modifier son appel dans la page de connexion :

Code : JSP - /WEB-INF/connexion.jsp

```
<!-- Dans le fichier connexion.jsp, remplacez l'appel suivant : -->
<link type="text/css" rel="stylesheet" href="form.css" />

<!-- Par celui-ci : -->
<link type="text/css" rel="stylesheet" href="inc/form.css" />
```

Pour terminer, nous devons réaliser dans la méthode `doFilter()` de notre filtre ce fameux passe-droit sur le dossier **inc** :

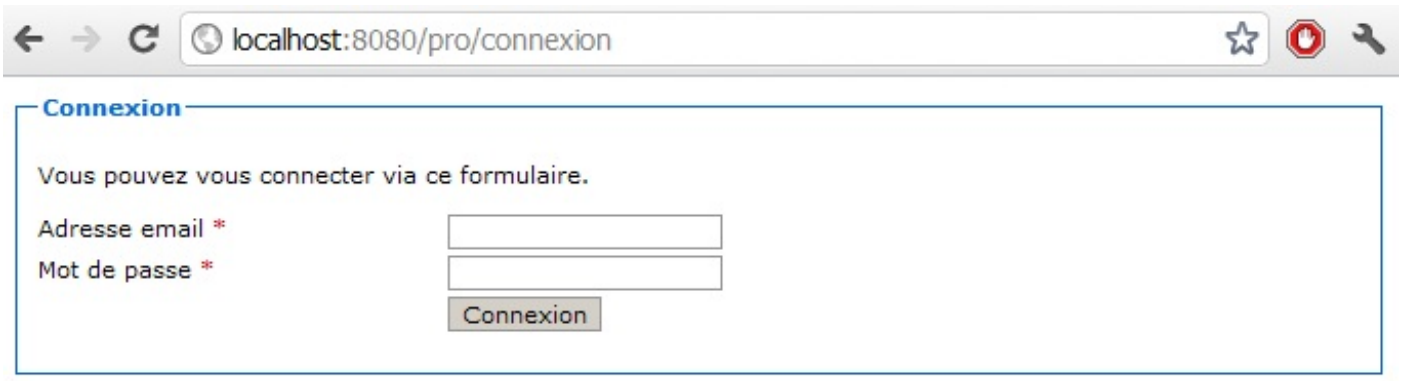
Code : Java - com.sdzee.filters.RestrictionFilter

```
package com.sdzee.filters;

import java.io.IOException;

import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

public class RestrictionFilter implements Filter {
    public static final String ACCES_CONNEXION  = "/connexion";
    public static final String ATT_SESSION_USER =
"sessionUtilisateur";

    public void init( FilterConfig config ) throws ServletException
{
    }

    public void doFilter( ServletRequest req, ServletResponse res,
FilterChain chain ) throws IOException,
            ServletException {
        /* Cast des objets request et response */
        HttpServletRequest request = (HttpServletRequest) req;
        HttpServletResponse response = (HttpServletResponse) res;

        /* Non-filtrage des ressources statiques */
String chemin = request.getRequestURI().substring(
request.getContextPath().length() );
if ( chemin.startsWith( "/inc" ) ) {
chain.doFilter( request, response );
return;
}

        /* Récupération de la session depuis la requête */
        HttpSession session = request.getSession();

        /**
* Si l'objet utilisateur n'existe pas dans la session en cours,
alors
* l'utilisateur n'est pas connecté.
*/
        if ( session.getAttribute( ATT_SESSION_USER ) == null ) {
            /* Redirection vers la page publique */
            request.getRequestDispatcher( ACCES_CONNEXION ).forward(
request, response );
        } else {
            /* Affichage de la page restreinte */
            chain.doFilter( request, response );
        }
    }

    public void destroy() {
    }
}
```

Explications :

- à la ligne 29, nous récupérons l'URL d'appel de la requête HTTP via la méthode `getRequestURI()`, puis nous plaçons dans la chaîne **chemin** sa partie finale, c'est-à-dire la partie située après le contexte de l'application. Typiquement, dans notre cas si nous nous rendons sur http://localhost:8080/pro/restreint/accesRestreint.jsp, la méthode `getRequestURI()` va renvoyer /pro/restreint/accesRestreint.jsp et **chemin** va contenir uniquement **/restreint/accesRestreint.jsp**.
- à la ligne 30, nous testons si cette chaîne **chemin** commence par **/inc** : si c'est le cas, cela signifie que la page demandée est une des ressources statiques que nous avons placées sous le répertoire **inc**, et qu'il ne faut donc pas lui appliquer le filtre !
- à la ligne 31, nous laissons donc la requête poursuivre son cheminement en appelant la méthode `doFilter()` de la chaîne.

Faites les modifications, enregistrez et tentez d'accéder à la page http://localhost:8080/pro/connexion :



CSS OK.

Le résultat est parfait, cela fonctionne ! Oui, mais...

Quoi encore ? Il n'y a plus de problème, toute l'application fonctionne maintenant !

Toute ? Non ! Un irréductible souci résiste encore et toujours... Par exemple, rendez-vous maintenant sur la page http://localhost:8080/pro/restreint/accesRestreint.jsp :

Ratage du CSS.

Pourquoi la feuille de style n'est-elle pas appliquée à notre formulaire de connexion dans ce cas ?

Eh bien cette fois, c'est à cause du *forwarding* que nous avons mis en place dans notre filtre ! Eh oui, souvenez-vous : le *forwarding* ne modifie pas l'URL côté client, comme vous pouvez d'ailleurs le voir sur cette dernière illustration. Cela veut dire que le navigateur du client reçoit bien le formulaire de connexion, mais ne sait pas que c'est la page **/connexion.jsp** qui le lui a renvoyé, il croit qu'il s'agit tout bonnement du retour de la page demandée, c'est-à-dire **/restreint/accesRestreint.jsp**.

De ce fait, lorsqu'il va silencieusement envoyer une requête au serveur pour récupérer la feuille CSS associée à la page de conexion, le navigateur va naïvement se baser sur l'URL qu'il a en mémoire pour interpréter l'appel suivant :

Code : JSP - Extrait de connexion.jsp

```
<link type="text/css" rel="stylesheet" href="inc/form.css" />
```

En conséquence, il va considérer que l'URL relative "inc/form.css" se rapporte au répertoire qu'il pense être le répertoire courant, à savoir **/restreint** (puisque pour lui, le formulaire a été affiché par **/restreint/accesRestreint.jsp**). Ainsi, le navigateur va demander au serveur de lui renvoyer la page **/restreint/inc/forms.css**, alors que cette page n'existe pas ! Voilà pourquoi le design de notre formulaire semble avoir disparu.

Pour régler ce problème, nous n'allons ni toucher au filtre ni au *forwarding*, mais nous allons tirer parti de la JSTL pour modifier la page **connexion.jsp** :

Code : JSP - /WEB-INF/connexion.jsp

```
<!-- Dans le fichier connexion.jsp, remplacez l'appel suivant : -->
<link type="text/css" rel="stylesheet" href="inc/form.css" />

<!-- Par celui-ci : -->
<link type="text/css" rel="stylesheet" href="<c:url
value="/inc/form.css"/>" />
```

Vous vous rappelez de ce que je vous avais expliqué à propos de la balise **<c:url/>** ? Je vous avais dit qu'elle ajoutait automatiquement aux URL absolues qu'elle contenait le contexte de l'application. C'est exactement ce que nous souhaitons : dans ce cas, le rendu de la balise sera **/pro/inc/form.css**. Le navigateur reconnaîtra donc ici une URL absolue et non plus une URL relative comme c'était le cas auparavant, et il réalisera correctement l'appel au fichier CSS !

Dans ce cas, pourquoi ne pas avoir directement écrit l'URL absolue "/pro/inc/form.css" dans l'appel ? Pourquoi s'embêter avec **<c:url/>** ?

Pour la même raison que nous avions utilisé `request.getContextPath()` dans la servlet que nous avions développée en première partie de ce chapitre : si demain nous décidons de changer le nom du contexte, notre page fonctionnera toujours avec la balise **<c:url/>**, alors qu'il faudra éditer et modifier l'URL absolue entrée à la main sinon. J'espère que cette fois, vous avez bien compris !

Une fois la modification effectuée, voici le résultat :

CSS OK.

Finalement, nous y sommes : tout fonctionne comme prévu !

Désactiver le filtre

Une fois vos développements et tests terminés, pour plus de commodité dans les exemples à suivre vous pouvez désactiver ce filtre, en commentant simplement sa déclaration dans le fichier **web.xml** de votre application :

Code : XML - /WEB-INF/web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
    ...

<!--
<filter>
<filter-name>RestrictionFilter</filter-name>
<filter-class>com.sdzee.filters.RestrictionFilter</filter-class>
</filter>
<filter-mapping>
<filter-name>RestrictionFilter</filter-name>
<url-pattern>/*</url-pattern>
</filter-mapping>
-->

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

Il faudra redémarrer Tomcat pour que la modification soit prise en compte.
Dans cet imposant chapitre, qu'il s'agisse de rappels ou de nouveaux concepts, je vous ai communiqué énormément d'informations utiles. Prenez bien le temps d'effectuer chacune des modifications au travers desquelles je vous guide, d'analyser calmement les différentes erreurs rencontrées et d'assimiler et retenir les conseils de conception que je vous enseigne.

Une fois que vous avez bien digéré, filez au chapitre suivant et attaquez le dessert : des cookies vous attendent ! Miam !

Le cookie : le navigateur vous ouvre ses portes

Nous avons déjà bien entamé la découverte du cookie malgré nous, lorsque nous avons étudié le fonctionnement des sessions, mais nous allons tout de même prendre la peine de rappeler les concepts qui se cachent sous cette technologie. Nous allons ensuite mettre en pratique la théorie dans un exemple concret, et terminer sur une discussion autour de leur sécurité.

Le principe du cookie

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

Côté HTTP

Pour commencer, le cookie est une notion qui est liée au protocole HTTP, et qui est définie par la RFC 6265 depuis avril 2011. Cette nouvelle version de la norme rend caduque la version 2965, qui elle-même remplaçait la version 2109. Si vous avez du temps à perdre, vous pouvez chercher les modifications apportées au fil des évolutions, c'est un excellent exercice d'entrainement d'analyse de RFC, ces documents massifs qui font office de référence absolue dans bon nombre de domaines !

Ça, c'était pour la théorie. Dans la pratique, dans le chapitre portant sur les sessions nous avons déjà analysé des échanges HTTP impliquant des transferts de cookies de serveur vers navigateur et inversement, et avons découvert que :

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

C'est tout ce qu'il se passe dans les coulisses du protocole HTTP, il s'agit uniquement d'un paramètre dans la requête ou dans la réponse.

Côté Java EE

La plate-forme Java EE permet de manipuler un cookie à travers l'objet Java Cookie. Sa documentation claire et concise nous informe notamment que :

- un cookie doit obligatoirement avoir un nom et une valeur ;
- il est possible d'attribuer des options à un cookie, telles qu'une date d'expiration ou un numéro de version. Toutefois, elle nous précise ici que certains navigateurs présentent des bugs dans leur gestion de ces options, et qu'il est préférable d'en limiter l'usage autant que faire se peut afin de rendre notre application aussi multiplateforme que possible ;
- la méthode `addCookie()` de l'objet `HttpServletResponse` est utilisée pour ajouter un cookie à la réponse qui sera envoyée client ;
- la méthode `getCookies()` de l'objet `HttpServletRequest` est utilisée pour récupérer la liste des cookies envoyés par le client ;
- par défaut, les objets ainsi créés respectent la toute première norme décrivant les cookies HTTP, une norme encore plus ancienne que la 2109 dont je vous ai parlé dans le paragraphe précédent, afin d'assurer la meilleure interopérabilité possible. La documentation de la méthode `setVersion()` nous précise même que la version 2109 est considérée comme "récente et expérimentale". Bref, la documentation commence sérieusement à dater... Peu importe, tout ce dont nous avons besoin pour le moment était déjà décrit dans le tout premier document, pas de soucis à se faire !

Voilà tout ce qu'il vous est nécessaire de savoir pour attaquer. Bien évidemment, n'hésitez pas à parcourir plus en profondeur la Javadoc de l'objet `Cookie` pour en connaître davantage !

Avant de passer à la pratique, comprenez bien que **cookies et sessions sont deux concepts totalement distincts** ! Même s'il est vrai que l'établissement d'une session en Java EE peut s'appuyer sur un cookie, il ne faut pas confondre les deux notions : la session est un espace mémoire alloué sur le serveur dans lequel vous pouvez placer n'importe quel type d'objets, alors que le cookie est un espace mémoire alloué dans le navigateur du client dans lequel vous ne pouvez placer que du texte.

Souvenez-vous de vos clients !

Pour illustrer la mise en place d'un cookie chez l'utilisateur, nous allons donner à notre formulaire de connexion... une mémoire ! Plus précisément, nous allons donner le choix à l'utilisateur d'enregistrer ou non la date de sa dernière connexion, via une case à

cocher dans notre formulaire. S'il fait ce choix, alors nous allons stocker la date et l'heure de sa connexion dans un cookie et le placer dans son navigateur. Ainsi, à son retour après déconnexion, nous serons en mesure de lui afficher depuis combien de temps il ne s'est pas connecté.

Ce système ne fera donc intervenir qu'un seul cookie, chargé de sauvegarder la date de connexion.

Alors bien évidemment, c'est une simple fonctionnalité que je vous fais mettre en place à titre d'application pratique uniquement : elle est aisément faillible, par exemple si l'utilisateur supprime les cookies de son navigateur, les bloque, ou encore s'il se connecte depuis un autre poste ou un autre navigateur. Mais peu importe, le principal est que vous travailliez la manipulation de cookies, et au passage cela vous donnera une occasion :

- de travailler à nouveau la manipulation des dates avec la bibliothèque JodaTime ;
- de découvrir comment traiter une case à cocher, c'est-à-dire un champ de formulaire HTML de type `<input type="checkbox"/>`.

D'une pierre... trois coups !

Reprise de la servlet

Le plus gros de notre travail va se concentrer sur la servlet de connexion. C'est ici que nous allons devoir manipuler notre unique cookie, et effectuer différentes vérifications. En reprenant calmement notre système, nous pouvons identifier les deux besoins suivants :

1. à l'affichage du formulaire de connexion par un visiteur, il nous faut vérifier si le cookie enregistrant la date de la précédente connexion a été envoyé dans la requête HTTP par le navigateur du client. Si oui, alors cela signifie que le visiteur s'est déjà connecté par le passé avec ce navigateur, et que nous pouvons donc lui afficher depuis combien de temps il ne s'est pas connecté. Si non, alors il ne s'est jamais connecté et nous lui affichons simplement le formulaire ;
2. à la connexion d'un visiteur, il nous faut vérifier s'il a coché la case dans le formulaire, et si oui il nous faut récupérer la date courante, l'enregistrer dans un cookie et l'envoyer au navigateur du client à travers la réponse HTTP.

À l'affichage du formulaire

Lors de la réception d'une demande d'accès à la page de connexion, la méthode `doGet()` de notre servlet va devoir :

- vérifier si un cookie a été envoyé par le navigateur dans les en-têtes de la requête ;
- si oui, alors elle doit calculer la différence entre la date courante et la date présente dans le cookie, et la transmettre à la JSP pour affichage.

Code : Java - com.sdzee.servlets.Connexion

```
package com.sdzee.servlets;

import java.io.IOException;

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

import org.joda.time.DateTime;
import org.joda.time.Period;
import org.joda.time.format.DateTimeFormat;
import org.joda.time.format.DateTimeFormatter;
import org.joda.time.format.PeriodFormatter;
import org.joda.time.format.PeriodFormatterBuilder;

import com.sdzee.beans.Utilisateur;
import com.sdzee.forms.ConnexionForm;
```

```
public class Connexion extends HttpServlet {
    public static final String  ATT_USER                  =
"utilisateur";
    public static final String  ATT_FORM                  = "form";
public static final String ATT_INTERVALLE_CONNEXIONS =
"intervalleConnexions";
    public static final String  ATT_SESSION_USER          =
"sessionUtilisateur";

public static final String COOKIE_DERNIERE_CONNEXION =
"derniereConnexion";

public static final String FORMAT_DATE = "dd/MM/yyyy HH:mm:ss";

    public static final String  VUE                       = "/WEB-
INF/connexion.jsp";

    public void doGet( HttpServletRequest request,
HttpServletResponse response ) throws ServletException, IOException
{
        /* Tentative de récupération du cookie depuis la requête */
String dernieреConnexion = getCookieValue( request,
COOKIE_DERNIERE_CONNEXION );
        /* Si le cookie existe, alors calcul de la durée */
if ( dernieреConnexion != null ) {
/* Récupération de la date courante */
DateTime dtCourante = new DateTime();
/* Récupération de la date présente dans le cookie */
DateTimeFormatter formatter = DateTimeFormat.forPattern( FORMAT_DATE
);
DateTime dtDernieреConnexion = formatter.parseDateTime(
dernieреConnexion );
/* Calcul de la durée de l'intervalle */
Period periode = new Period( dtDernieреConnexion, dtCourante );
/* Formatage de la durée de l'intervalle */
PeriodFormatter periodFormatter = new PeriodFormatterBuilder()
.appendYears().appendSuffix( " an ", " ans " )
.appendMonths().appendSuffix( " mois " )
.appendDays().appendSuffix( " jour ", " jours " )
.appendHours().appendSuffix( " heure ", " heures " )
.appendMinutes().appendSuffix( " minute ", " minutes " )
.appendSeparator( "et " )
.appendSeconds().appendSuffix( " seconde", " secondes" )
.toFormatter();
String intervalleConnexions = periodFormatter.print( periode );
/* Ajout de l'intervalle en tant qu'attribut de la requête */
request.setAttribute( ATT_INTERVALLE_CONNEXIONS,
intervalleConnexions );
}

        /* Affichage de la page de connexion */
        this.getServletContext().getRequestDispatcher( VUE
).forward( request, response );
    }

    public void doPost( HttpServletRequest request,
HttpServletResponse response ) throws ServletException, IOException
{
        ...
    }

    /**
* Méthode utilitaire gérant la récupération de la valeur d'un
cookie donné
* depuis la requête HTTP.
*/
private static String getCookieValue( HttpServletRequest request,
String nom ) {
Cookie[] cookies = request.getCookies();
```

```
        if ( cookies != null ) {
        for ( Cookie cookie : cookies ) {
        if ( cookie != null && nom.equals( cookie.getName() ) ) {
        return cookie.getValue();
        }
        }
        }
        return null;
        }
        }
```

De plus amples explications :

- j'ai choisi de nommer le cookie stocké chez le client **dernièreConnexion** ;
- j'ai mis en place une méthode nommée `getCookieValue()`, dédiée à la recherche d'un cookie donné dans une requête HTTP :
  - à la ligne 74, elle récupère tous les cookies présents dans la requête grâce à la méthode `request.getCookies()`, que je vous ai présentée un peu plus tôt ;
  - à la ligne 75, elle vérifie si des cookies existent, c'est-à-dire si `request.getCookies()` n'a pas retourné `null` ;
  - à la ligne 76, elle parcourt le tableau de cookies récupéré ;
  - à la ligne 77, elle vérifie si un des éventuels cookies présents dans le tableau a pour nom, récupéré par un appel à `cookie.getName()`, le paramètre **nom** passé en argument ;
  - à la ligne 78, si un tel cookie est trouvé, elle retourne sa valeur via un appel à `cookie.getValue()`.
- à la ligne 38, je teste si ma méthode `getCookieValue()` a retourné une valeur ou non ;
- de la ligne 39 à la ligne 58, je traite les dates grâce aux méthodes de la bibliothèque JodaTime. Je vous recommande fortement d'aller vous-même parcourir son guide d'utilisation ainsi que sa FAQ. C'est en anglais, mais les codes d'exemples sont très explicites. Voici quelques détails en supplément des commentaires déjà présents dans le code de la servlet :
  - j'ai pris pour convention le format `"dd/MM/yyyy HH:mm:ss"`, et considère donc que la date sera stockée sous ce format dans le cookie **dernièreConnexion** placé dans le navigateur le client ;
  - les lignes 42 et 43 permettent de traduire la date présente au format texte dans le cookie du client, en un objet `DateTime` que nous utiliserons par la suite pour effectuer la différence avec la date courante ;
  - à la ligne 45, je calcule la différence entre la date courante et la date de la dernière visite, c'est-à-dire l'intervalle de temps écoulé ;
  - de la ligne 47 à 55, je crée un format d'affichage de mon choix à l'aide de l'objet `PeriodFormatterBuilder` ;
  - à la ligne 56 j'enregistre dans un `String`, via la méthode `print()`, l'intervalle mis en forme avec le format que j'ai fraîchement défini ;
  - enfin à la ligne 58, je transmet l'intervalle mis en forme à notre JSP, via un simple attribut de requête nommé **intervalleConnexions**.

En fin de compte, si vous mettez de côté la tambouille que nous réalisons pour manipuler nos dates et calculer l'intervalle entre deux connexions, vous vous rendrez compte que le traitement lié au cookie en lui-même est assez court : il suffit simplement de vérifier le retour de la méthode `request.getCookies()`, chose que nous faisons ici grâce à notre méthode `getCookieValue()`.

À la connexion du visiteur

La seconde étape, c'est maintenant de gérer la connexion d'un visiteur. Il va falloir :

- vérifier si la case est cochée ou non ;
- si oui, alors l'utilisateur souhaite qu'on se souvienne de lui et il nous faut :
  - récupérer la date courante ;
  - la convertir au format texte choisi ;
  - l'enregistrer dans un cookie nommé **dernièreConnexion** et l'envoyer au navigateur du client à travers la réponse HTTP.
- si non, alors l'utilisateur ne souhaite pas qu'on se souvienne de lui et il nous faut :
  - demander la suppression du cookie nommé **dernièreConnexion** qui existe éventuellement déjà dans le navigateur du client.

Code : Java - com.sdzee.servlets.Connexion

```
package com.sdzee.servlets;

import java.io.IOException;

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

import org.joda.time.DateTime;
import org.joda.time.Period;
import org.joda.time.format.DateTimeFormat;
import org.joda.time.format.DateTimeFormatter;
import org.joda.time.format.PeriodFormatter;
import org.joda.time.format.PeriodFormatterBuilder;

import com.sdzee.beans.Utilisateur;
import com.sdzee.forms.ConnexionForm;

public class Connexion extends HttpServlet {
public static final String CHAMP_MEMOIRE = "memoire";

    public static final String  ATT_USER                  =
"utilisateur";
    public static final String  ATT_FORM                  = "form";
    public static final String  ATT_INTERVALLE_CONNEXIONS =
"intervalleConnexions";
    public static final String  ATT_SESSION_USER          =
"sessionUtilisateur";

    public static final String  COOKIE_DERNIERE_CONNEXION =
"derniereConnexion";
public static final int COOKIE_MAX_AGE = 60 * 60 * 24 * 365; // 1an

    public static final String  FORMAT_DATE               =
"dd/MM/yyyy HH:mm:ss";

    public static final String  VUE                       = "/WEB-
INF/connexion.jsp";

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

    public void doPost( HttpServletRequest request,
HttpServletResponse response ) throws ServletException, IOException
{
        /* Préparation de l'objet formulaire */
        ConnexionForm form = new ConnexionForm();

        /* Traitement de la requête et récupération du bean en
résultant */
        Utilisateur utilisateur = form.connecterUtilisateur( request
);

        /* Récupération de la session depuis la requête */
        HttpSession session = request.getSession();

        /*
* Si aucune erreur de validation n'a eu lieu, alors ajout du bean
* Utilisateur à la session, sinon suppression du bean de la
session.
```

```
        */
                if ( form.getErreurs.isEmpty() ) {
                    session.setAttribute( ATT_SESSION_USER, utilisateur );
                } else {
                    session.setAttribute( ATT_SESSION_USER, null );
                }

                /* Si et seulement si la case du formulaire est cochée */
        if ( request.getParameter( CHAMP_MEMOIRE ) != null ) {
                    /* Récupération de la date courante */
        DateTime dt = new DateTime();
                    /* Formatage de la date et conversion en texte */
        DateTimeFormatter formatter = DateTimeFormat.forPattern( FORMAT_DATE
        );
        String dateDerniereConnexion = dt.toString( formatter );
                    /* Création du cookie, et ajout à la réponse HTTP */
        setCookie( response, COOKIE_DERNIERE_CONNEXION,
        dateDerniereConnexion, COOKIE_MAX_AGE );
        } else {
                    /* Demande de suppression du cookie du navigateur */
        setCookie( response, COOKIE_DERNIERE_CONNEXION, "", 0 );
        }

                /* Stockage du formulaire et du bean dans l'objet request
        */
                request.setAttribute( ATT_FORM, form );
                request.setAttribute( ATT_USER, utilisateur );

                this.getServletContext().getRequestDispatcher( VUE
        ).forward( request, response );
            }

            ...

            /**
        * Méthode utilitaire gérant la création d'un cookie et son ajout à
        la
        * réponse HTTP.
        */
        private static void setCookie( HttpServletResponse response, String
        nom, String valeur, int maxAge ) {
        Cookie cookie = new Cookie( nom, valeur );
        cookie.setMaxAge( maxAge );
        response.addCookie( cookie );
        }
        }
```

Quelques explications supplémentaires :

- la condition du bloc `if` à la ligne 62 permet de déterminer si la case du formulaire, que j'ai choisi de nommer **memoire**, est cochée ;
- les lignes 66 et 67 se basent sur la convention d'affichage choisie, à savoir `"dd/MM/yyyy HH:mm:ss"`, pour mettre en forme la date proprement à partir de l'objet `DateTime` fraîchement créé ;
- j'utilise alors une méthode `setCookie()`, à laquelle je transmets la réponse accompagnée de trois paramètres :
  - un **nom** et une **valeur**, qui sont alors utilisés pour créer un nouvel objet `Cookie` à la ligne 86 ;
  - un entier **maxAge**, utilisé pour définir la durée de vie du cookie grâce à la méthode `cookie.setMaxAge()` ;
  - cette méthode se base pour finir sur un appel à `response.addCookie()` dont je vous ai déjà parlé, pour mettre en place une instruction **Set-Cookie** dans les en-têtes de la réponse HTTP.
- à la ligne 72, je demande au navigateur du client de supprimer l'éventuel cookie nommé **derniereConnexion** qu'il aurait déjà enregistré par le passé. En effet, si l'utilisateur n'a pas coché la case du formulaire, cela signifie qu'il ne souhaite pas que nous lui affichions un message, et il nous faut donc nous assurer qu'aucun cookie enregistré lors d'une connexion précédente n'existe. Pour ce faire, il suffit de placer un nouveau cookie **dernieреConnexion** dans la réponse HTTP avec une durée de vie égale à zéro.

Au passage, si vous vous rendez sur la documentation de la méthode `setMaxAge()`, vous découvrirez les trois types de valeurs qu'elle accepte :

- un entier positif, représentant **le nombre de secondes** avant expiration du cookie sauvegardé. En l'occurrence, j'ai donné à notre cookie une durée de vie d'un an, soit 60*60*24*365 = 31536000 secondes ;
- un entier négatif, signifiant que le cookie ne sera stocké que de manière temporaire et sera supprimé dès que le navigateur sera fermé. Si vous avez bien suivi et compris le chapitre sur les sessions, alors vous en avez probablement déjà déduit que c'est de cette manière qu'est stocké le cookie **JSESSIONID** ;
- zéro, qui permet de supprimer simplement le cookie du navigateur.

À retenir également, le seul test valable pour s'assurer qu'un champ de type `<input type="checkbox"/>` est coché, c'est de vérifier le retour de la méthode `request.getParameter()` :

- si **null**, alors la case n'est pas cochée ;
- sinon, alors la case est cochée.

Reprise de la JSP

Pour achever notre système, il nous reste à ajouter la case à cocher à notre formulaire, ainsi qu'un message sur notre page de connexion précisant depuis combien de temps l'utilisateur ne s'est pas connecté. Deux contraintes sont à prendre en compte :

- si l'utilisateur est déjà connecté, on ne lui affiche pas le message ;
- si l'utilisateur ne s'est jamais connecté, ou s'il n'a pas coché la case lors de sa dernière connexion, on ne lui affiche pas le message.

Code : JSP - /WEB-INF/connexion.jsp

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Connexion</title>
        <link type="text/css" rel="stylesheet" href="<c:url
value="/inc/form.css"/>" />
    </head>
    <body>
        <form method="post" action="<c:url value="/connexion" />">
            <fieldset>
                <legend>Connexion</legend>
                <p>Vous pouvez vous connecter via ce formulaire.</p>
<c:if test="${empty sessionScope.sessionUtilisateur && !empty
requestScope.intervalleConnexions}">
<p class="info">(Vous ne vous êtes pas connecté(e) depuis ce
navigateur depuis ${requestScope.intervalleConnexions})</p>
</c:if>

                <label for="nom">Adresse email <span
class="requis">*</span></label>
                <input type="email" id="email" name="email"
value="<c:out value="${utilisateur.email}"/>" size="20"
maxlength="60" />
                <span class="erreur">${form.erreurs['email']}</span>
                <br />

                <label for="motdepasse">Mot de passe <span
class="requis">*</span></label>
                <input type="password" id="motdepasse"
name="motdepasse" value="" size="20" maxlength="20" />
                <span
class="erreur">${form.erreurs['motdepasse']}</span>
                <br />
```

```
                <br />
<label for="memoire">Se souvenir de moi</label>
<input type="checkbox" id="memoire" name="memoire" />
                <br />

                <input type="submit" value="Connexion"
class="sansLabel" />
                <br />

                <p class="${empty form.erreurs ? 'succes' :
'erreur'}">${form.resultat}</p>

                <%-- Vérification de la présence d'un objet
utilisateur en session --%>
                <c:if test="${!empty
sessionScope.sessionUtilisateur}">
                    <%-- Si l'utilisateur existe en session, alors
on affiche son adresse email. --%>
                     <p class="succes">Vous êtes connecté(e) avec
l'adresse : ${sessionScope.sessionUtilisateur.email}</p>
                </c:if>
            </fieldset>
        </form>
    </body>
</html>
```

À la ligne 14, vous remarquerez l'utilisation d'un test conditionnel par le biais de la balise **<c:if/>** de la JSTL Core :

- la première moitié du test permet de vérifier que la session ne contient pas d'objet nommé **sessionUtilisateur**, autrement dit de vérifier que l'utilisateur n'est actuellement pas connecté. Souvenez-vous : **sessionUtilisateur** est l'objet que nous plaçons en session lors de la connexion d'un visiteur, et qui n'existe donc que si une connexion a déjà eu lieu ;
- la seconde moitié du test permet de vérifier que la requête contient bien un intervalle de connexions, autrement dit de vérifier si l'utilisateur s'était déjà connecté ou non, et si oui s'il avait coché la case ou non. Rappelez-vous de nos méthodes `doGet()` et `doPost()` : si le client n'a pas envoyé le cookie **derniereConnexion**, alors cela signifie qu'il ne s'est jamais connecté par le passé, ou bien que lors de sa dernière connexion il n'a pas coché la case, et nous ne transmettons pas d'intervalle à la JSP.

Le corps de la balise **<c:if/>**, contenant notre message ainsi que l'intervalle transmis pour affichage à travers l'attribut de requête **intervalleConnexions**, est alors uniquement affiché si à la fois l'utilisateur n'est pas connecté à l'instant présent, et qu'il a coché la case lors de sa précédente connexion.

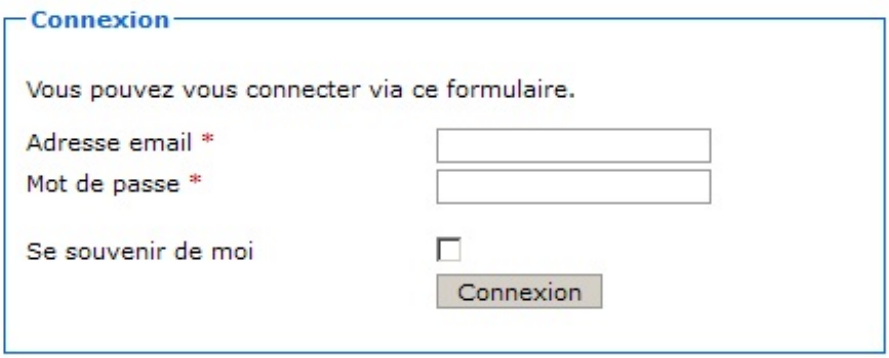
Enfin, vous voyez que pour l'occasion j'ai ajouté un style **info** à notre feuille CSS, afin de mettre en forme le nouveau message :

Code : CSS - /inc/form.css

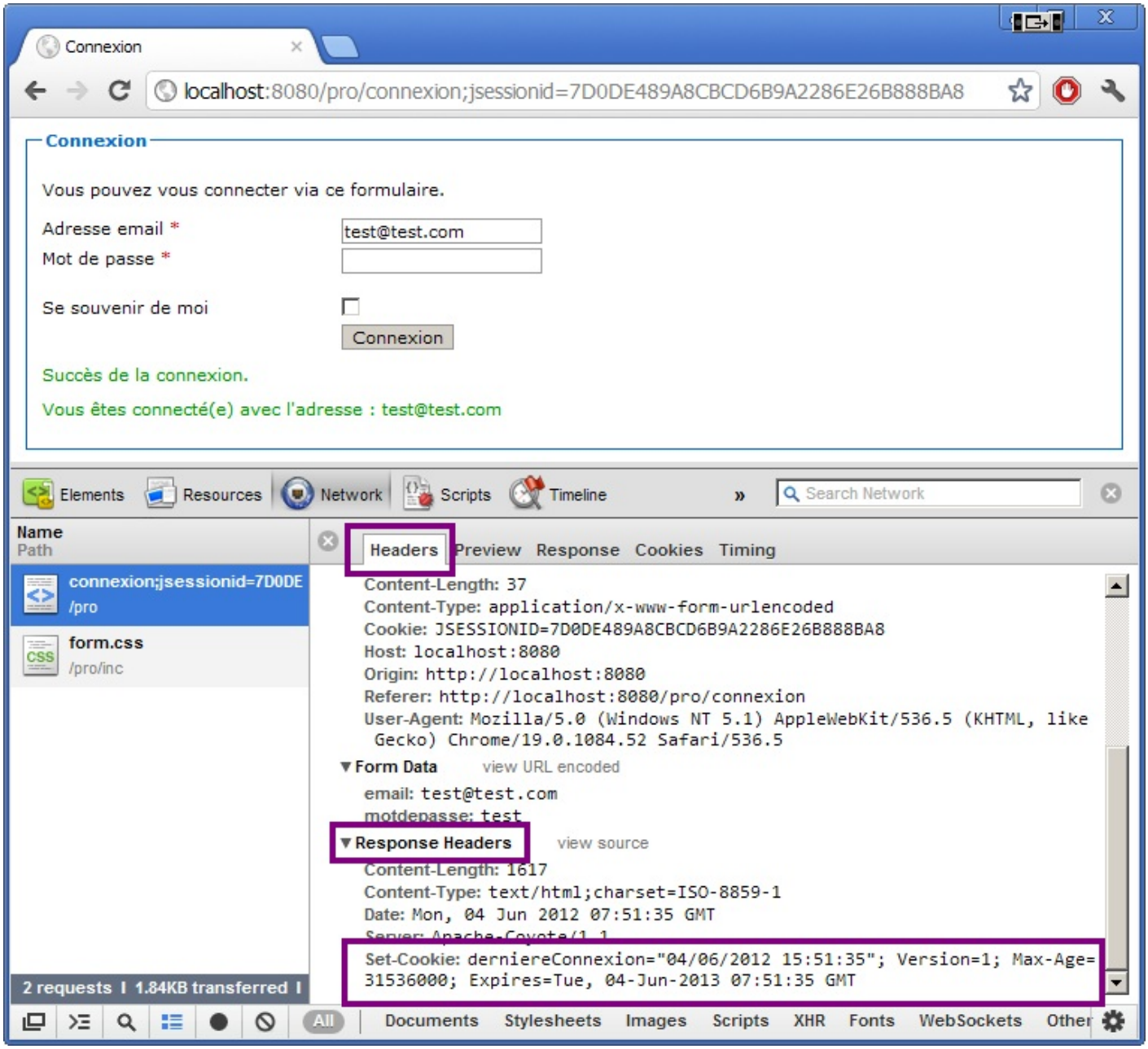
```
form .info {
    font-style: italic;
    color: #E8A22B;
}
```

Vérifications

Le scénario de tests va être plutôt léger. Pour commencer, redémarrez Tomcat, nettoyez les données de votre navigateur via Ctrl + Maj + Suppr, et accédez à la page de connexion. Vous devez alors visualiser le formulaire vierge :



Appuyez sur F12 pour ouvrir l'outil d'analyse de votre navigateur, entrez des données valides dans le formulaire et connectez-vous **en cochant la case "Se souvenir de moi"**. Examinez alors la réponse renvoyée par le serveur :

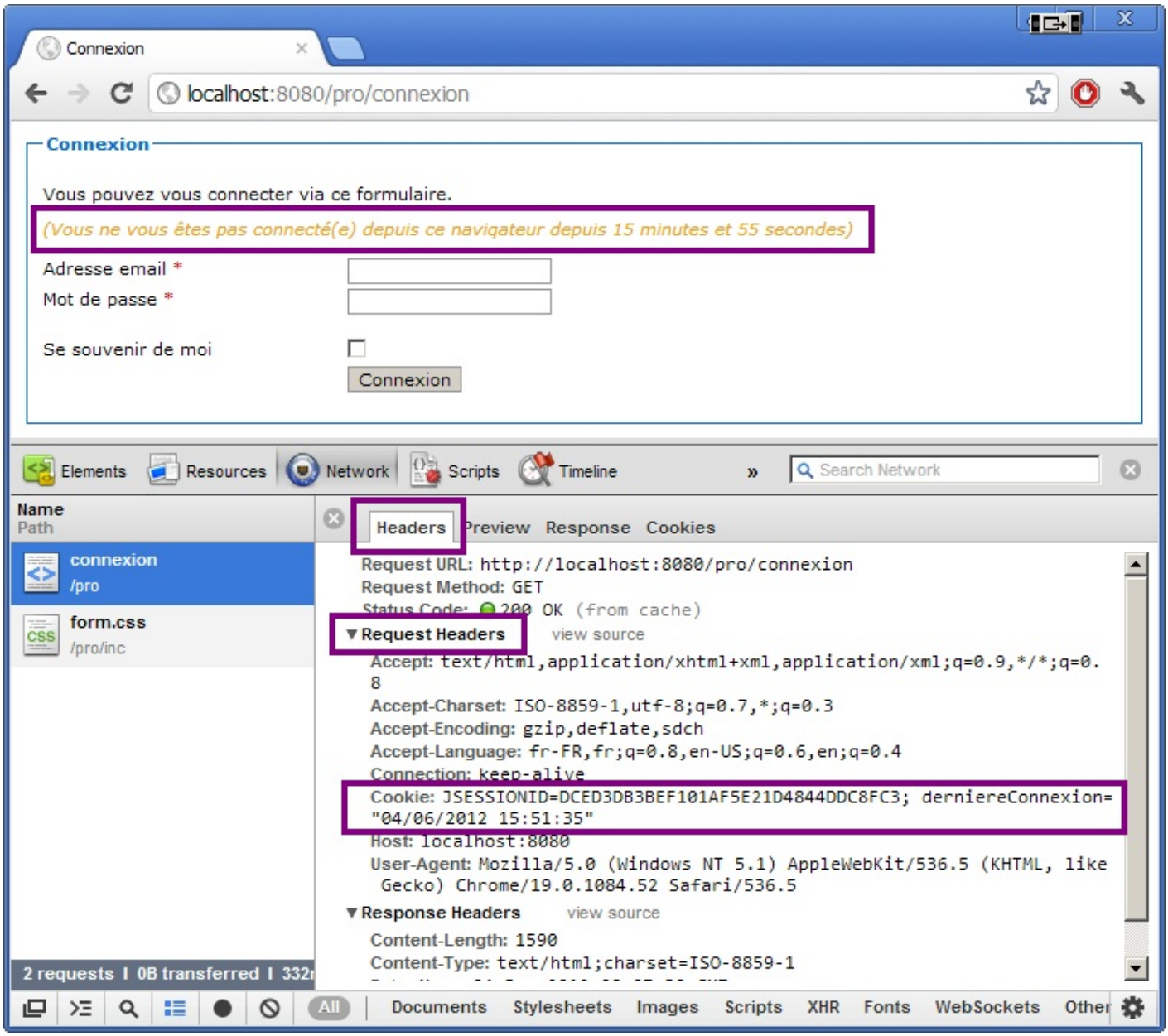


Vous pouvez constater la présence de l'instruction **Set-Cookie** dans l'en-tête de la réponse HTTP, demandant la création d'un cookie nommé **dernièreConnexion** qui :

- contient bien la date de connexion, formatée selon la convention choisie dans notre servlet ;
- expire bien dans 31536000 secondes, soit un an après création.

Ouvrez alors un nouvel onglet et rendez-vous à nouveau sur la page de connexion. Vous constaterez que le message contenant l'intervalle ne vous est toujours pas affiché, puisque vous êtes connectés et que l'expression EL dans notre JSP l'a détecté :

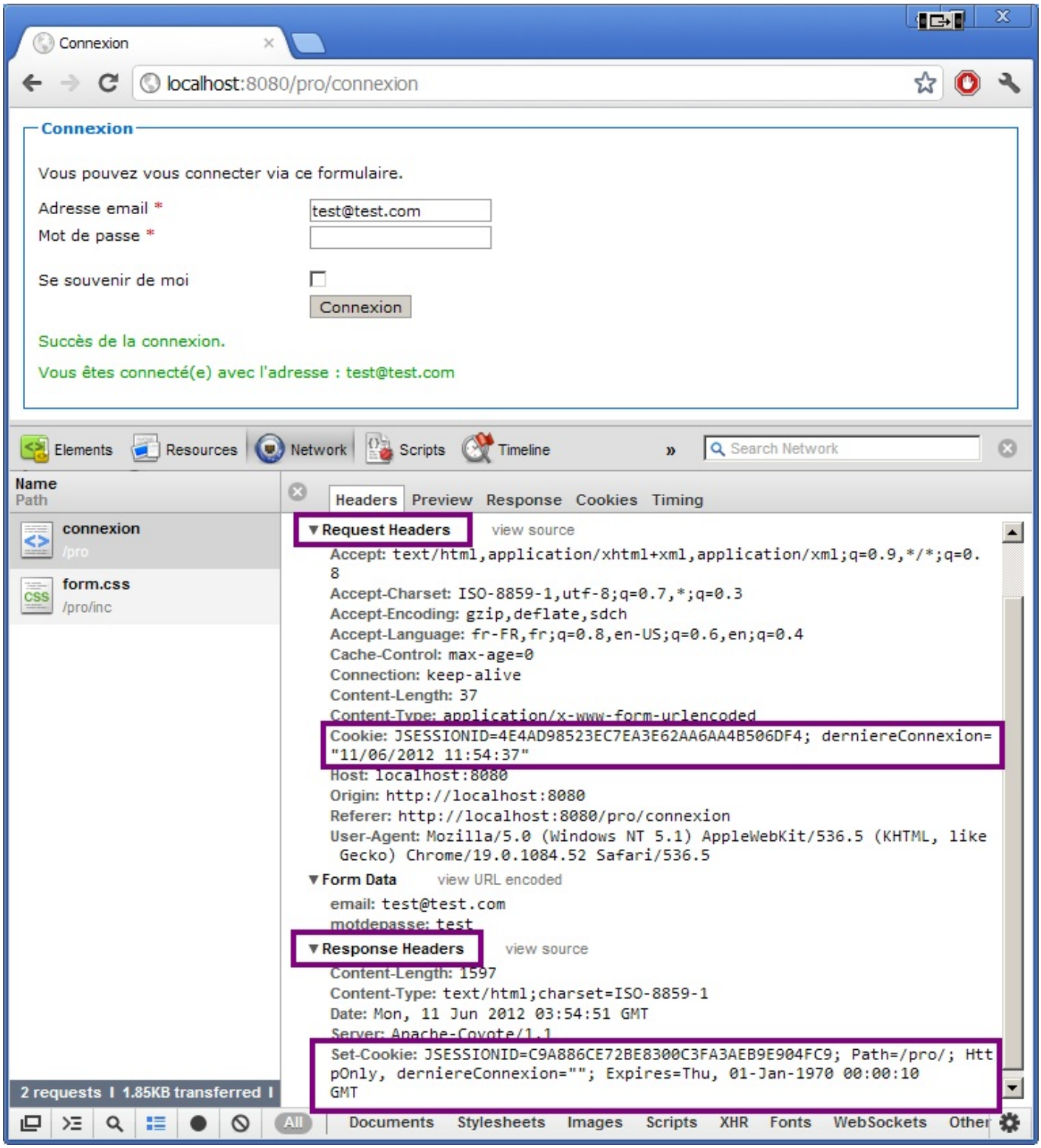
Déconnectez-vous alors en vous rendant sur http://localhost:8080/pro/deconnexion, puis rendez-vous à nouveau sur la page de connexion et observez :



Vous pouvez constater la présence, dans l'en-tête **Cookie** de la requête envoyée par le navigateur, du cookie nommé

dernièreConnexion, sauvegardé lors de la précédente connexion. Et bien évidemment cette fois, le message vous est affiché au sein du formulaire de connexion, et vous précise depuis combien de temps vous ne vous êtes pas connectés.

Connectez-vous à nouveau, mais cette fois **sans cocher la case "Se souvenir de moi"**. Observez alors l'échange HTTP qui a lieu :



Vous pouvez constater deux choses :

- puisque vous vous étiez connectés en cochant la case du formulaire auparavant, votre navigateur avait enregistré la date de connexion dans un cookie, et ce cookie existe encore. Cela explique pourquoi le navigateur envoie un cookie **dernièreConnexion** contenant la date de votre première connexion dans l'en-tête **Cookie** de la requête ;
- puisque cette fois vous n'avez pas coché la case du formulaire, la servlet renvoie une demande de suppression du cookie à travers la réponse HTTP. Cela explique la présence de l'instruction **Set-Cookie** contenant un cookie vide et dont la date d'expiration est le... 1er janvier 1970 !

Pourquoi le cookie contient-il une date d'expiration ?

Tout simplement parce qu'il n'existe pas de propriété ni de champ optionnel dans l'instruction **Set-Cookie** permettant de supprimer un cookie. Ainsi, la seule solution qui s'offre à nous est de préciser une date d'expiration antérieure à la date courante, afin que le navigateur en déduise que le cookie est expiré et qu'il doit le supprimer. Ceci est fait automatiquement lorsque nous donnons à notre cookie un **maxAge** égal à zéro.

Pourquoi le 1er janvier 1970 ?

Il s'agit de la date considérée comme le temps d'origine par votre système. Ainsi en réalité, lorsque nous donnons à notre cookie un **maxAge** égal à zéro, cela se traduit dans l'en-tête HTTP par cette date butoir : de cette manière, le serveur est certain que le cookie sera supprimé par le navigateur, peu importe la date courante sur la machine cliente.

À propos de la sécurité

Vous devez bien comprendre que contrairement aux sessions, bien gardées sur le serveur, le système des cookies est loin d'être un coffre-fort. Principalement parce que l'information que vous y stockez est placée chez le client, et que par conséquent vous n'avez absolument aucun contrôle dessus. Par exemple, rien n'indique que ce que vous y placez ne soit pas lu et détourné par une personne malveillante qui aurait accès à la machine du client à son insu.

L'exemple le plus flagrant est le stockage du nom d'utilisateur et du mot de passe directement dans un cookie. En plaçant en clair ces informations dans le navigateur du client, vous les exposez au vol par un tiers malveillant, qui pourra alors voler le compte de votre client...

Ainsi, il y a une règle à suivre lorsque vous utilisez des cookies : **n'y stockez jamais d'informations sensibles en clair**.

Rassurez-vous toutefois, le système est loin d'être une vraie passoire. Simplement, lorsque vous développerez des applications nécessitant un bon niveau de sécurité, ce sont des considérations que vous devrez tôt ou tard prendre en compte. Autant vous en faire prendre conscience dès maintenant !

La base de connaissance associée aux cookies est plutôt mince, comme vous pouvez le constater à travers ce léger chapitre. En revanche, le panel d'usages qui gravite autour d'eux est immense : nous pouvons par exemple citer la gestion des paniers de commandes sur les sites d'achats en ligne, les systèmes de connexion ou d'authentification automatiques, ou encore le choix de la langue par défaut dans le cas d'un site multilingue, autant de fonctionnalités qui sont très souvent basées sur les cookies !

Dans la partie suivante de ce cours, lorsque nous aurons découvert et assimilé la gestion des bases de données, nous reviendrons sur les cookies et améliorerons notre système de connexion et d'espace membre.

Cette partie est en cours de rédaction.

Nous avons pris les choses par le bon bout, en faisant de gros efforts d'organisation et de découpage du code pour suivre les recommandations MVC.

C'est un très gros morceau que nous venons d'engloutir, mais nous sommes loin d'avoir terminé et j'espère que vous avez encore de l'appétit !
Ce qu'il nous faut maintenant, c'est un moyen de stocker nos données : en route vers les bases de données...

Avancement du cours

Ce tutoriel est en cours de rédaction ! Je publierai de nouveaux chapitres au fur et à mesure dans les semaines à venir. N'hésitez pas à repasser plus tard ou à vous abonner au flux RSS du cours pour suivre son évolution.

Et après ?

J'espère que ce modeste cours vous ouvrira les portes vers des technologies non abordées ici, notamment :

- les frameworks MVC du type JSF, Struts ou Spring ;
- les frameworks de persistance comme JPA et Hibernate.

Autour du développement web en Java, les sujets sont vastes et les outils nombreux, mais tous vous sont accessibles : n'hésitez pas à vous y plonger, et pourquoi pas à vous lancer dans la rédaction d'un tutoriel à leur sujet.