

Créez votre application web avec Java EE

Par Coyote

Ce PDF vous est offert par



Découvrez des métiers plein d'envies

http://www.fr.capgemini.com/carrieres/technology_services/



www.openclassrooms.com

*Licence Creative Commons 6 2.0
Dernière mise à jour le 31/01/2013*

Sommaire

Sommaire	2
Lire aussi	7
Créez votre application web avec Java EE	9
Comment lire ce cours ?	9
Partie 1 : Les bases du Java EE	10
Introduction au Java EE	10
Pour commencer	10
Comment lire ce cours ?	10
Prérequis	10
Qu'est-ce que Java EE ?	10
Java EE n'est pas Java	10
Java EE n'est pas Javascript	10
Internet n'est pas le web !	11
Comment ça marche	11
Les langages du web	12
Le Java EE mis à nu !	12
Principes de fonctionnement	13
Le modèle MVC : en théorie	14
Le modèle MVC : en pratique	15
Outils et environnement de développement	17
L'IDE Eclipse	17
Présentation	17
Téléchargement et installation	17
Configuration	18
Le serveur Tomcat	19
Présentation	19
Installation	19
Création du projet web avec Eclipse	22
Structure d'une application Java EE	32
Structure standard	32
Votre première page web	33
Partie 2 : Premiers pas avec Java EE	37
La servlet	38
Derrière les rideaux	38
Retour sur HTTP	38
Pendant ce temps-là, sur le serveur	39
Création	39
Mise en place	42
Définition de la servlet	43
Mapping de la servlet	44
Mise en service	45
Do you « GET » it?	45
Cycle de vie d'une servlet	47
Envoyer des données au client	48
Servlet avec vue...	51
Introduction aux JSP	51
Nature d'une JSP	51
Mise en place d'une JSP	52
Création de la vue	52
Cycle de vie d'une JSP	54
Mise en relation avec notre servlet	56
Transmission de données	59
Données issues du serveur : les attributs	59
Données issues du client : les paramètres	60
Le JavaBean	64
Objectifs	64
Pourquoi le JavaBean ?	64
Un JavaBean n'est pas un EJB	64
Structure	64
Mise en place	65
Création de notre bean d'exemple	65
Configuration du projet sous Eclipse	68
Mise en service dans notre application	70
La technologie JSP (1/2)	71
Les balises	72
Les directives	73
La portée des objets	76
Les actions standard	78
L'action standard useBean	78
L'action standard getProperty	79
L'action standard setProperty	79
L'action standard forward	80
La technologie JSP (2/2)	81

Expression Language	82
Présentation	82
La réalisation de tests	82
La manipulation d'objets	84
Désactiver l'évaluation des expressions EL	89
Les objets implicites	91
Les objets de la technologie JSP	91
Les objets de la technologie EL	94
Des problèmes de vue ?	99
Nettoyons notre exemple	100
Complétons notre exemple	103
Le point sur ce qu'il nous manque	107
Documentation	107
Liens utiles	107
TP Fil rouge - Étape 1	108
Objectifs	109
Contexte	109
Fonctionnalités	109
Contraintes	109
Conseils	112
À propos des formulaires	112
Le modèle	113
Les contrôleurs	113
Les vues	113
Création du projet	114
Illustration du comportement attendu	114
Exemples de rendu du comportement attendu	115
Correction	118
Le code des beans	118
Le code des servlets	120
Le code des JSP	123
Partie 3 : Une bonne vue grâce à la JSTL	124
Objectifs et configuration	125
C'est sa raison d'être	125
Lisibilité du code produit	125
Moins de code à écrire	126
Vous avez dit MVC ?	126
À retenir	127
Plusieurs versions	127
Configuration	127
Configuration de la JSTL	127
La bibliothèque Core	131
Les variables et expressions	131
Affichage d'une expression	131
Gestion d'une variable	132
Les conditions	135
Une condition simple	135
Des conditions multiples	136
Les boucles	136
Boucle "classique"	136
Itération sur une collection	138
Itération sur une chaîne de caractères	141
Ce que la JSTL ne permet pas (encore) de faire	141
Les liens	141
Liens	141
Redirection	144
Imports	145
Les autres bibliothèques de la JSTL	146
JSTL core : exercice d'application	148
Les bases de l'exercice	148
Correction	150
La bibliothèque xml	153
La syntaxe XPath	154
Le langage XPath	154
Les actions de base	155
Récupérer et analyser un document	155
Afficher une expression	158
Créer une variable	159
Les conditions	159
Les conditions	159
Les boucles	160
Les boucles	160
Les transformations	161
Transformations	161
JSTL xml : exercice d'application	164
Les bases de l'exercice	165
Correction	166
Faisons le point !	167
Reprenons notre exemple	168
Quelques conseils	170
Utilisation de constantes	170

Inclure automatiquement la JSTL Core à toutes vos JSP	172
Formater proprement et automatiquement votre code avec Eclipse	173
Documentation	178
Liens utiles	178
TP Fil rouge - Étape 2	180
Objectifs	180
Utilisation de la JSTL	180
Application des bonnes pratiques	180
Exemples de rendus	180
Conseils	182
Utilisation de la JSTL	182
Application des bonnes pratiques	182
Correction	183
Code des servlets	183
Code des JSP	186
Partie 4 : Une application interactive !	192
Formulaires : le b.a.-ba	192
Mise en place	192
JSP & CSS	193
La servlet	195
L'envoi des données	196
Contrôle : côté servlet	197
Affichage : côté JSP	201
Formulaires : à la mode MVC	209
Analyse de notre conception	209
Création du modèle	209
Reprise de la servlet	213
Reprise de la JSP	215
TP Fil rouge - Étape 3	216
Objectifs	217
Fonctionnalités	217
Exemples de rendus	217
Conseils	219
Correction	220
Code des objets métier	220
Code des servlets	226
Code des JSP	228
La session : connectez vos clients	233
Le formulaire	233
Le principe de la session	233
Le modèle	236
La servlet	238
Les vérifications	240
Test du formulaire de connexion	240
Test de la destruction de session	242
Derrière les rideaux	247
La théorie : principe de fonctionnement	247
La pratique : scrutons nos requêtes et réponses	248
En résumé	261
Le filtre : créez un espace membre	262
Restreindre l'accès à une page	263
Les pages d'exemple	263
La servlet de contrôle	264
Test du système	265
Le problème	266
Le principe du filtre	266
Généralités	266
Fonctionnement	267
Cycle de vie	268
Restreindre l'accès à un ensemble de pages	269
Restreindre un répertoire	269
Restreindre l'application entière	273
Désactiver le filtre	280
Modifier le mode de déclenchement d'un filtre	280
Retour sur l'encodage UTF-8	281
Le cookie : le navigateur vous ouvre ses portes	284
Le principe du cookie	284
Côté HTTP	284
Côté Java EE	284
Souvenez-vous de vos clients !	284
Reprise de la servlet	285
Reprise de la JSP	289
Vérifications	291
À propos de la sécurité	294
TP Fil rouge - Étape 4	295
Objectifs	295
Fonctionnalités	295
Exemples de rendus	295
Conseils	298
Correction	300
Le code des vues	300

Le code des servlets	306
Le code des objets métiers	312
Formulaires : l'envoi de fichiers	317
Création du formulaire	317
Récupération des données	317
Mise en place	317
Traitement des données	319
La différence entre la théorie et la pratique	325
Enregistrement du fichier	327
Définition du chemin physique	327
Écriture du fichier sur le disque	328
Test du formulaire d'upload	331
Problèmes et limites	331
Comment gérer les fichiers de mêmes noms ?	331
Comment éviter les doublons ?	331
Où stocker les fichiers reçus ?	332
Rendre le tout entièrement automatique	332
Intégration dans MVC	333
Création du bean représentant un fichier	333
Création de l'objet métier en charge du traitement du formulaire	333
Reprise de la servlet	338
Adaptation de la page JSP aux nouvelles informations transmises	339
Comportement de la solution finale	340
Le téléchargement de fichiers	342
Une servlet dédiée	342
Création de la servlet	342
Paramétrage de la servlet	342
Analyse du fichier	344
Génération de la réponse HTTP	345
Lecture et envoi du fichier	346
Vérification de la solution	347
Une solution plus simple	348
L'état d'un téléchargement	348
Réaliser des statistiques	349
TP Fil rouge - Étape 5	349
Objectifs	350
Fonctionnalités	350
Conseils	351
Envoi du fichier	351
Validation et enregistrement du fichier	351
Affichage d'un lien vers l'image	352
Ré-affichage de l'image	353
Correction	353
Le code des objets métiers	353
Le code de l'exception personnalisée	362
Le code des servlets	362
Le code des JSP	369
Partie 5 : Les bases de données avec Java EE	375
Introduction à MySQL et JDBC	375
Présentation des bases de données	375
Structure	375
SGBD	376
SQL	377
Préparation de la base avec MySQL	377
Installation	377
Création d'une base	379
Création d'un utilisateur	379
Création d'une table	380
Insertion de données d'exemple	380
Mise en place de JDBC dans le projet	380
JDBC	380
Mise en place	381
Création d'un bac à sable	381
Création de l'objet Java	382
Création de la servlet	382
Création de la page JSP	383
Communiquez avec votre BDD	385
Chargement du driver	385
Connexion à la base, création et exécution d'une requête	385
Connexion à la base de données	385
Création d'une requête	387
Exécution de la requête	388
Accès aux résultats de la requête	388
Libération des ressources	390
Mise en pratique	391
Afficher le contenu de la table Utilisateur	391
Insérer des données dans la table Utilisateur	393
Les limites du système	395
Insérer des données saisies par l'utilisateur	395
Le problème des valeurs nulles	395
Le cas idéal	397
Les injections SQL	397

Les requêtes préparées	398
Pourquoi préparer ses requêtes ?	398
Comment préparer ses requêtes ?	400
Mise en pratique	402
Le modèle DAO	405
Objectifs	406
Inconvénients de notre solution	406
Isoler le stockage des données	406
Principe	407
Constitution	407
Intégration	408
Création	408
Modification de la table Utilisateur	408
Reprise du bean Utilisateur	409
Création des exceptions du DAO	411
Création d'un fichier de configuration	412
Création d'une Factory	412
Création de l'interface du DAO Utilisateur	415
Création de l'implémentation du DAO	416
Intégration	422
Chargement de la DAOFactory	423
Utilisation depuis la servlet	424
Reprise de l'objet métier	426
Création d'une exception dédiée aux erreurs de validation	429
Vérifications	429
Le code final	429
Le scénario de tests	430
TP Fil rouge - Étape 6	432
Objectifs	433
Fonctionnalités	433
Conseils	433
Création de la base de données	433
Mise en place de JDBC	435
Réutilisation de la structure DAO développée dans le cadre du cours	435
Création des interfaces et implémentations du DAO	435
Intégration dans le code existant	436
Correction	438
Code de la structure DAO	438
Code des interfaces DAO	440
Code des implémentations DAO	441
Code des beans	447
Code des objets métier	449
Code des servlets	459
Code du filtre	467
Code des JSP	469
Gérer un pool de connexions avec BoneCP	474
Contexte	475
Une application multi-utilisateurs	475
Le coût d'une connexion à la BDD	475
La structure actuelle de notre solution	475
Principe	476
Réutilisation des connexions	476
Remplacement du DriverManager par une DataSource	477
Choix d'une implémentation	477
Mise en place	477
Ajout des jar au projet	477
Prise en main de la bibliothèque	478
Modification de la DAOFactory	478
Vérifications	481
Configuration fine du pool	481
En résumé	481
Partie 6 : Aller plus loin avec JPA et JSF	482
Les annotations	483
Présentation	483
Écrire des méta-données	483
Pallier certaines carences	483
Simplifier le développement	484
Principe	484
Syntaxe sans paramètres	484
Syntaxe avec paramètres	484
Avec l'API Servlet 3.0	485
WebServlet	485
WebFilter	486
WebInitParam	487
WebListener	487
MultipartConfig	488
Et le web.xml dans tout ça ?	489
La persistance des données avec JPA	490
Généralités	491
Principe	492
Des EJB dans un conteneur	492
Un gestionnaire d'entités	493

Mise en place	493
Le serveur d'applications GlassFish	493
Création du projet	497
Création d'une entité Utilisateur	499
Création d'un EJB Session	500
Modification de la servlet	503
Modification de l'objet métier	505
Tests et vérifications	506
Vérification du bon fonctionnement d'une inscription	506
Analyse des requêtes SQL générées lors d'une inscription	506
Aller plus loin	508
ORM, ou ne pas ORM ?	508
TP Fil rouge - Étape 7	511
Objectifs	511
Fonctionnalités	511
Conseils	511
Environnement de développement	511
Reprise du code existant	511
Correction	513
Le code de configuration	513
Le code des EJB Entity	514
Le code des EJB Session	517
Le code des servlets	519
Le code du filtre	527
Introduction aux frameworks MVC	530
Généralités	530
Rappel concernant MVC	530
Qu'est-ce qu'un framework MVC ?	530
À quels besoins répond-il ?	530
Quand utiliser un framework MVC, et quand s'en passer ?	530
Framework MVC basé sur les requêtes	531
Définition	531
Principe	531
Solutions existantes	531
Framework MVC basé sur les composants	531
Définition	531
Principe	532
Solutions existantes	532
Les "dissidents"	532
Premiers pas avec JSF	533
Qu'est-ce que JSF ?	534
Présentation	534
Principe	534
Historique	534
Structure d'une application JSF	536
Facelets et composants	537
La page JSP mise au placard ?	537
Structure et syntaxe	537
Comment ça marche ?	540
Créer un template de Facelet par défaut avec Eclipse	541
Premier projet	543
De quoi avons-nous besoin ?	543
Création du projet	544
Création du bean	544
Création des facelets	545
Configuration de l'application	547
Tests & observations	548
Les ressources	551
JSF, ou ne pas JSF ?	553
La gestion d'un formulaire avec JSF	555
Une inscription classique	555
Préparation du projet	555
Création de la couche d'accès aux données	555
Création du backing bean	556
Création de la vue	558
Tests & observations	561
Amélioration des messages affichés lors de la validation	562
Une inscription ajaxisée	565
Présentation	565
L'AJAX avec JSF	565
L'importance de la portée d'un objet	567
Une inscription contrôlée	568
Déporter la validation de la vue vers l'entité	568
Affiner les contrôles effectués	570
Ajouter des contrôles "métier"	571
L'envoi de fichiers avec JSF	579
Le problème	579
Les bibliothèques de composants	580
L'envoi de fichier avec Tomahawk	580
Préparation du projet	580
Création de la Facelet	581
Création du JavaBean	582

Création du backing-bean	582
Tests et vérifications	584
Limitation de la taille maximale autorisée	585
Et plus si affinités...	587
TP Fil rouge - Étape 8	587
Ce que l'avenir vous réserve	588
Partie 7 : Annexes	589
Débugger un projet	590
Les fichiers de logs	590
Quels fichiers ?	590
Comment les utiliser ?	592
Le mode debug d'Eclipse	592
Principe	592
Interface	593
Exemple pratique	594
Conseils au sujet de la thread-safety	598
Quelques outils de tests	599
Les tests unitaires	599
Les tests de charge	601
Empaquetage et déploiement d'un projet	603
Mise en boîte du projet	603
JAR, WAR ou EAR ?	603
Mise en pratique	604
Déploiement du projet	604
Contexte	604
Mise en pratique	605
Avancement du cours	606
Et après ?	606



Créez votre application web avec Java EE



Par

Coyote

Mise à jour : 31/01/2013

Difficulté : Intermédiaire Durée d'étude : 2 mois



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.



Je profite de cette introduction pour tordre le coup à une erreur trop courante : l'appellation « JEE » n'existe pas ! Les créateurs de Java EE ont même dédié une page web à cette fausse appellation.

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 : Les bases du 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...

Introduction au Java EE

Avant de nous plonger dans l'univers Java EE, commençons par faire une mise au point sur ce que vous devez connaître avant d'attaquer ce cours, et penchons-nous 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 existants, et les raisons qui nous poussent à choisir Java EE.

Pour commencer

Comment lire ce cours ?

Un contenu conséquent est prévu dans ce cours, mais je ne vais volontairement pas être exhaustif : les technologies abordées sont très vastes, et l'objectif est avant tout 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 ; plus vous ferez preuve de curiosité et d'assiduité, plus votre apprentissage sera efficace.

Prérequis

Avant d'attaquer, sachez que ce cours ne part pas totalement de zéro :

- **des notions en développement Java sont nécessaires** ([lire les parties 1 et 2 du cours de Java](#)) ;
- des notions en langages HTML et CSS sont préférables, pour une meilleure compréhension des exemples ([lire le cours de HTML5 / CSS3](#)) ;
- des notions en langage SQL sont préférables, pour une meilleure compréhension de la partie 5 du cours ([lire le cours de MySQL](#)).

Qu'est-ce que Java EE ?

Pour commencer, tordons le coup à certaines confusions plutôt tenaces chez les débutants...

Java EE n'est pas Java

Le terme « Java » fait bien évidemment référence à un langage, mais également à une plate-forme : son nom complet est « Java SE » pour *Java Standard Edition*, et était anciennement raccourci « J2SE ». Celle-ci est constituée de nombreuses bibliothèques, ou API : citons par exemple `java.lang`, `java.io`, `java.math`, `java.util`, etc. Bref, toutes ces bibliothèques que vous devez déjà connaître et qui contiennent un nombre conséquent de classes et de méthodes prêtes à l'emploi pour effectuer toutes sortes de tâches.

Le terme « Java EE » signifie *Java Enterprise Edition*, et était anciennement raccourci en « J2EE ». Il fait quant à lui référence à une extension de la plate-forme standard. Autrement dit, la plate-forme Java EE est construite sur le langage Java et la plate-forme Java SE, et elle y ajoute un grand nombre de bibliothèques remplissant tout un tas de fonctionnalités que la plate-forme standard ne remplit pas d'origine. L'objectif majeur de Java EE est de faciliter le développement d'applications web robustes et distribuées, déployées et exécutées sur un serveur d'applications. Inutile de rentrer plus loin dans les détails, tout ceci étant bien entendu l'objet des chapitres à venir.

Si le cœur vous en dit, vous pouvez consulter [les spécifications de la plate-forme Java EE actuelle](#), finalisées depuis décembre 2009.

Java EE n'est pas Javascript

S'il est vrai que Java EE permet la création d'applications web, il ne faut pas pour autant le confondre avec le langage Javascript, souvent raccourci en « JS », qui est lui aussi massivement utilisé dans les applications web. Ce sont là deux langages totalement

différents, qui n'ont comme ressemblance que leur nom ! En d'autres termes, Java est au Javascript ce que le bal est à la balustrade... ☺

Ne vous leurrez donc pas, et lorsque vous entendrez parler de *scripts Java*, rappelez-vous bien que cela désigne simplement du code Java, et surtout pas du code Javascript.

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 réseau 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.

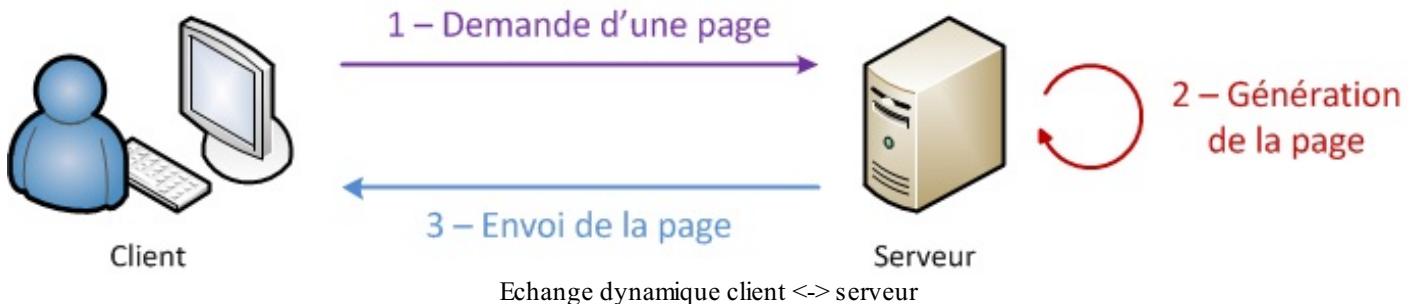
Un site web est un ensemble constitué de pages web (elles-mêmes faites de fichiers HTML, CSS, Javascript, etc.). Lorsqu'on développe puis publie un site web, on met en réalité en ligne du contenu sur internet. On distingue deux types de sites :

- **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 !

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 (voir la figure suivante).

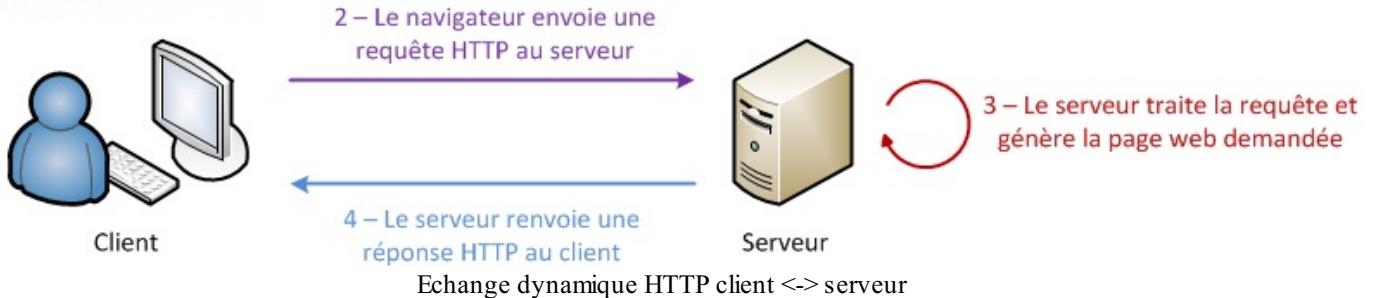
- **le client** : dans la plupart des cas, c'est le navigateur installé sur votre ordinateur. Retenez que ce n'est pas le seul moyen d'accéder au web, mais c'est celui qui nous intéresse dans ce cours.
- **le serveur** : c'est la machine sur laquelle le site est hébergé, où les fichiers sont stockés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



La communication qui s'effectue entre le client et le serveur est régie par des règles bien définies : le **protocole HTTP** (voir la figure suivante). 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.

1 - Le client saisit une URL



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 poussés à lire ce cours, nous sommes bien là pour apprendre le Java EE ! 😊

- Java EE est une extension de la plate-forme standard Java SE, principalement destinée au développement d'applications web.
- Internet désigne le réseau physique ; le web désigne le contenu accessible à travers ce réseau.
- Pour interagir avec un site web (le serveur), l'utilisateur (le client) passe par son navigateur.
- À travers le protocole HTTP, le navigateur envoie des requêtes au serveur et le serveur lui renvoie des réponses :
 - le travail du serveur est de recevoir des requêtes, de générer les pages web et de les envoyer au client.
 - le travail du navigateur est de transmettre les actions de l'utilisateur au serveur, et d'afficher les informations qu'il renvoie.

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).



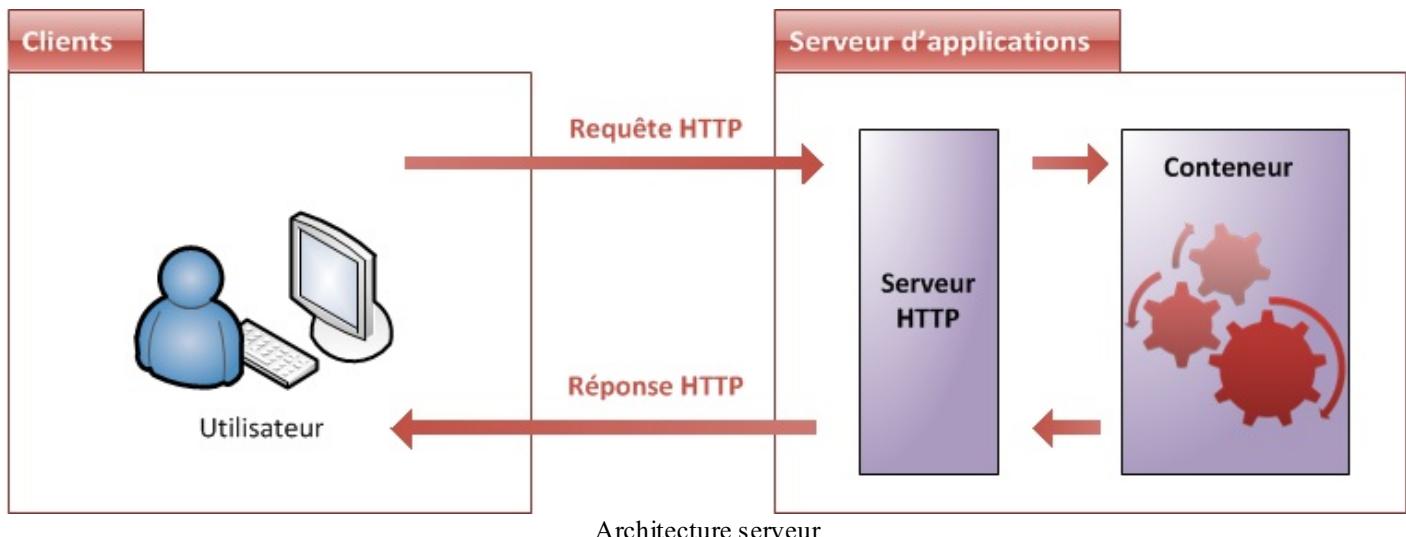
Cependant, nous n'allons directement utiliser ni l'un ni l'autre. Pourquoi ?

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 ce qu'il faut 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 web à part entière, incluant des *plugins* qui les rendent capables de supporter des langages de script 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** (voir la figure suivante).



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 qui sont 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. Ça 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 opte éventuellement pour une telle solution.

Avant de découvrir et de 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 entreprise implique entre autres :

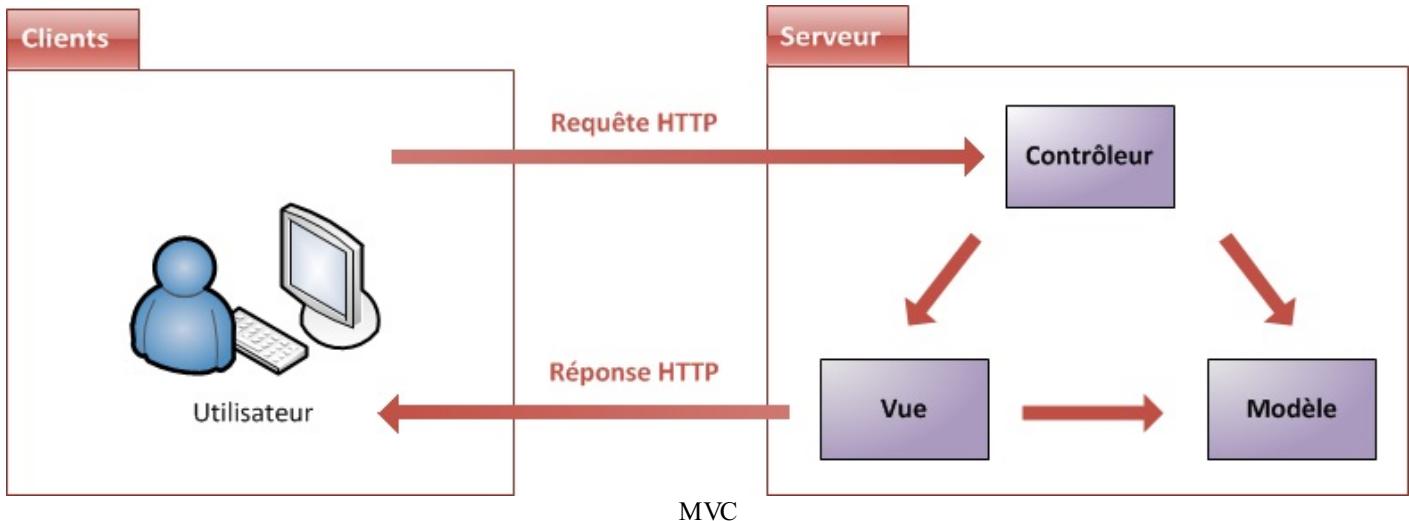
- que l'on puisse être amené à travailler à plusieurs contributeurs sur un même projet ou une même application (travail en équipe) ;
- que l'on puisse être amené à maintenir et corriger une application que l'on n'a pas créée soi-même ;
- que l'on puisse être amené à faire évoluer une application que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



Ne vous faites pas de souci 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, afin de 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 données produites 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, et dépend fortement de l'utilisation ou non de frameworks...



Qu'est-ce qu'un framework ?

Il est encore bien trop tôt pour que nous nous penchions sur ce sujet. Toutefois, vous pouvez d'ores et déjà retenir qu'un *framework* est un ensemble de composants qui servent à créer l'architecture et les grandes lignes d'une application. Vous pouvez le voir comme une boîte à outils géante, conçue par un ou plusieurs développeurs et mise à disposition d'autres développeurs, afin de faciliter leur travail. Il existe des *frameworks* dans beaucoup de langages et plate-formes, ce n'est pas un concept propre à Java EE ni au développement web en particulier.

En ce qui concerne Java EE, nous pouvons par exemple citer JSF, Spring, Struts ou encore Hibernate. Toutes ces solutions sont des *frameworks* que les développeurs sont libres d'utiliser ou non dans leurs projets.

Bref, nous sommes encore loin d'être assez à l'aise avec la plate-forme Java EE pour étudier ces fameux *frameworks*, mais nous pouvons d'ores et déjà étudier les applications Java EE "nues", sans *frameworks* ni fioritures. Voici donc une courte introduction de chacune des couches composant une telle 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 des méthodes (traitements) qui leur sont propres, et un système capable de stocker des données d'autre part. Rien de bien transcendant ici, la complexité du code dépendra bien évidemment de la complexité des traitements à effectuer par votre application.

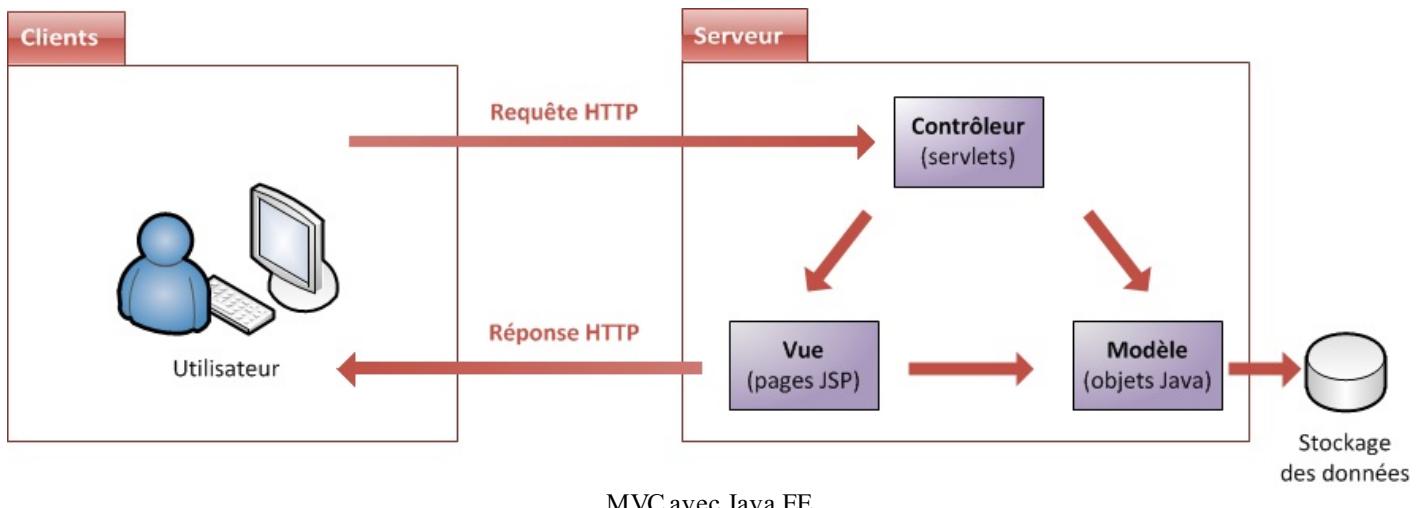
Vue : des pages JSP

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

Contrôleur : des servlets

Une servlet est un objet qui permet d'intercepter les requêtes faites par un client, et qui peut personnaliser une réponse en conséquence. Il fournit pour cela des méthodes permettant de scruter les requêtes HTTP. **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 (voir la figure suivante).



- Un serveur d'applications est constitué d'un serveur HTTP et d'un conteneur web.
- Le modèle de conception MVC impose une répartition stricte des tâches au sein d'une application :
 - la couche Modèle se charge des traitements à effectuer sur les données et de leur stockage ;
 - la couche Vue se charge de la présentation des données pour l'utilisateur et de l'interaction ;
 - la couche Contrôle se charge d'aiguiller les requêtes entrantes vers les traitements et vues correspondants.
- Un framework est une boîte à outils mise à disposition du développeur pour lui alléger certaines tâches.
- Dans une application Java EE sans frameworks :
 - la couche Modèle est constituée d'objets Java ;
 - la couche Vue est constituée de pages JSP ;
 - la couche Contrôle est constituée de servlets.

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 en *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 aisément centralisé des composants d'une application ;
- multiples moyens de visualisation de l'architecture d'une application ;
- génération automatique de portions de code ;
- assistance à la volée lors de l'écriture du code ;
- outils de débogage...

Téléchargement et installation

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 (voir la figure suivante).

The screenshot shows the Eclipse Downloads page with a purple header. Below it, there's a navigation bar with tabs for 'Packages', 'Developer Builds', and 'Projects'. A dropdown menu shows 'Eclipse Indigo (3.7.1) Packages for Windows'. The main content area lists four packages:

- Eclipse IDE for Java Developers**, 128 MB
Downloaded 3,759,620 Times [Details](#) [Windows 32 Bit](#) [Windows 64 Bit](#)
- Eclipse IDE for Java EE Developers**, 212 MB
Downloaded 2,650,976 Times [Details](#) [Windows 32 Bit](#) [Windows 64 Bit](#)
- Eclipse Classic 3.7.1**, 174 MB
Downloaded 1,241,158 Times [Details](#) [Other Downloads](#) [Windows 32 Bit](#) [Windows 64 Bit](#)
- Eclipse IDE for C/C++ Developers (includes Incubating components)**, 107 MB
Downloaded 788,834 Times [Details](#) [Windows 32 Bit](#) [Windows 64 Bit](#)

At the bottom of the page, it says 'Page de téléchargement d'Eclipse pour Java EE'.

Cliquez sur "Eclipse IDE for Java EE Developers", puis choisissez et téléchargez la version correspondant à votre système d'exploitation, comme indiqué à la figure suivante.

The screenshot shows the "Package Details" section with a brief description of tools for Java EE and Web application development. The "Feature List" section lists several features: org.eclipse.csv, org.eclipse.datatools.common.doc.user, and org.eclipse.datatools.connectivity.doc.user. The "Download Links" section provides links for Windows 32-bit, Windows 64-bit, Mac OS X (Cocoa 32 and 64), Linux 32-bit, and Linux 64-bit, with a note that it has been downloaded 2,651,296 times. A link to "Checksums..." is also present.

Choix de 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. Typiquement, évitez les dossiers du genre "Program Files" et consorts. 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.

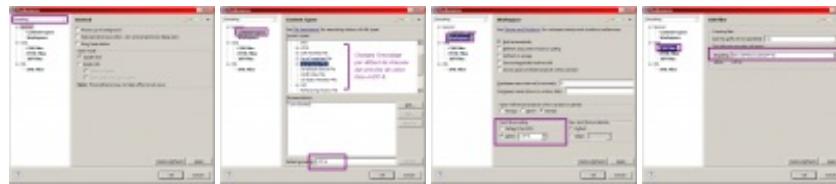
Notez bien toutefois que je vous conseille de ne pas procéder ainsi, et de repartir d'une version vierge d'Eclipse pour Java EE.

Configuration

Ci-dessous, je vous donne quelques conseils pour configurer votre Eclipse efficacement. Je ne vais pas vous expliquer en détail pourquoi ces réglages sont importants, faites-moi simplement confiance et suivez le guide !

Modification de l'encodage par défaut

Si vous ouvrez Eclipse pour la première fois, commencez par fermer l'onglet de bienvenue qui s'affiche. Rendez-vous alors dans la barre de menus supérieure, et cliquez sur Window, puis Preferences. Dans la fenêtre qui s'affiche alors, il y a un champ vous permettant de taper du texte en haut à gauche. Saisissez-y le mot "*encoding*", et dans chaque section qui apparaît alors dans le volet de gauche, changez l'encodage par défaut (il est généralement réglé à Cp1252 ou ISO-8859-1) par la valeur **UTF-8**, comme indiqué à la figure suivante.

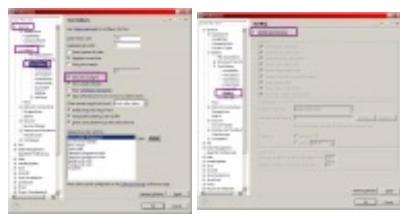


Validez pour finir en cliquant sur Ok afin d'appliquer les modifications.

Désactivation de la vérification de l'orthographe

Rendez-vous à nouveau dans le menu Window > Preferences, puis dans le volet de gauche rendez-vous dans General > Editors > Text Editors, et dans le volet de droite cochez la case "Show line numbers". Dans le volet de gauche, cliquez

alors sur le sous-menu Spelling, et dans le nouveau volet de droite qui apparaît, décochez la case "Enable spell checking" (voir la figure suivante).



Validez pour finir en cliquant sur Ok afin d'appliquer les modifications.

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, comme indiqué à la figure suivante.

Mirrors

You are currently using <http://apache.etoak.com/>. If you encounter a problem with this mirror, please

Other mirrors:

7.0.26

Please see the [README](#) file for packaging information. It explains what every distribution contains.

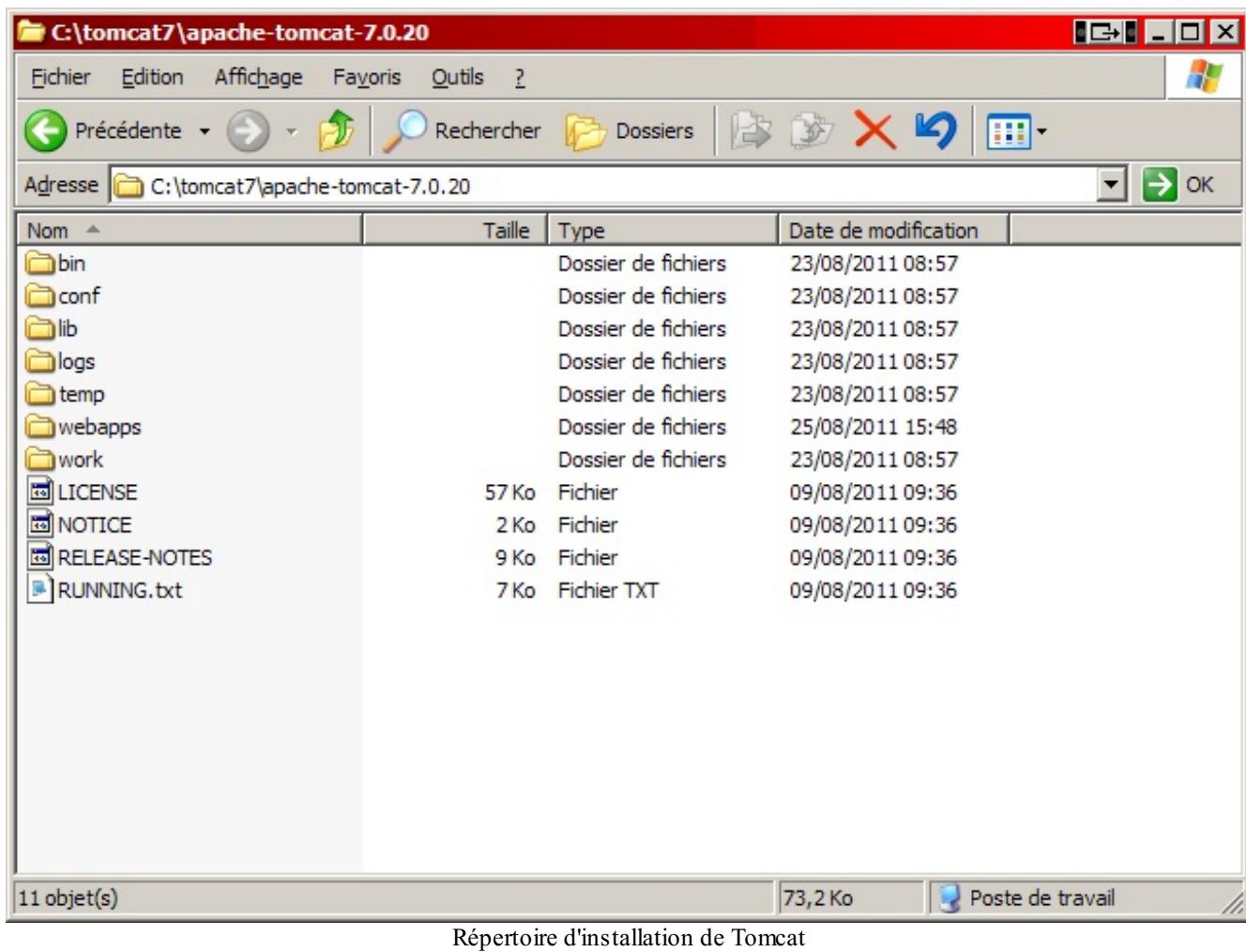
Binary Distributions

- Core:
 - [zip \(pgp, md5\)](#)
 - [tar.gz \(pgp, md5\)](#)
 - [32-bit Windows zip \(pgp, md5\)](#)
 - [64-bit Windows zip \(pgp, md5\)](#)
 - [64-bit Itanium Windows zip \(pgp, md5\)](#)
 - [32-bit/64-bit Windows Service Installer \(pgp, md5\)](#)
- Full documentation:
 - [tar.gz \(pgp, md5\)](#)

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 sous l'appellation *Tomcat Home*. Voici à la figure suivante ce que j'obtiens sur mon poste.



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 œil 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](#) ; 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 sous 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 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 Mac OS

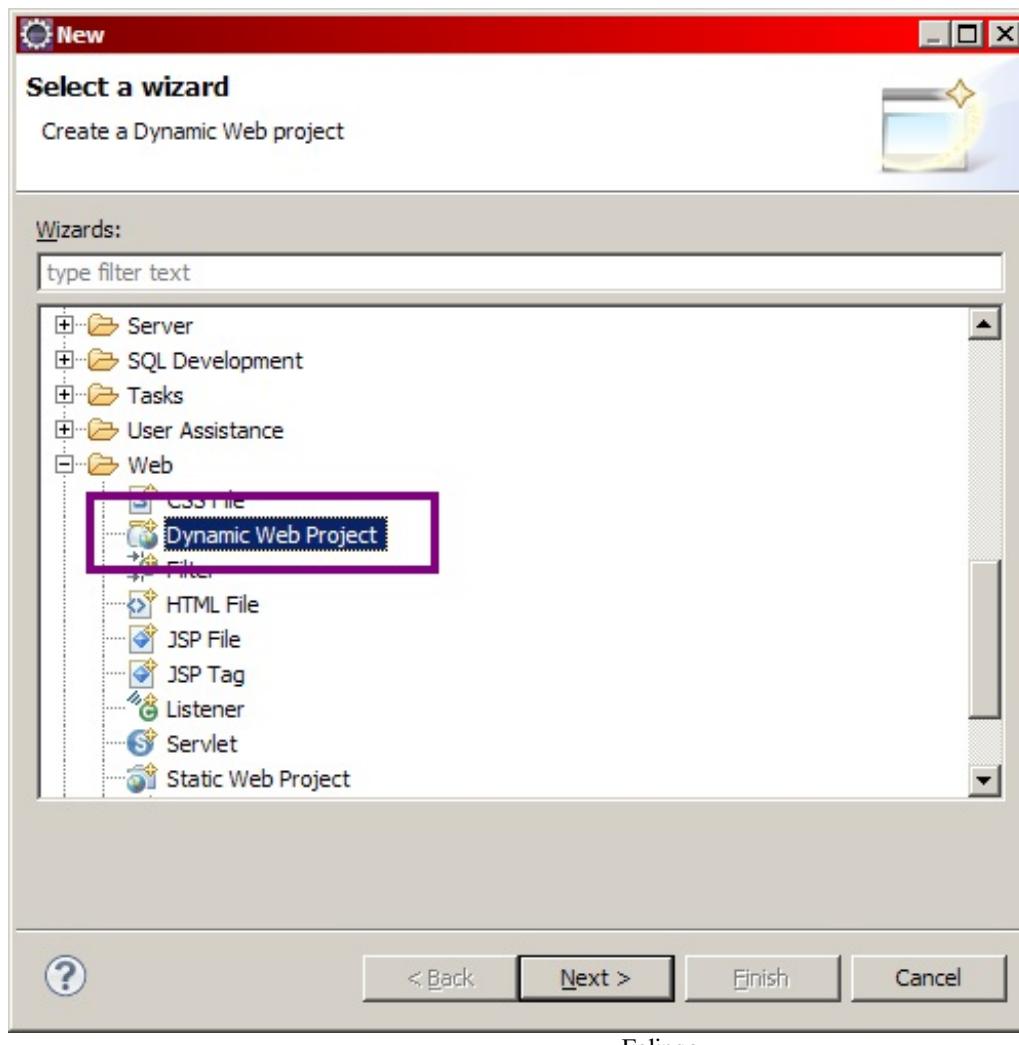
Je n'ai malheureusement pas à ma disposition de machine tournant sous Mac OS. Si vous êtes un aficionado de la marque à la pomme, voici deux liens qui expliquent comment installer Tomcat 7 sur OS X :

- 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'éventuelles erreurs ou changements dans le procédé présenté, je modifierai cette section du chapitre en conséquence.

Création du projet web avec Eclipse

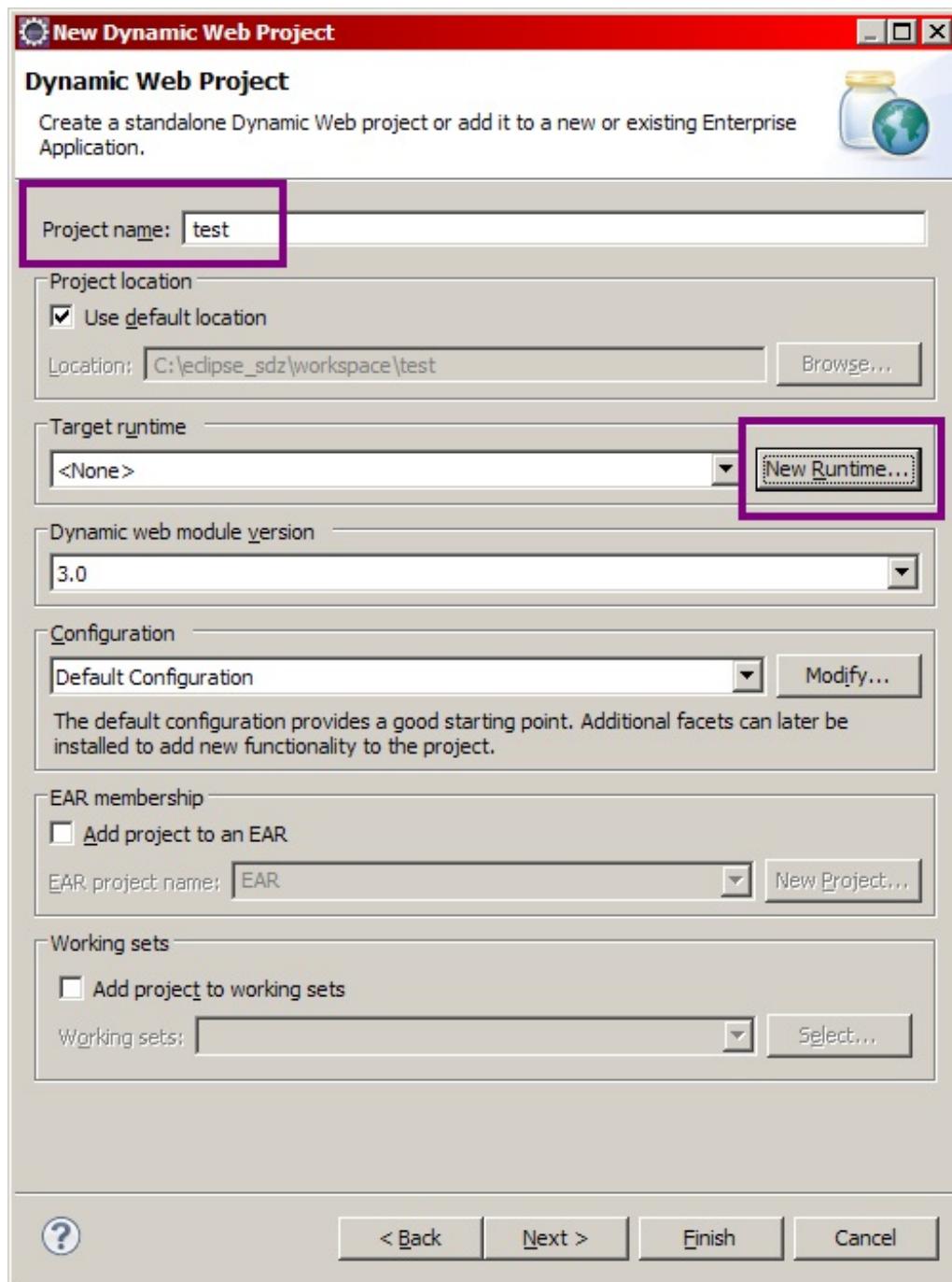
Depuis Eclipse, suivez le chemin suivant : File > New > Project... (voir la figure suivante). Ceci peut d'ailleurs être raccourci en tapant au clavier Ctrl + N.



Nouveau projet web sous

Eclipse

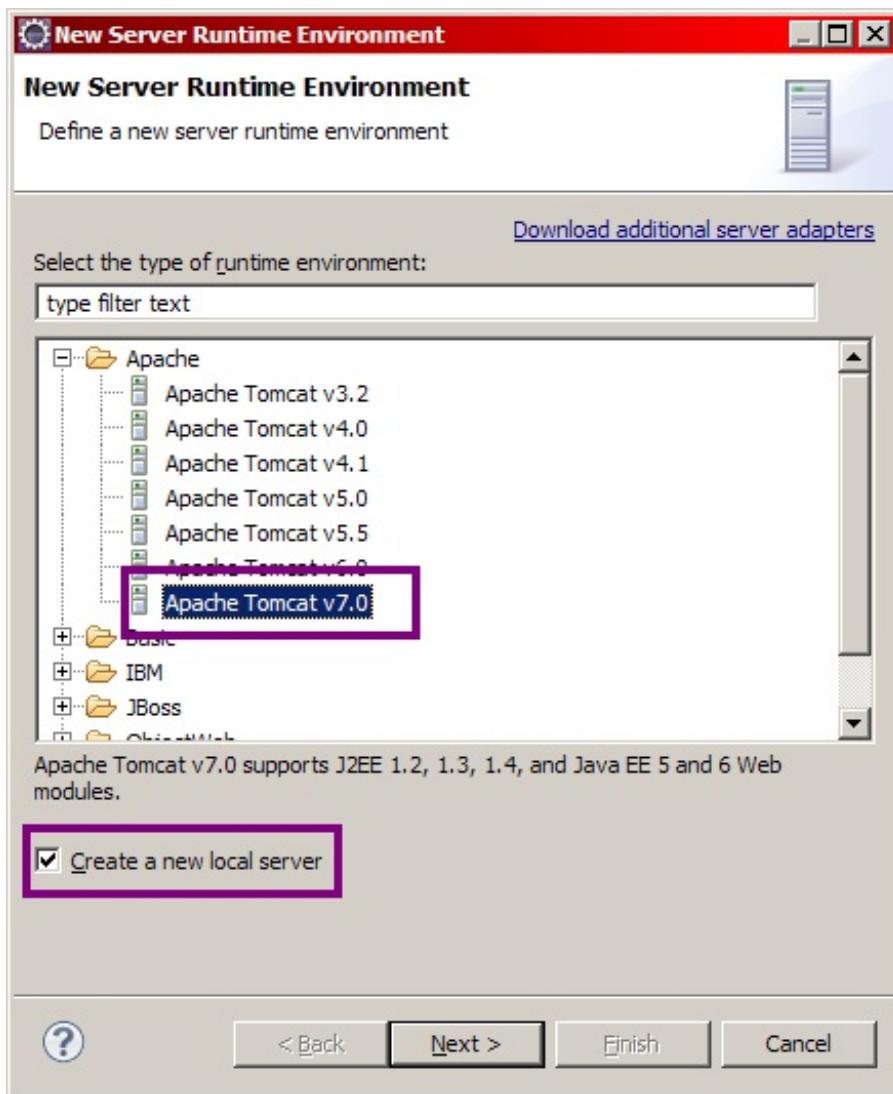
Sélectionnez alors **Dynamic Web Project** comme le montre l'image ci-dessus, puis cliquez sur **Next >**. J'appelle ici mon projet **test**. Remarquez ensuite à la figure suivante le passage concernant le serveur.



Mise en place de Tomcat - Étape

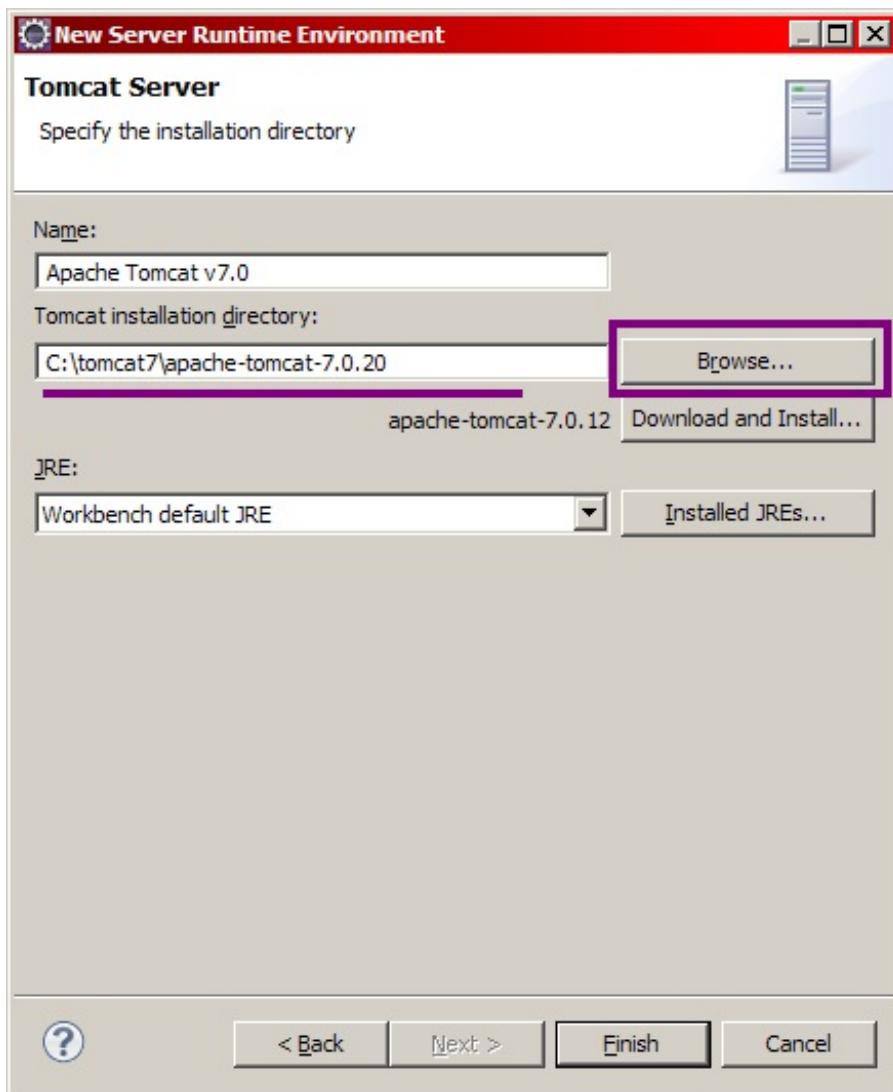
1

Cliquez sur le bouton New Runtime... et sélectionnez alors Apache Tomcat 7.0 dans la liste des possibilités, comme indiqué à la figure suivante.



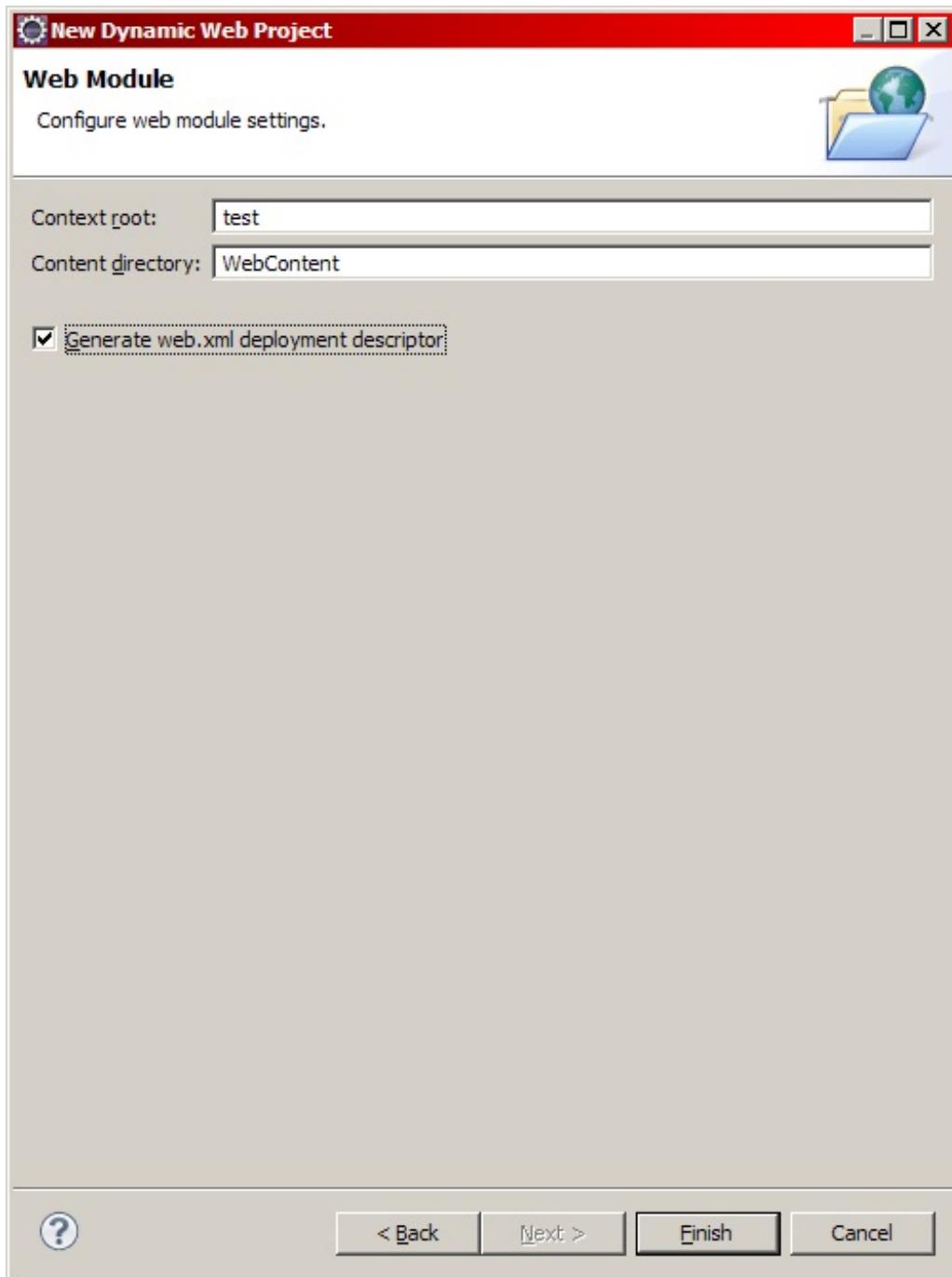
Mise en place de Tomcat - Étape 2

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 ceux de la figure suivante, 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 (voir la figure suivante).



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...

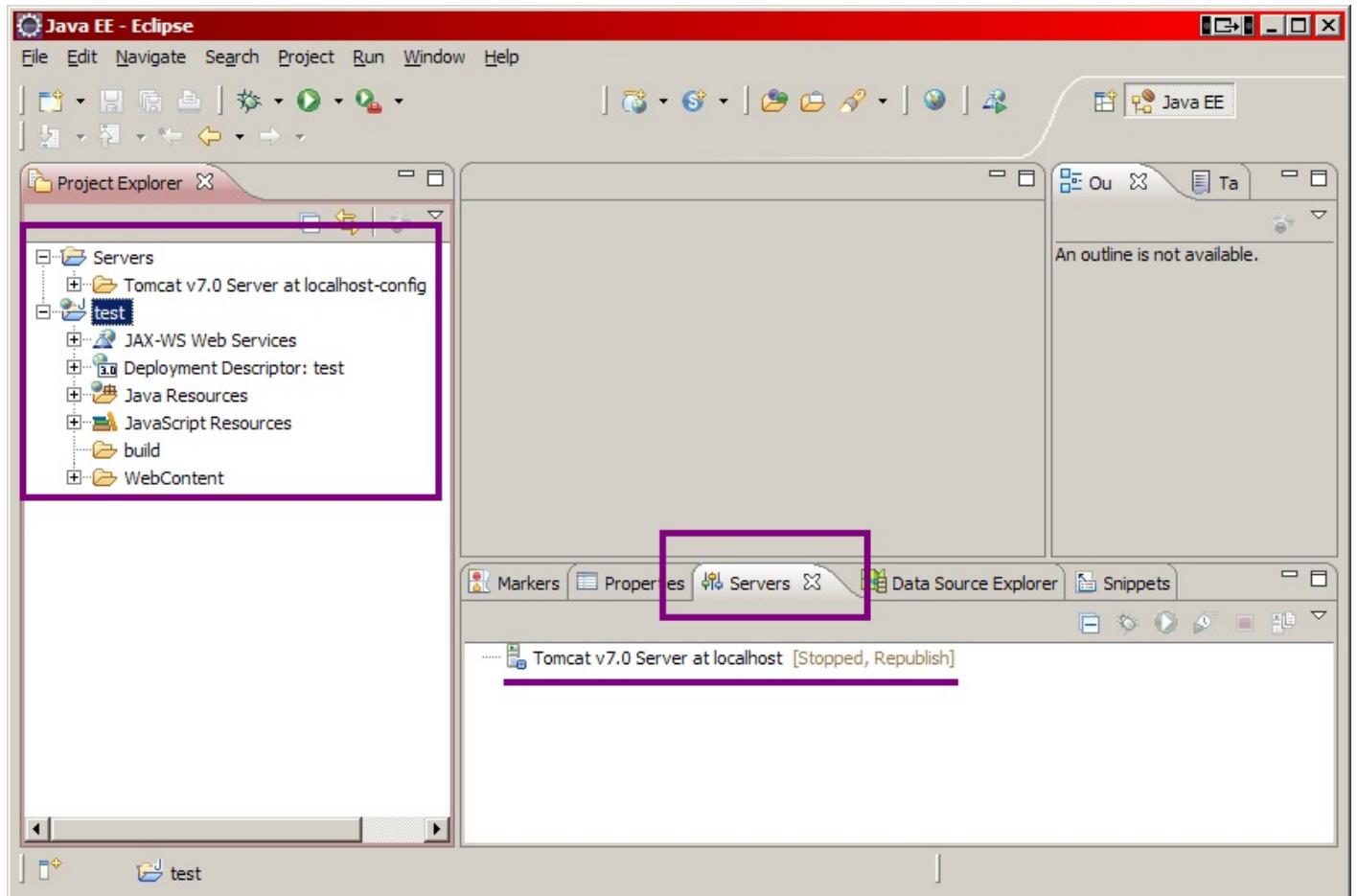
Le souci, et certains d'entre vous l'auront peut-être déjà compris, c'est que notre projet à nous, créé depuis Eclipse, se trouve dans un répertoire de notre *workspace* Eclipse : il n'est pas du tout dans ce fameux répertoire *webapps* de Tomcat. Pour que notre serveur prenne en compte notre future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

- créer un répertoire 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 donné 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 votre 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-mêmes 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?* 😊

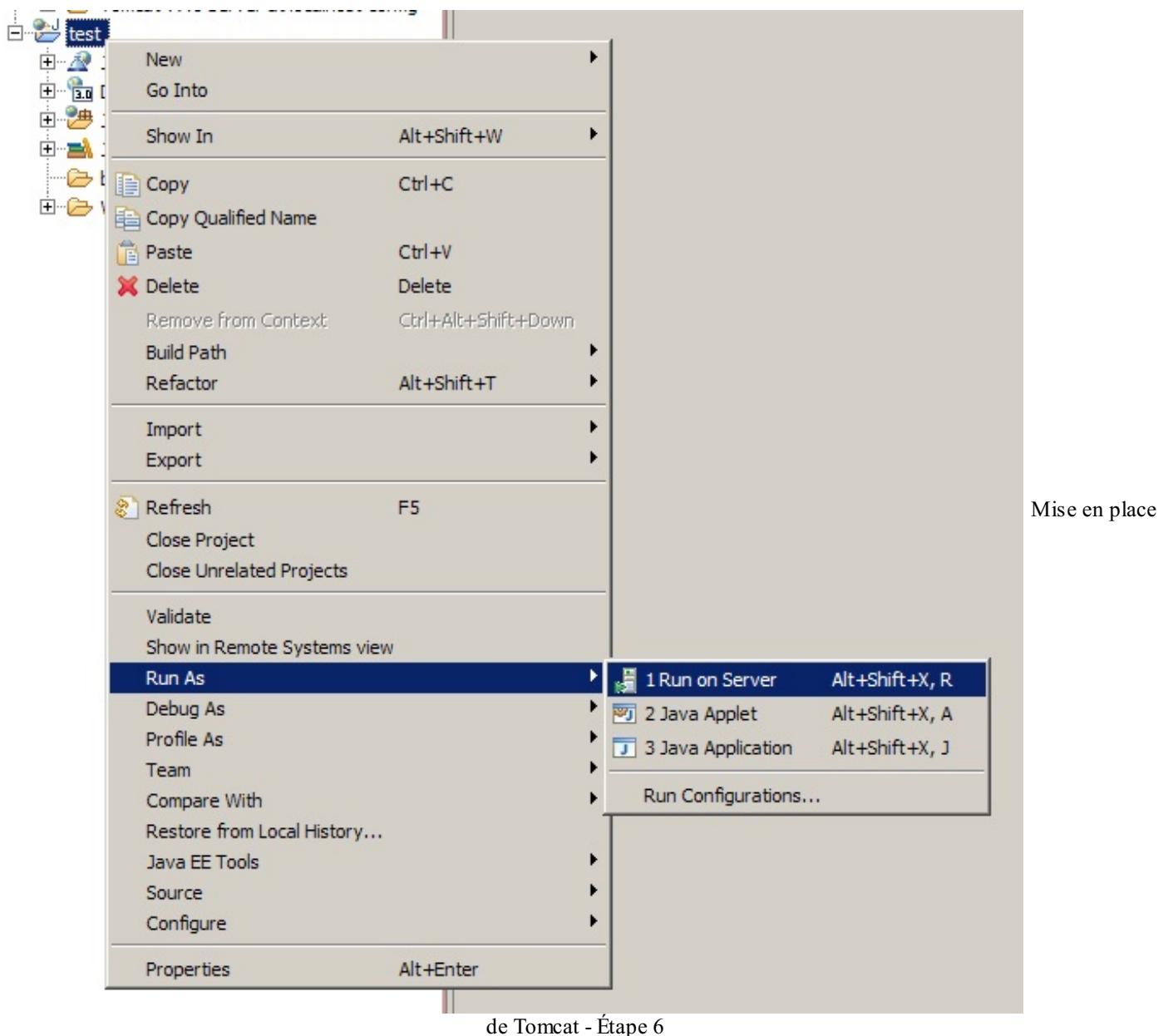
Voici maintenant à la figure suivante ce à quoi doit ressembler votre fenêtre Eclipse.



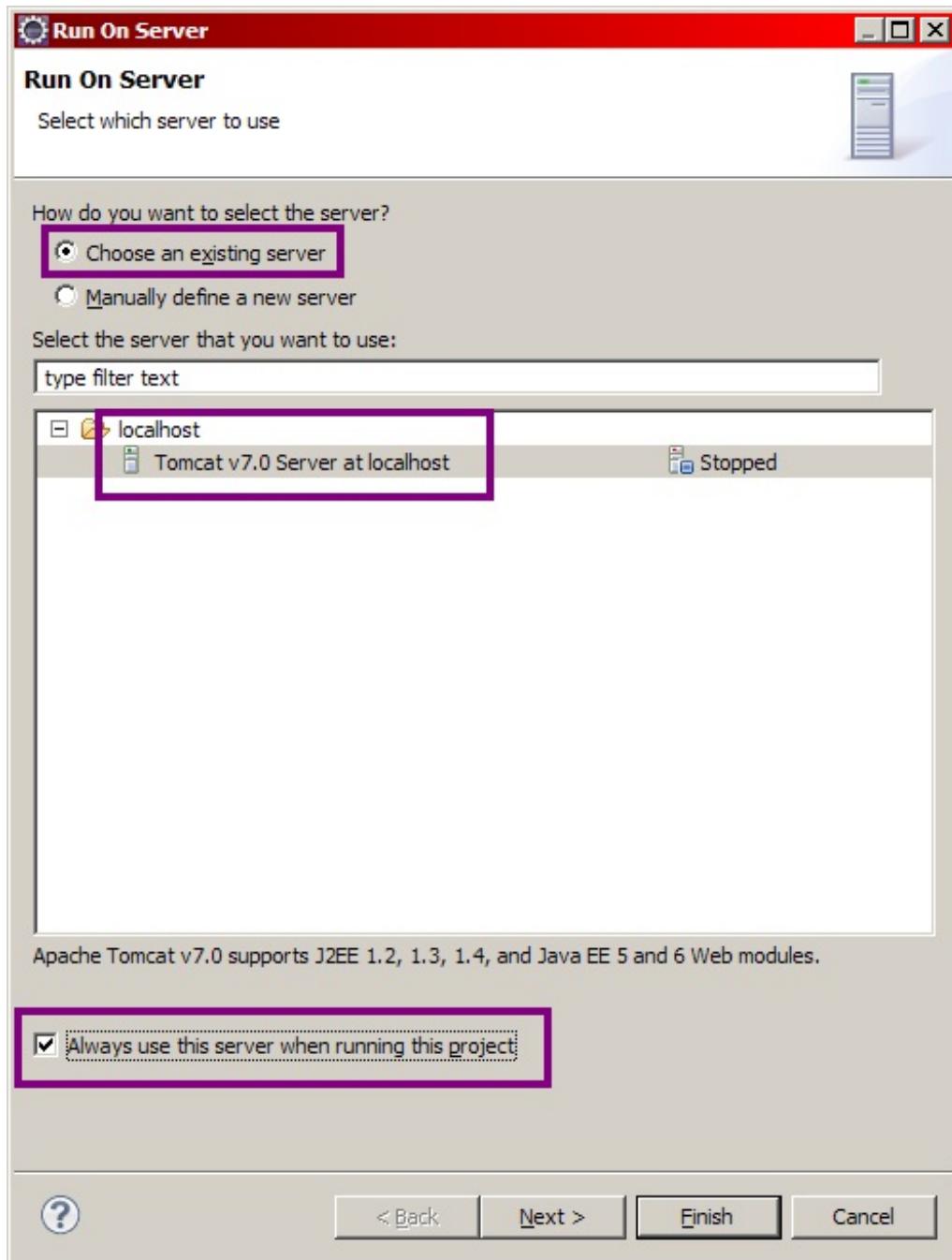
Mise en place de Tomcat - Étape 5

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*, comme indiqué à la figure suivante.



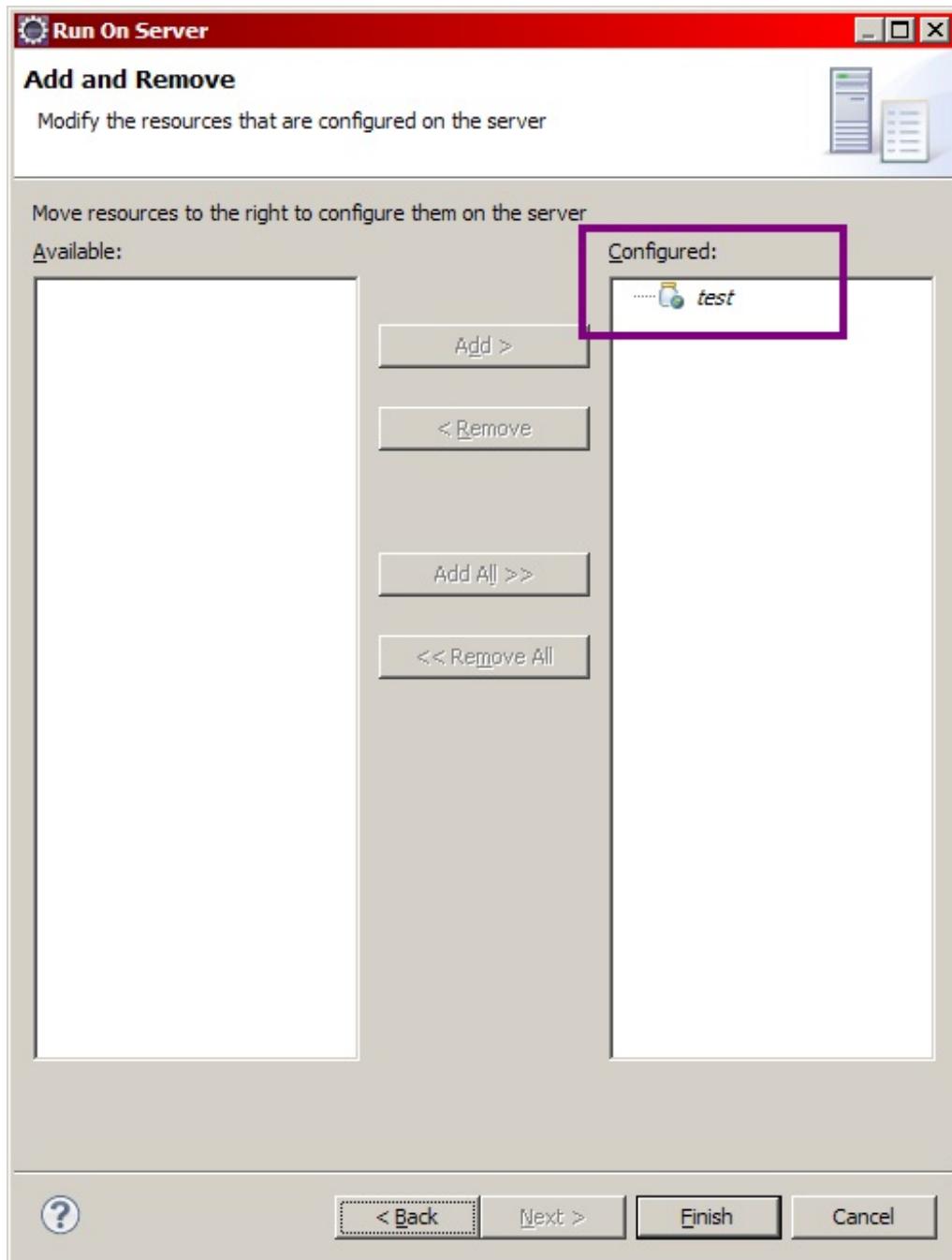
Dans la fenêtre qui s'ouvre alors (voir la figure suivante), nous allons sélectionner le serveur Tomcat que nous venons de mettre en place lors de la création de notre projet web, et préciser que l'on souhaite associer par défaut notre projet à ce serveur.



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 (voir la figure suivante).

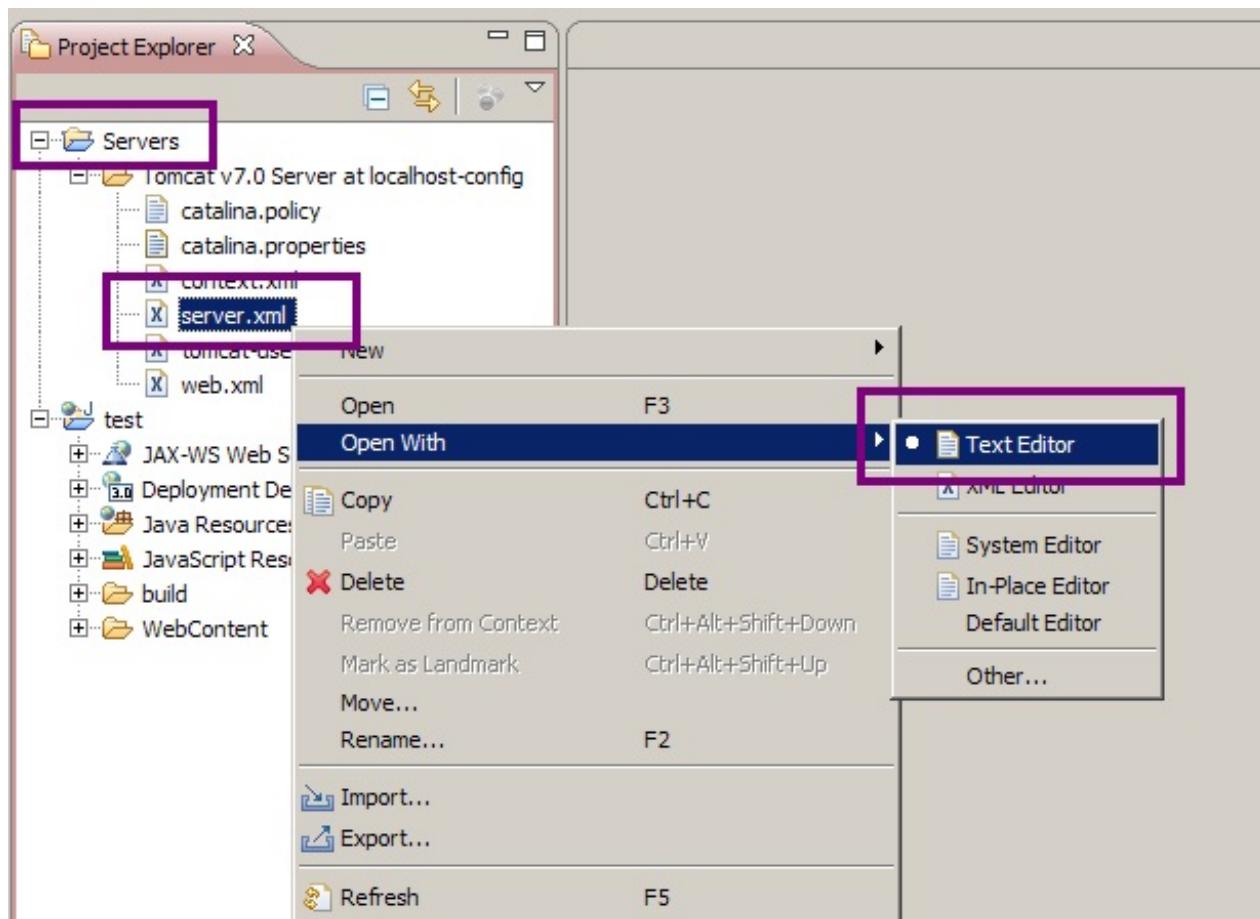


Mise en place de Tomcat - Étape

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, comme vous pouvez le voir sur la figure suivante.

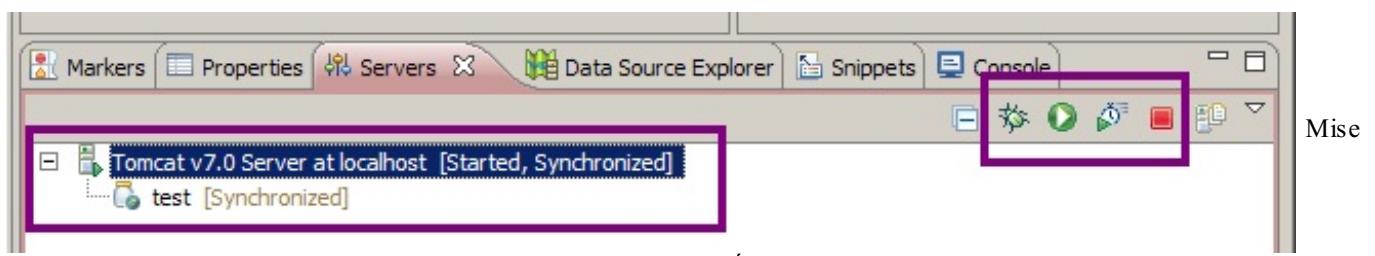


Si vous êtes curieux, éditez-le ! Vous verrez qu'il contient effectivement en fin de fichier une section de ce type :

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), comme indiqué à la figure suivante.



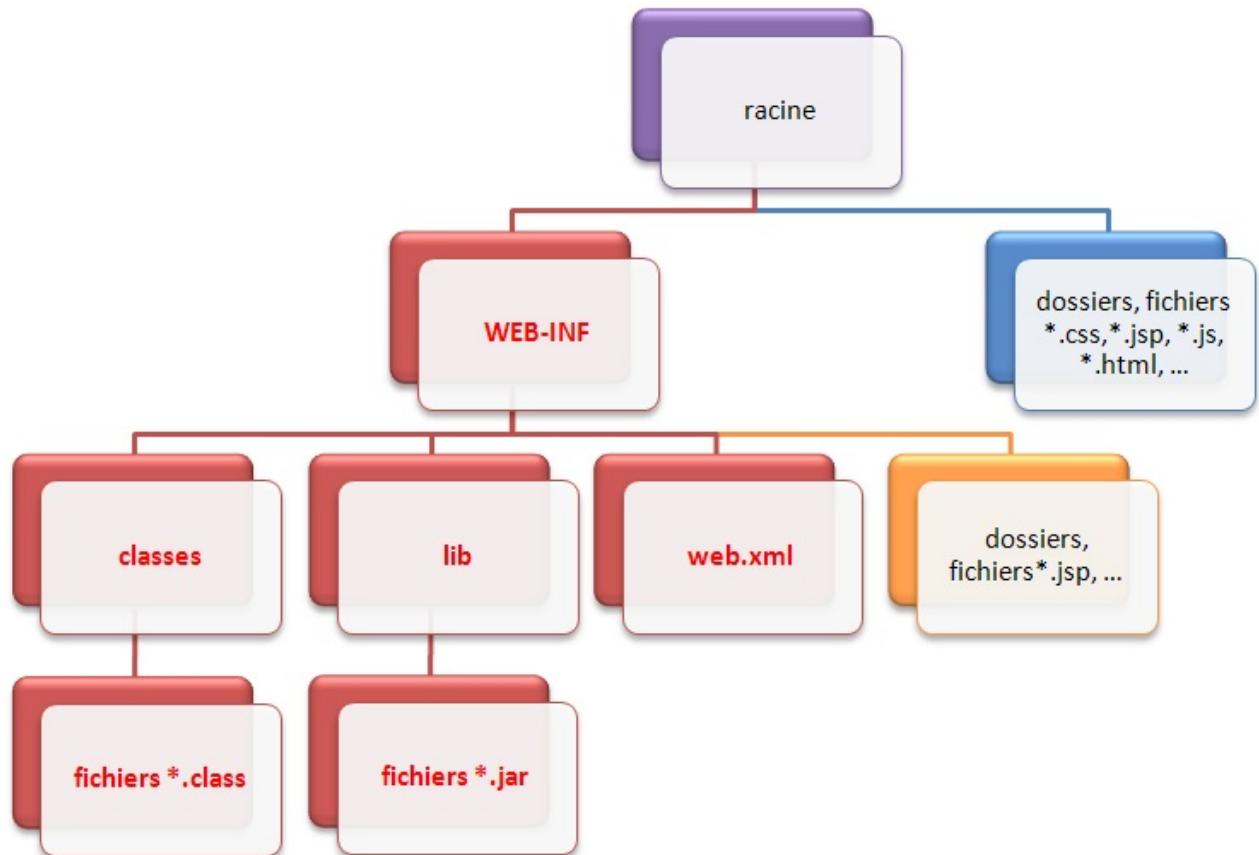
i Sachez pour finir, que cette manipulation n'est pas limitée à Tomcat. Vous pouvez utiliser d'autres types de serveurs, cela ne pose pas de problèmes. De même, une fois que vous avez correctement paramétré un serveur Tomcat depuis Eclipse, vous n'êtes pas forcés de recréer localement un nouveau serveur 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. Vous en trouverez le schéma à la figure suivante.



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 perso 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 perso 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.

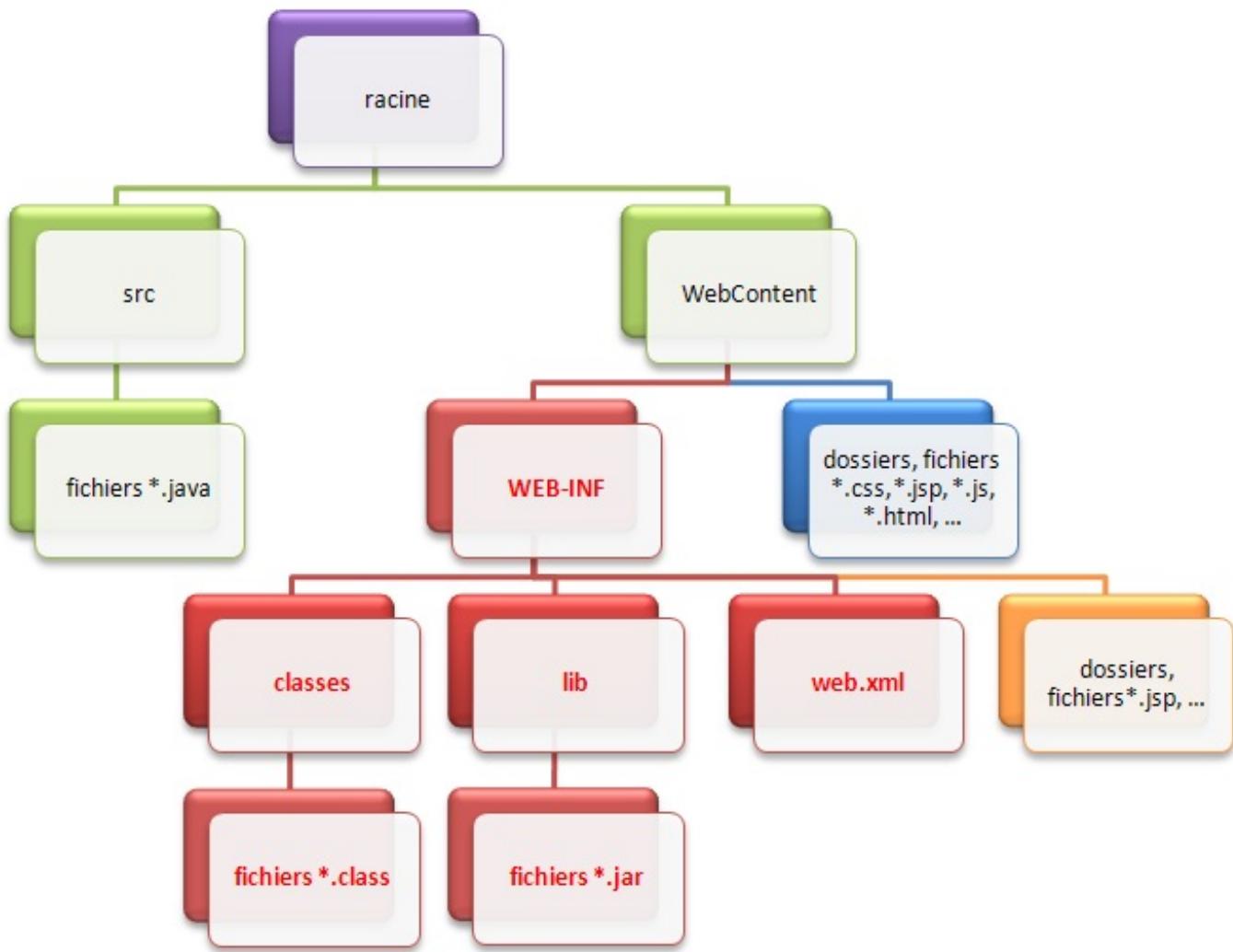


Voilà tout concernant la structure officielle : si votre application n'est pas organisée de cette manière, le serveur d'applications ne sera pas capable de la déployer ni de la faire fonctionner correctement.

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 (voir la figure suivante).



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 vers 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), sachez qu'Eclipse ajoute également sous la racine quelques fichiers de configuration qui lui permettront, via une tambouille interne, de gérer correctement l'application !



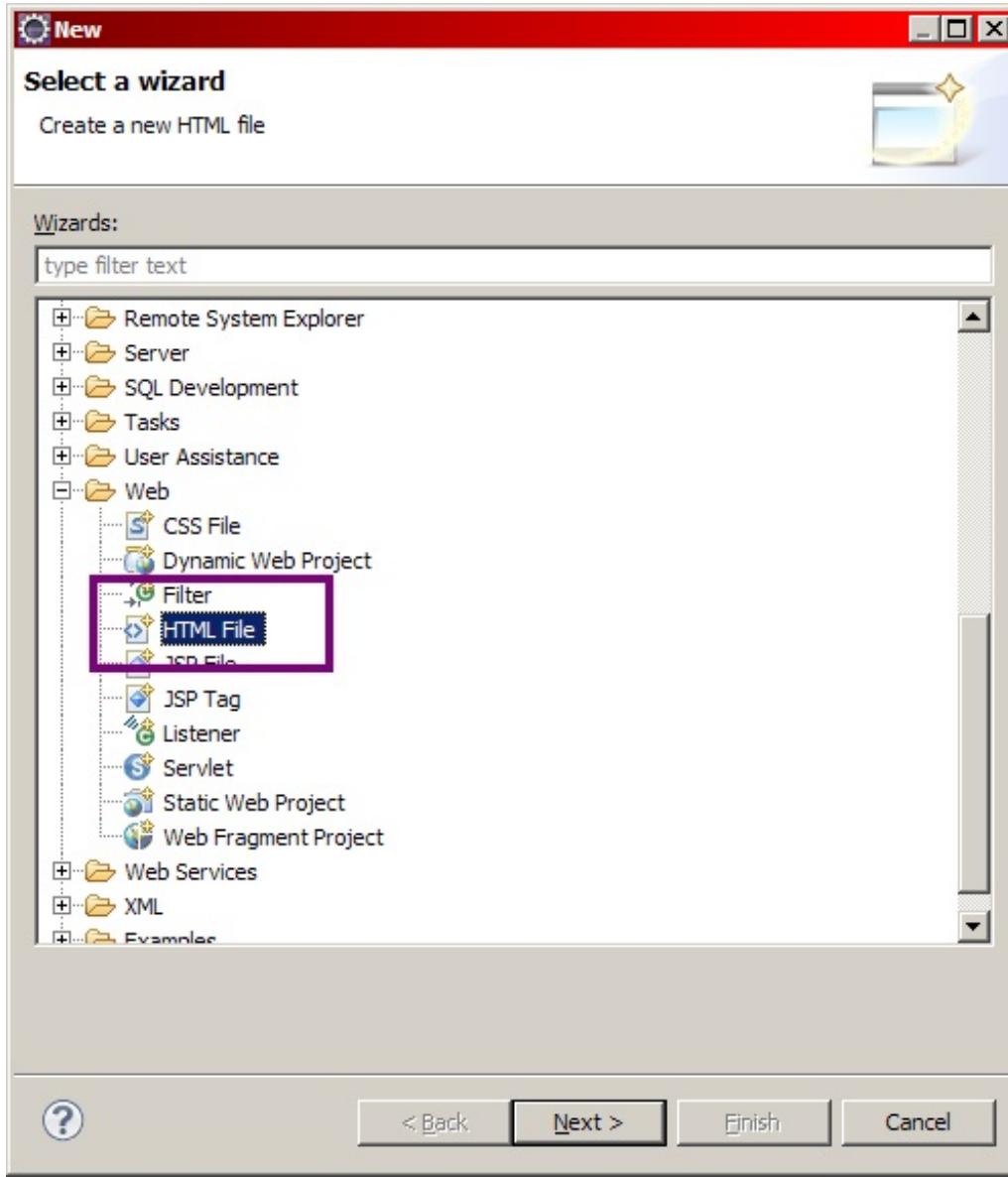
Attendez... Je viens de vous dire que si notre application n'était pas correctement structurée, notre serveur d'applications ne saurait pas la gérer. Si Eclipse vient mettre son nez dans cette histoire, comment notre application va-t-elle pouvoir fonctionner ?

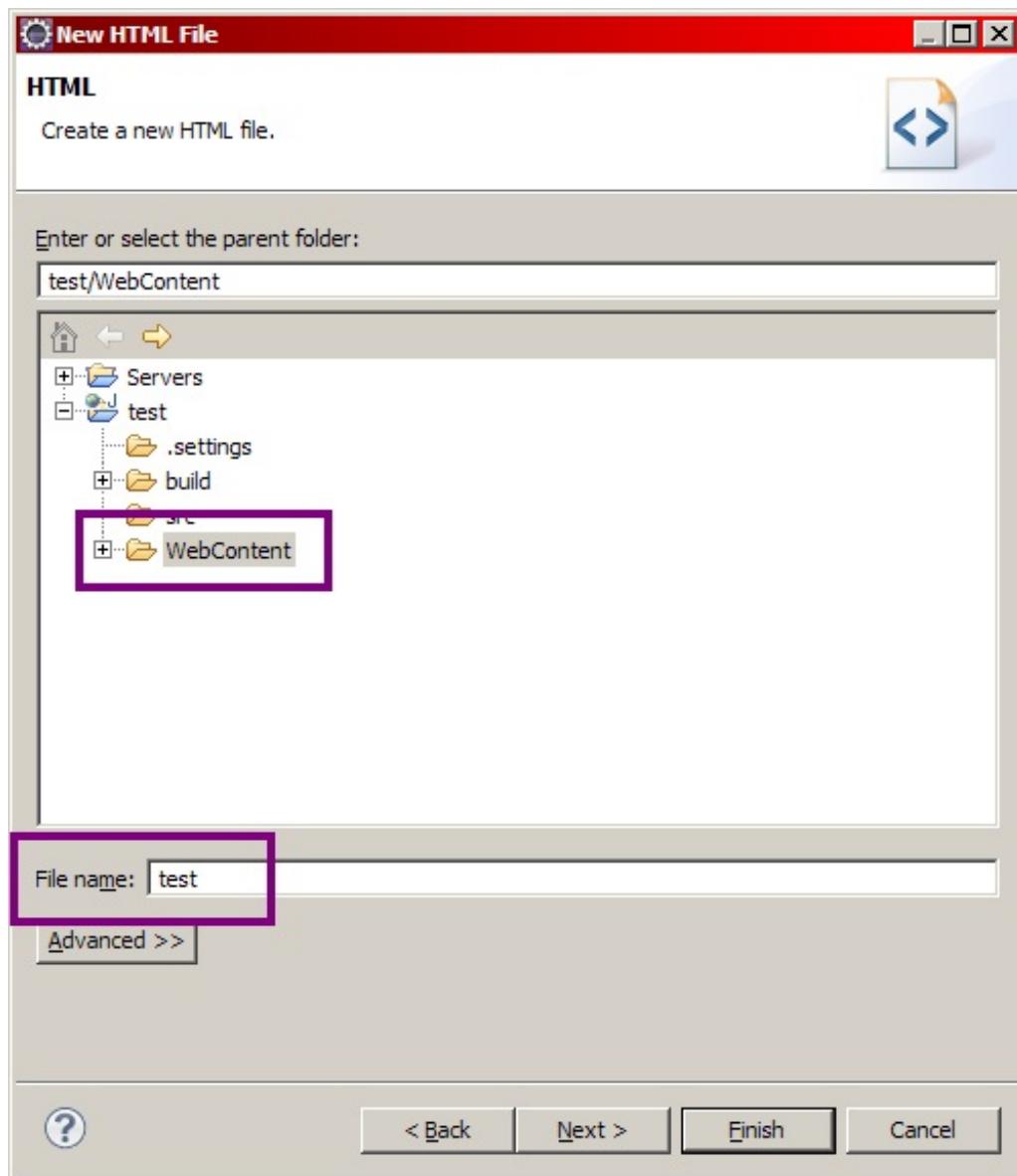
Eh bien comme je viens de vous l'annoncer, Eclipse se débrouille via une tambouille interne pour que la structure qu'il a modifiée soit, malgré tout, utilisable sur le serveur d'applications que nous lui avons intégré. Ceci implique donc deux choses très importantes :

- 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 copier-coller 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 le dossier **WebContent** de votre projet, puis donnez un nom à votre page et enfin validez. Je nomme ici ma page *test.html* (voir les figures suivantes).





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 si vous avez bien suivi les instructions que je vous ai présentées 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 flan pour afficher trois mots ?

Patience, patience... Notre serveur étant maintenant fonctionnel, nous voici prêts à entrer dans le vif du sujet.

- Un IDE permet de simplifier le développement d'un projet dans son ensemble.
- Tomcat n'est pas un serveur d'applications Java EE au sens complet du terme.
- La configuration du serveur passe principalement par deux fichiers : server.xml et web.xml.
- Une application web Java EE doit respecter une architecture bien définie.
- Eclipse modifie l'architecture des applications pour les intégrer correctement à son système.

Nous sommes maintenant prêts pour développer notre première application web. Allons-y !

Partie 2 : Premiers pas avec Java EE

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 détail, 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'adresses 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 *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 est que, comme la taille d'une URL est limitée, **on ne peut pas utiliser cette méthode pour envoyer des données volumineuses au serveur**, par exemple un fichier.

 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 de lire** des informations, sans que cela ait un quelconque impact sur la ressource demandée : ainsi, une requête GET est censée pouvoir être répétée indéfiniment sans risques pour la ressource concernée.

POST

La taille du corps du message d'une requête POST n'est pas limitée, c'est donc cette méthode qu'il faut utiliser pour soumettre au serveur des données de tailles variables, ou que l'on sait volumineuses. C'est 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 rafraîchissement de la page entraînera 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 que 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...

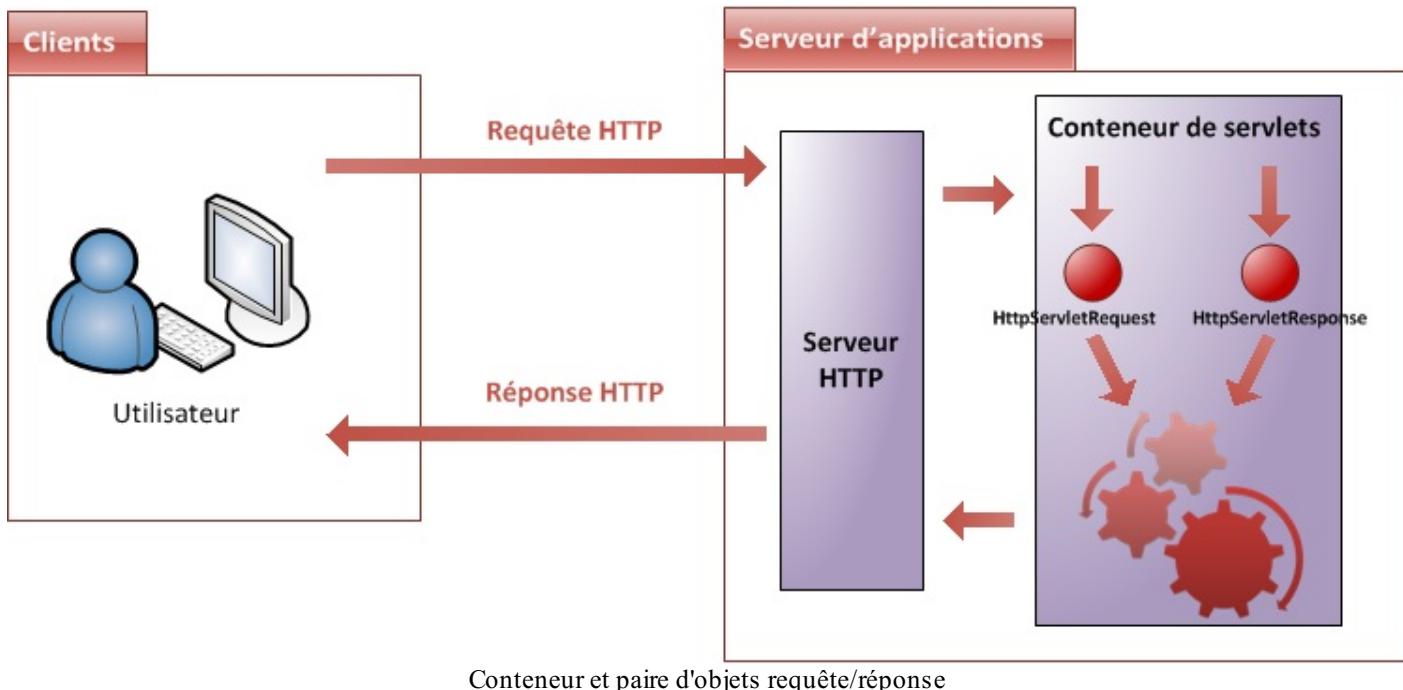
Rappelez-vous notre schéma global : la requête HTTP part du client et arrive sur le serveur. L'élément qui entre en jeu est alors le **serveur HTTP** (on parle également de **serveur web**), qui ne fait qu'écouter les requêtes HTTP sur un certain port, en général le port 80.



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** (voir la figure suivante). 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 (voir la figure suivante).

javax.servlet Interface Servlet

All Known Subinterfaces:

[HttpJspPage](#), [JspPage](#)

All Known Implementing Classes:

[FacesServlet](#), [GenericServlet](#), [HttpServlet](#)

[Javadoc de](#)

public interface Servlet

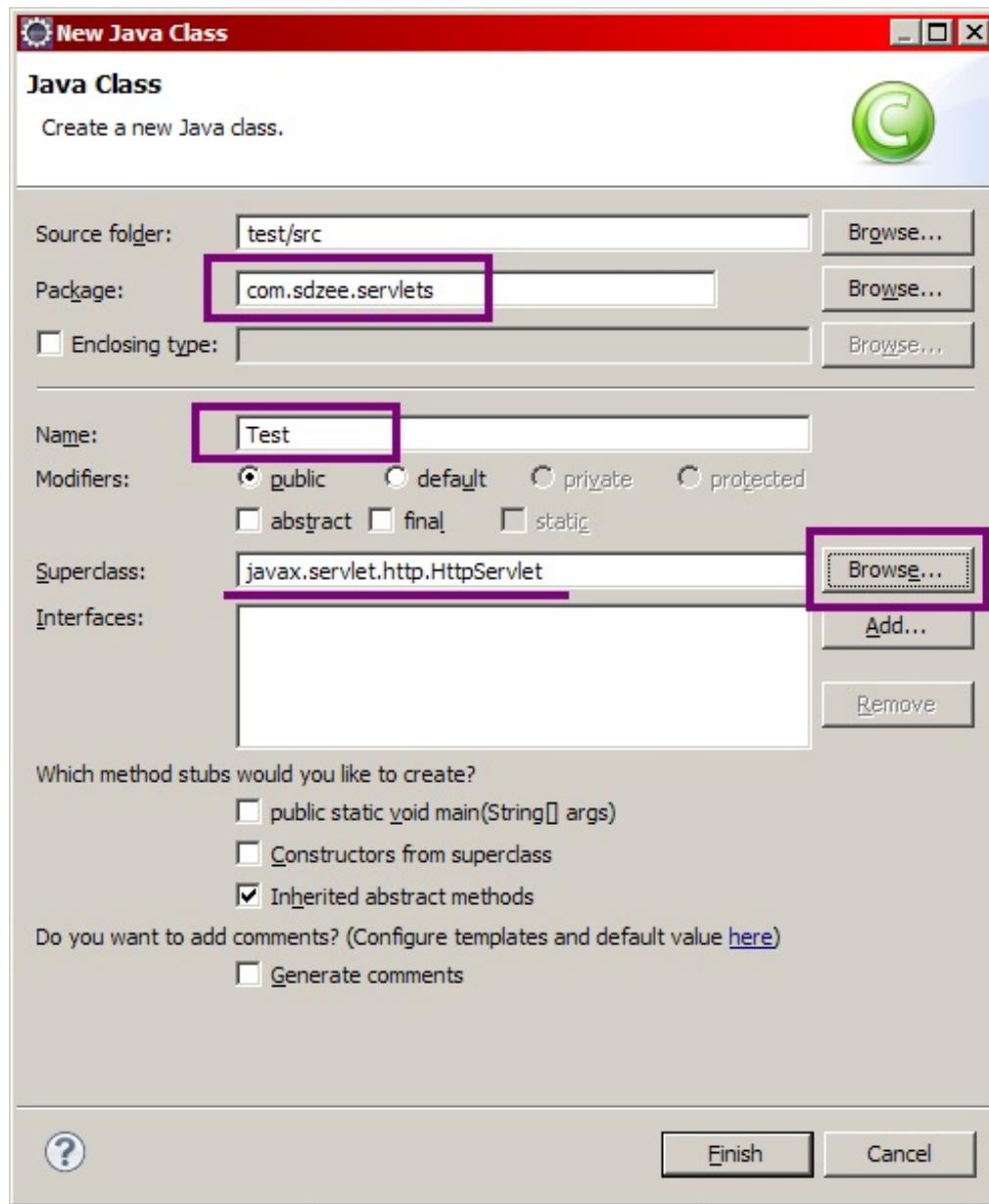
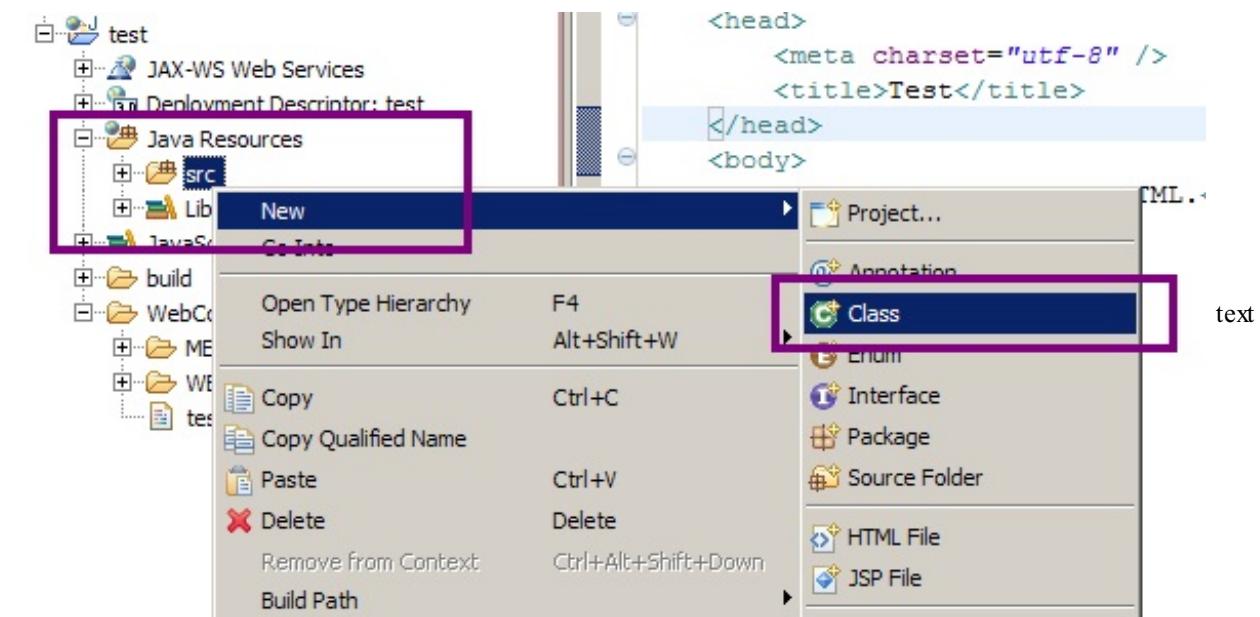
Defines methods that all servlets must implement.

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

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 indiqué sur les figures suivantes.



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 pas en Java EE 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>
    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">

</web-app>
```

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



Pour le moment, ne prenez pas attention aux nombreux attributs présents au sein de cette balise **<web-app>**, nous reviendrons sur leur rôle lorsque nous découvrirons les expressions EL.

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 des 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 détail 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 d'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 assurez-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>** sont basés 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 <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
    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">
    <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**, comme indiqué à la figure suivante.



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 <http://localhost:8080/test/toto> que nous venons de mapper sur notre servlet (voir la figure suivante).

Etat HTTP 405 - La méthode HTTP GET n'est pas supportée par cette URL

type Rapport d'état

message La méthode HTTP GET n'est pas supportée par cette URL

description La méthode HTTP spécifiée n'est pas autorisée pour la ressource demandée (La méthode HTTP GET n'est pas supportée par cette URL).

Apache Tomcat/7.0.20

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 la 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... (voir la figure suivante). 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).

The screenshot shows the Firebug Network tab with a single entry for a 'GET toto' request. The status is '200 OK' and the domain is 'localhost:8080'. The 'Réponse' (Response) section is highlighted with a purple border and contains the following text:

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Length: 0
Date: Tue, 17 Apr 2012 03:45:31 GMT
```

The 'Requête' (Request) section shows the following headers:

```
GET /test/toto HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0 (Windows NT 5.1; rv:11.0) Gecko/20100101 Firefox/11.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: fr,fr-fr;q=0.8,en-us;q=0.6,en;q=0.4,zh-cn;q=0.2
Accept-Encoding: gzip, deflate
Connection: keep-alive
Cache-Control: max-age=0
```

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 aucun 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 leverons 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-elle 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 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");
    response.setCharacterEncoding( "UTF-8" );
    PrintWriter out = response.getWriter();
    out.println("<!DOCTYPE html>");
    out.println("<html>");
    out.println("<head>");
    out.println("<meta charset=\"utf-8\" />");
    out.println("<title>Test</title>");
    out.println("</head>");
    out.println("<body>");
    out.println("<p>Ceci est une page générée depuis une servlet.</p>");
    out.println("</body>");
    out.println("</html>");
}
```

Comment procérons-nous ?

1. Nous commençons par modifier l'en-tête **Content-Type** de la réponse HTTP, pour préciser au client que nous allons lui envoyer une page HTML, en faisant appel à la méthode `setContentType()` de l'objet `HttpServletResponse`.
2. Par défaut, l'encodage de la réponse envoyée au client est initialisé à **ISO-8859-1**. Si vous faites quelques recherches au sujet de cet encodage, vous apprendrez qu'il permet de gérer sans problème les caractères de notre alphabet, mais qu'il ne permet pas de manipuler les caractères asiatiques, les alphabets arabes, cyrilliques, scandinaves ainsi que d'autres caractères plus exotiques. Afin de permettre une gestion globale d'un maximum de caractères différents, il est recommandé d'utiliser l'encodage **UTF-8** à la place. Voilà pourquoi nous modifions l'encodage par défaut en réalisant un appel à la méthode `setCharacterEncoding()` de l'objet `HttpServletResponse`. Par ailleurs, c'est également pour cette raison que je vous ai fait modifier les encodages par défaut lors de la configuration d'Eclipse !



Si vous regardez la documentation de cette méthode, vous découvrirez qu'il est également possible de s'en passer et d'initialiser l'encodage de la réponse directement via un appel à la méthode `setContentType("text/html; charset=UTF-8")`.

3. 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. Cet objet utilise l'encodage que nous avons défini précédemment, c'est-à-dire UTF-8.
4. Nous écrivons alors du texte dans la réponse via la méthode `println()` de l'objet `PrintWriter`.

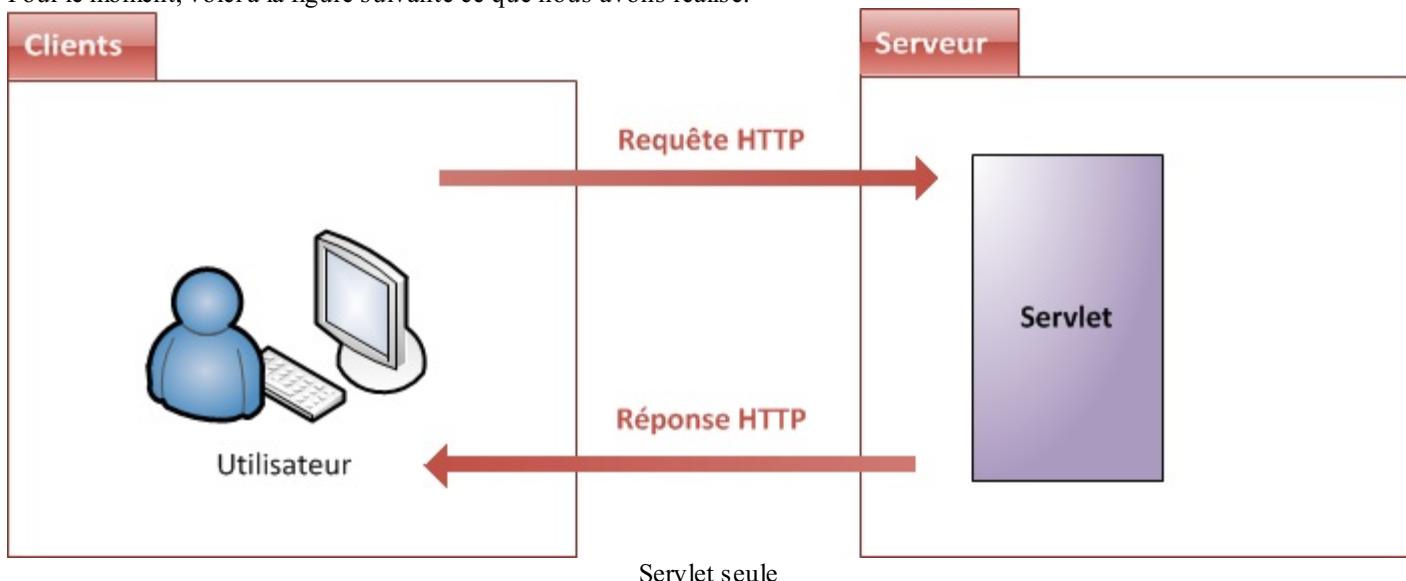
Enregistrez, testez et vous verrez enfin la page s'afficher dans votre navigateur : ça y est, vous savez maintenant utiliser une servlet et transmettre des données au client. 😊



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 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 procéderons 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 à la figure suivante ce que nous avons réalisé.



Note : dorénavant et afin d'alléger les 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.

- Le client envoie des requêtes au serveur grâce aux méthodes du protocole HTTP, notamment GET, POST et HEAD.
- Le conteneur web place chaque requête reçue dans un objet `HttpServletRequest`, et place chaque réponse qu'il initialise dans l'objet `HttpServletResponse`.
- Le conteneur transmet chaque couple requête/réponse à une servlet : c'est un objet Java assigné à une requête et capable de générer une réponse en conséquence.
- La servlet est donc le point d'entrée d'une application web, et se déclare dans son fichier de configuration `web.xml`.
- Une servlet peut se charger de répondre à une requête en particulier, ou à un groupe entier de requêtes.
- Pour pouvoir traiter une requête HTTP de type GET, une servlet doit implémenter la méthode `doGet()` ; pour répondre à une requête de type POST, la méthode `doPost()` ; etc.
- Une servlet n'est pas chargée de l'affichage des données, elle ne doit donc pas s'occuper de la présentation (HTML, CSS, etc.).

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 rapidement survolé dans la première partie de ce cours 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 pas pour le moment 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 contenus statiques**. Ceci est rendu possible par :

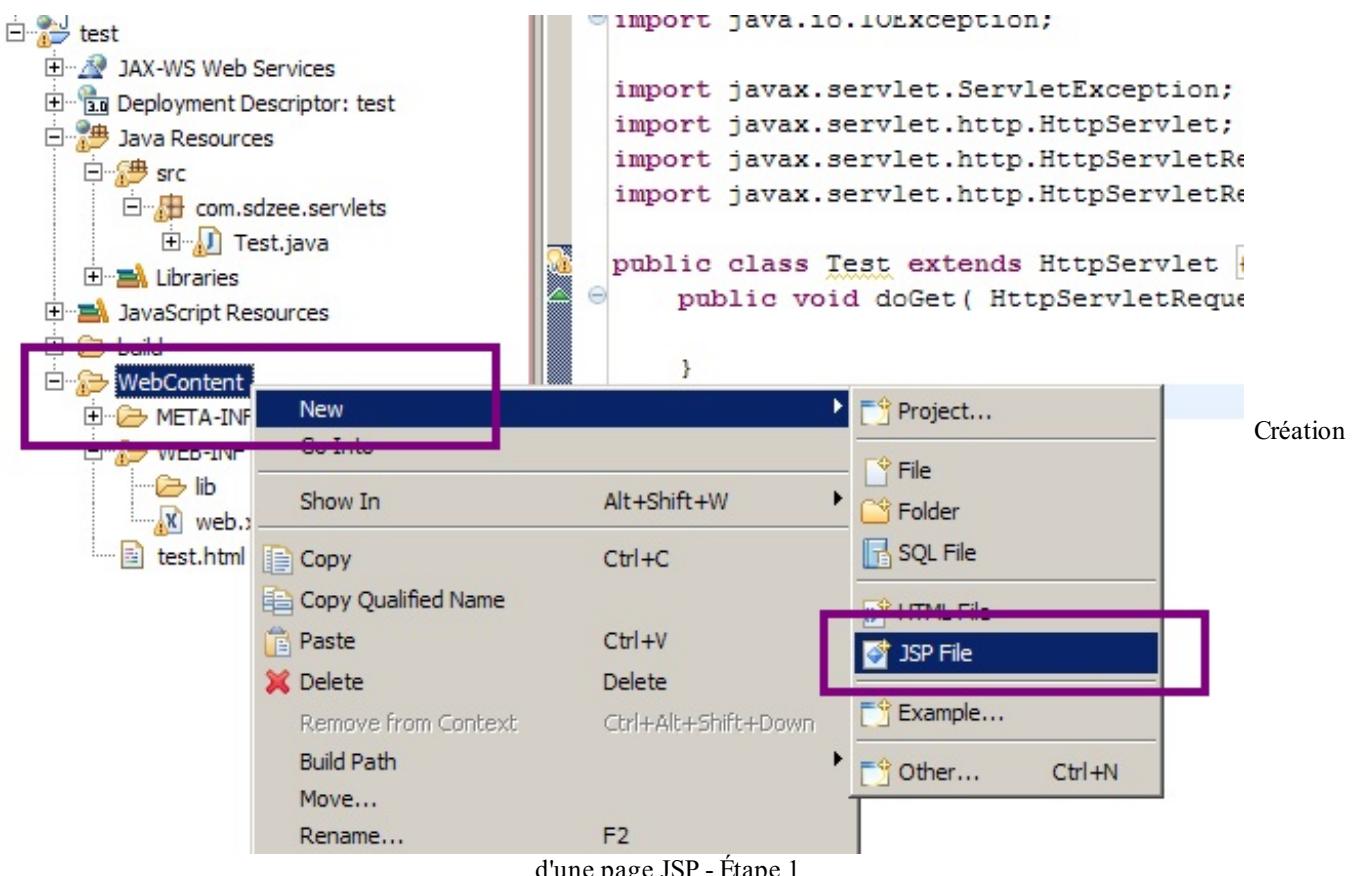
- **un langage dédié** : les pages JSP sont des documents au format texte, à l'opposé des classes Java que sont les servlets, qui décrivent indirectement comment traiter une requête et construire une réponse. Elles contiennent des balises qui combinent à la fois simplicité et puissance, via une syntaxe simple, semblable au HTML et donc aisément compréhensible 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 ! 😊

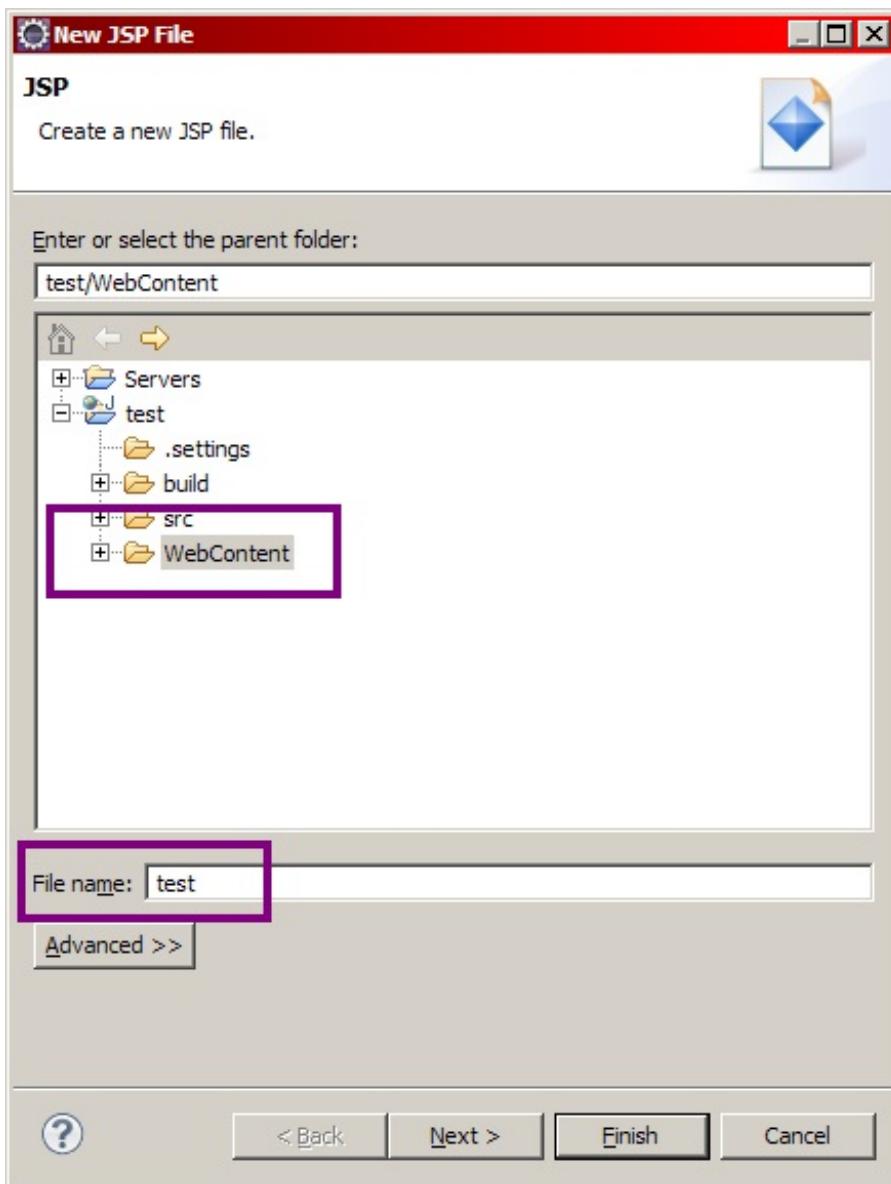
Bon, assez gambergé ! Maintenant que nous avons une bonne idée de ce que sont les pages JSP, rentrons dans le concret en étudiant leur vie au sein d'une application !

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, ainsi qu'indiqué aux figures suivantes.





Création d'une page JSP - Étape 2

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'adresses 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 résultat de la figure suivante.

Ceci est une page générée depuis une JSP.



Qu'est-ce que c'est que cette horreur ? Que s'est-il passé ?

Le problème que vous voyez ici, c'est l'encodage ! Eh oui, dans le chapitre précédent nous avions modifié l'encodage par défaut directement depuis notre servlet, et je ne vous avais alors pas expliqué pourquoi c'était vraiment nécessaire de procéder ainsi, je vous avais uniquement expliqué l'intérêt d'utiliser UTF-8. Cette fois dans notre JSP, nous n'avons rien modifié. Par conséquent, la réponse créée par notre page utilise la valeur par défaut, c'est-à-dire l'encodage latin ISO-8859-1.



Si l'encodage latin est utilisé par défaut, alors pourquoi les lettres accentuées ne s'affichent-elles pas correctement ? Ce sont bien des caractères de notre alphabet !

Nous y voilà, vous devez bien comprendre comment le principe d'encodage fonctionne. Il s'agit d'un processus en deux étapes, avec d'une part la manière dont sont encodés et gérés les caractères sur le serveur, et d'autre part la manière dont les données contenues dans la réponse envoyée vont être interprétées par le navigateur :

- en ce qui concerne le serveur, c'est simple : si vous avez bien suivi mes conseils lors de la configuration d'Eclipse, vos fichiers source sont tous encodés en UTF-8 ;
- en ce qui concerne le navigateur, celui-ci va uniquement se baser sur le contenu de l'en-tête **Content-type** de la réponse HTTP afin d'interpréter les données reçues.

Or, comme je viens de vous le dire, votre page JSP a par défaut envoyé la réponse au client en spécifiant l'encodage latin dans l'en-tête HTTP. Voilà donc l'explication de l'affreux micmac observé : votre navigateur a essayé d'afficher des caractères encodés en UTF-8 en utilisant l'encodage latin ISO-8859-1, et il se trouve que l'encodage des caractères accentués en ISO n'a rien à voir avec celui des caractères accentués en UTF ! D'où les symboles bizarroïdes observés...



Comment modifier l'en-tête HTTP depuis notre page JSP, afin de faire savoir au navigateur qu'il doit utiliser l'encodage UTF-8 pour interpréter les données reçues ?

Pour cela, il va falloir ajouter une instruction en tête de votre page JSP. Je ne vais pas vous l'expliquer dès maintenant, je reviendrai sur son fonctionnement dans un chapitre à venir. Contentez-vous simplement d'ajouter cette ligne tout en haut du code de votre page pour le moment :

Code : JSP

```
<%@ page pageEncoding="UTF-8" %>
```

Une fois la modification effectuée et enregistrée, actualisez la page dans votre navigateur et vous constaterez qu'il vous affiche maintenant correctement le texte attendu.

Tout va bien donc, notre JSP est 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, assurez-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 !

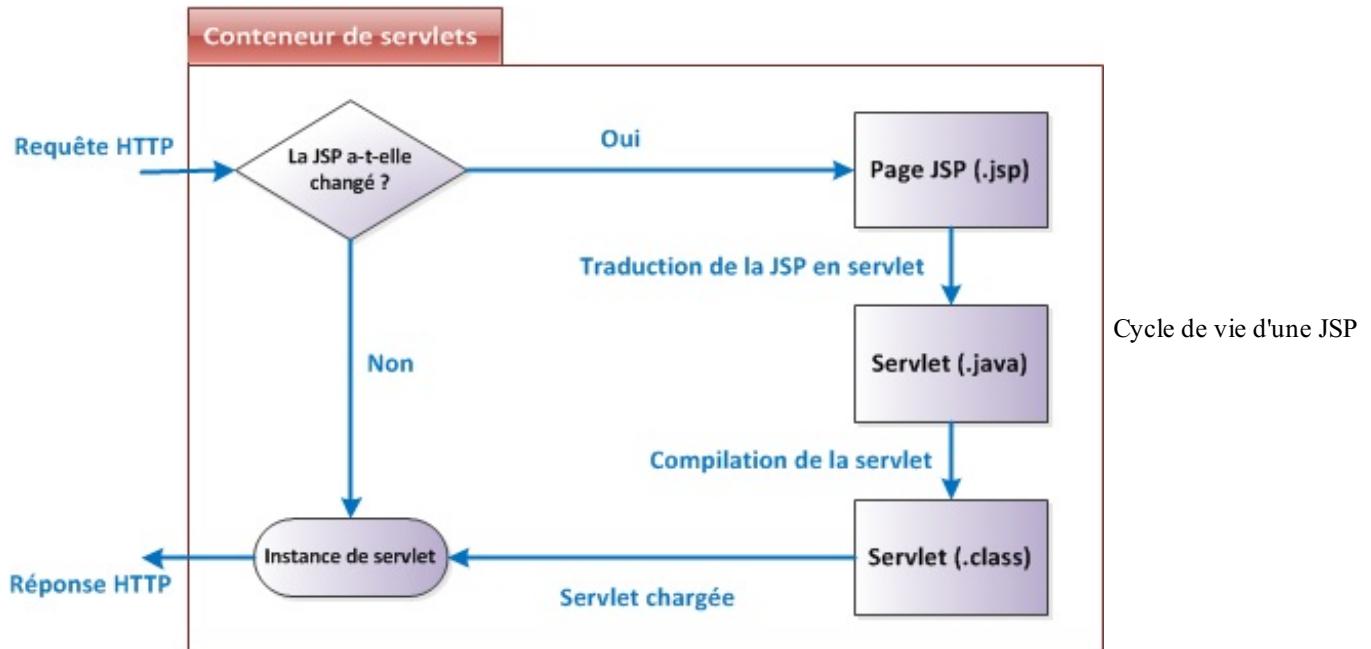


Que se passe-t-il si le contenu d'une page JSP est modifié ? Que devient la servlet auto-générée correspondante ?

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 est identique :
 - 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 sur la figure suivante, 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, à partir du moment où vous avez appelé au moins une fois cette JSP depuis votre navigateur ! 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 suffit 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 préoccupons 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 plus loin, et déplacer notre page JSP dans 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.

Reprendons 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, en 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 et 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.

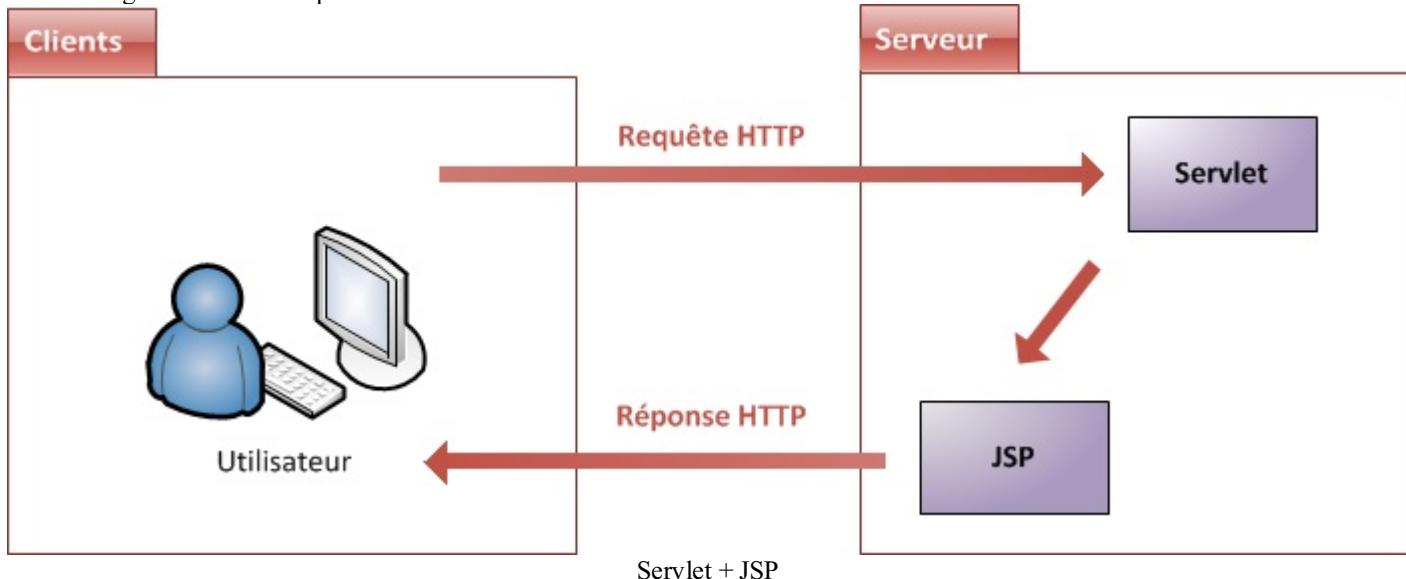
 Si vous parcourez la documentation de l'objet `HttpServletRequest`, vous remarquerez qu'il contient lui aussi une méthode `getRequestDispatcher()` ! Toutefois, cette dernière présente une différence notable : elle peut prendre en argument un chemin relatif, alors que sa grande sœur n'accepte qu'un chemin 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 la page !

Voici à la figure suivante ce que nous venons de réaliser.



- Une page JSP ressemble en apparence à une page HTML, mais en réalité elle est bien plus proche d'une servlet : elle contient des balises derrière lesquelles se cache du code Java.
- Une page JSP est exécutée sur le serveur, et la page finale générée et envoyée au client est une simple page HTML : le client ne voit pas le code de la JSP.
- Idéalement dans le modèle MVC, une page JSP est accessible à l'utilisateur à travers une servlet, et non pas directement.
- Le répertoire /WEB-INF cache les fichiers qu'il contient à l'extérieur de l'application.
- La méthode `forward()` de l'objet `RequestDispatcher` permet depuis une servlet de rediriger la paire requête/réponse HTTP vers une autre servlet ou vers une page JSP.

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 (la ligne 3). 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 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 nommé par la suite `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

```
<%@ page pageEncoding="UTF-8" %>
<!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 (lignes 11 à 14) 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 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 avertis 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 autorisera 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 réafficher 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 la ligne 2 : 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 lappelez 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 a 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

```

<%@ page pageEncoding="UTF-8" %>
<!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>

```

```
</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 à 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 `getParameter()` 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.

- Un attribut de requête est en réalité un objet stocké dans l'objet `HttpServletRequest`, et peut contenir n'importe quel type de données.
- Les attributs de requête sont utilisés pour permettre à une servlet de transmettre des données à d'autres servlets ou à des pages JSP.
- Un paramètre de requête est une chaîne de caractères placée par le client à la fin de l'URL de la requête HTTP.
- Les paramètres de requête sont utilisés pour permettre à un client de transmettre des données au serveur.

Le JavaBean

Ce court chapitre a pour unique objectif de vous présenter un type d'objet un peu particulier : le **JavaBean**. Souvent raccourci en "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

Pourquoi le JavaBean ?

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, et représente généralement des données du monde réel.

Voici un récapitulatif des principaux concepts mis en jeu. Je vous donne ici des définitions plutôt abstraites, mais il faut bien en 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 paramétriser 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 logiques dans une application**.

Un JavaBean n'est pas un EJB

Certains d'entre vous ont peut-être déjà entendu parler d'un composant Java EE nommé « EJB », signifiant *Enterprise JavaBean*. Si ce nom ressemble très fortement aux beans que nous étudions ici, ne tombez pas dans le piège et ne confondez pas les deux : les EJB suivent un concept complètement différent. Je ne m'attarde pas sur le sujet mais ne vous inquiétez pas, nous reviendrons sur ce que sont ces fameux EJB en temps voulu.

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 proprietNumero1;
    private int proprietNumero2;

    /* 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 proprietNumero1 ) {
        this.proprieteNumero1 = proprietNumero1;
    }

    public void setProprieteNumero2( int proprietNumero2 ) {
        this.proprieteNumero2 = proprietNumero2;
    }

    /* 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 pour le moment 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 !

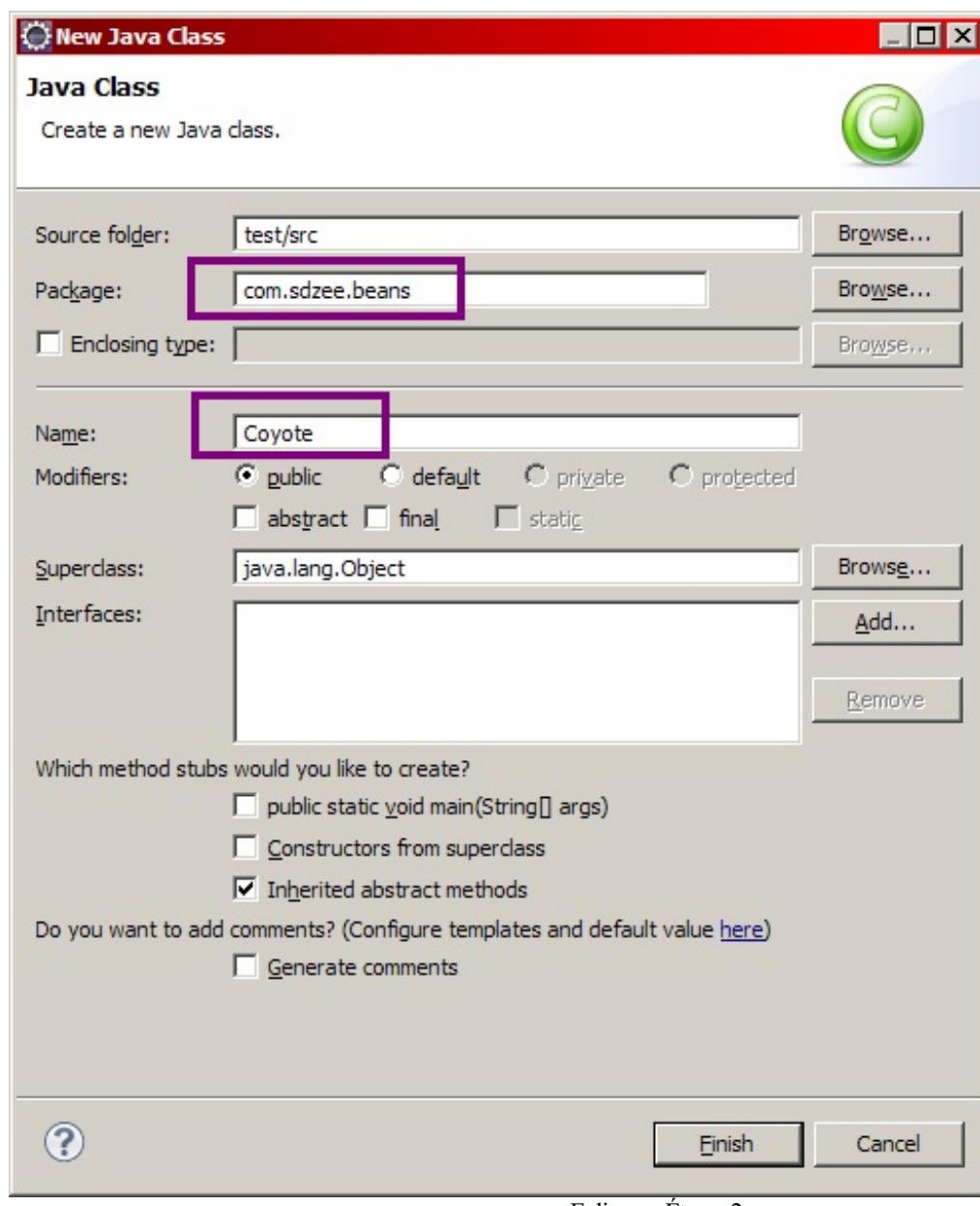
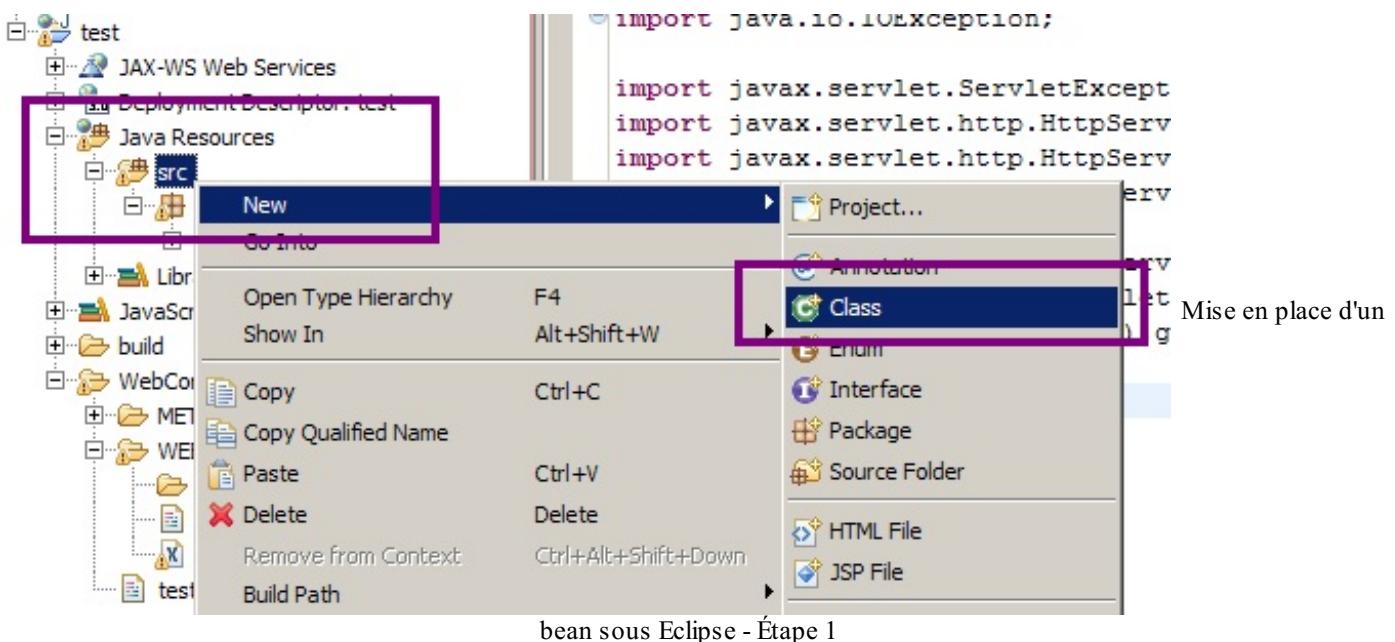
J'ai ici créé un bean contenant seulement trois propriétés, à savoir les trois champs non publics *nom*, *prenom* et *genius*. Inutile de s'attarder sur la nature des types utilisés ici, ceci n'est qu'un exemple totalement bidon qui ne sert à rien d'autre qu'à vous permettre de bien visualiser le concept.

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 figures suivantes pour visualiser la démarche sous Eclipse.



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 !

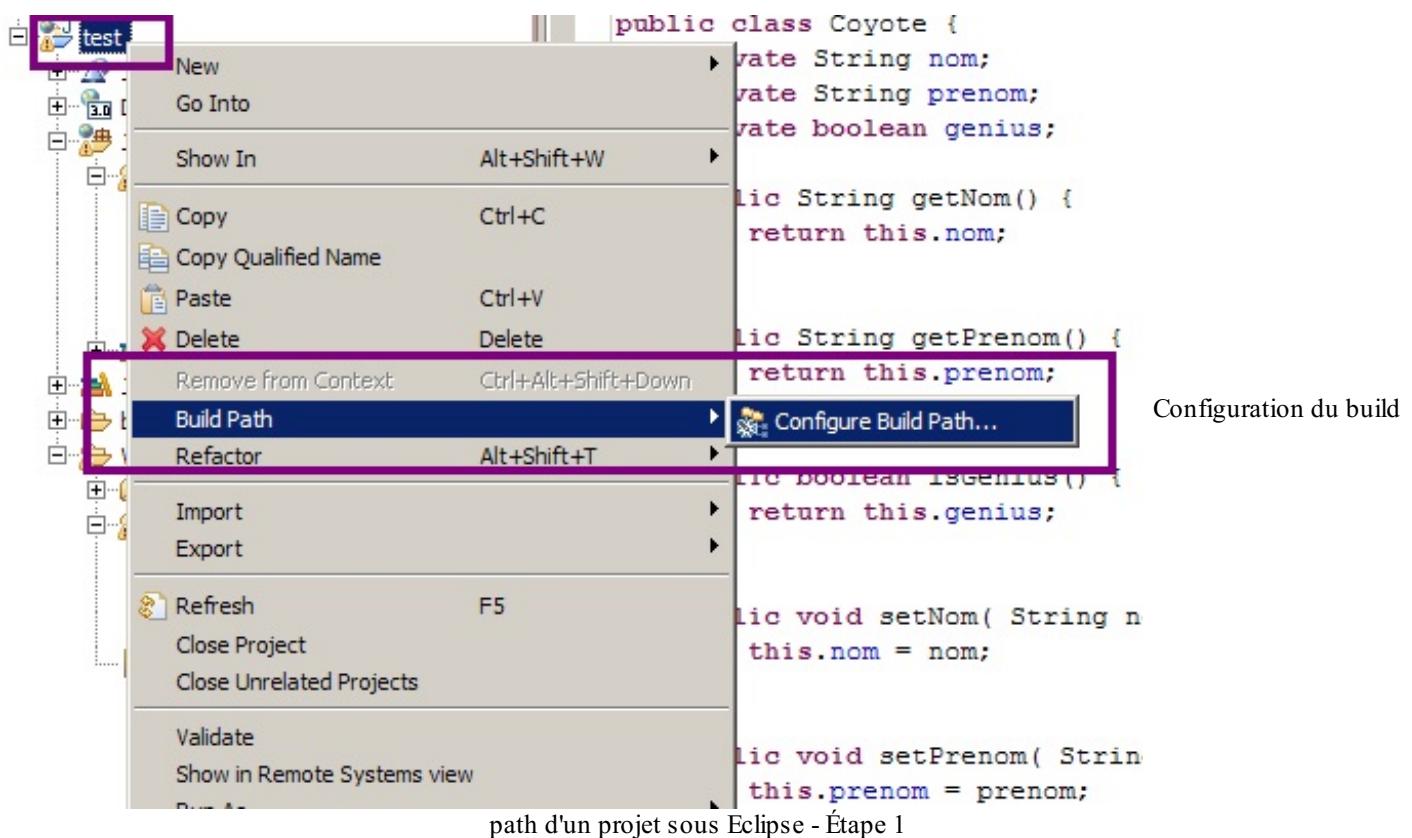
Configuration du projet sous Eclipse

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. Souvenez-vous, nous en avions déjà parlé dans le troisième chapitre de la première partie.

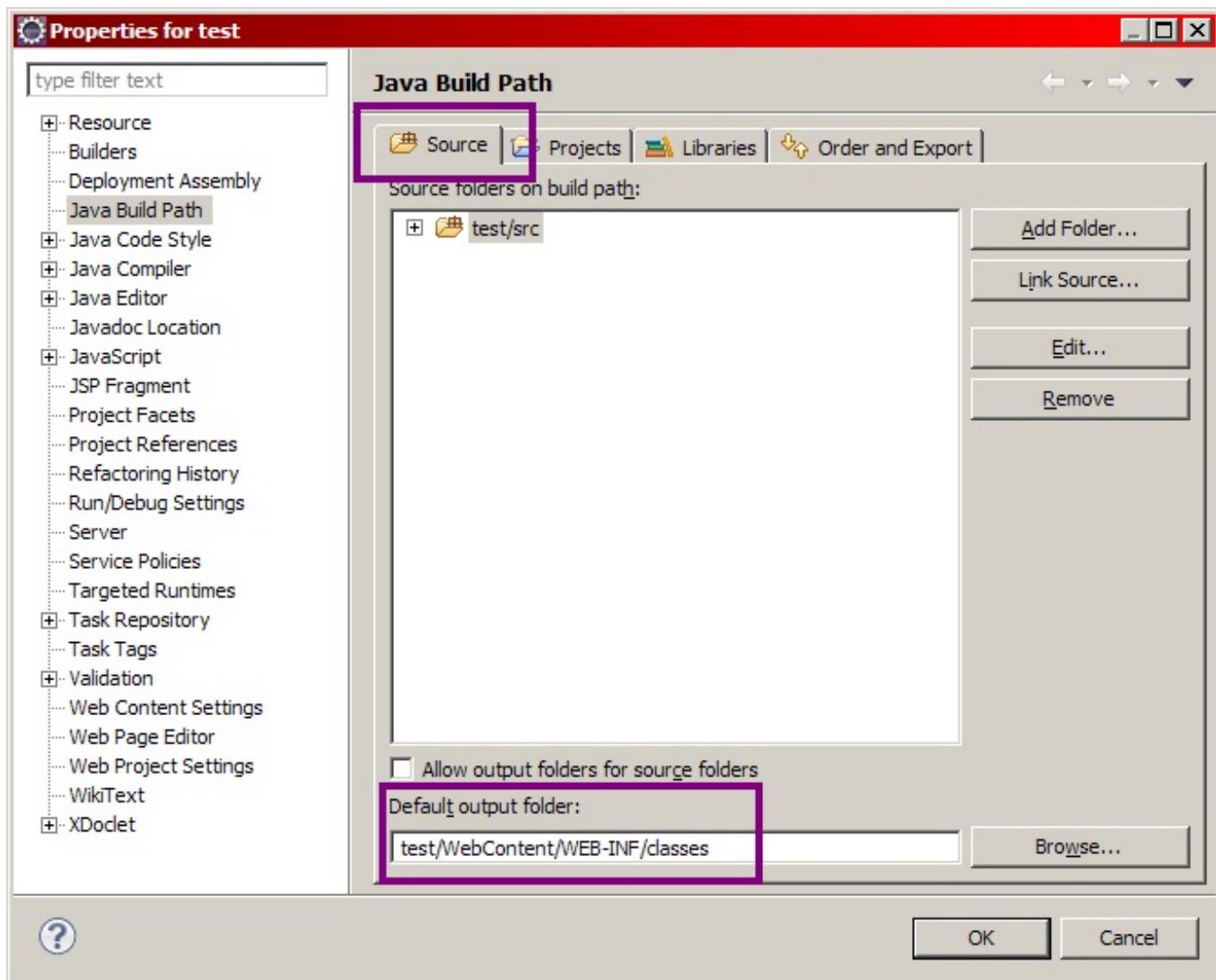
Par défaut Eclipse, toujours aussi fourbe, ne procède pas ainsi et envoie automatiquement vos classes compilées dans un dossier nommé "build".

Afin de changer ce 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...", comme indiqué à la figure suivante.



Sélectionnez alors l'onglet source, puis regardez en bas le champ Default output folder, comme sur la figure suivante.

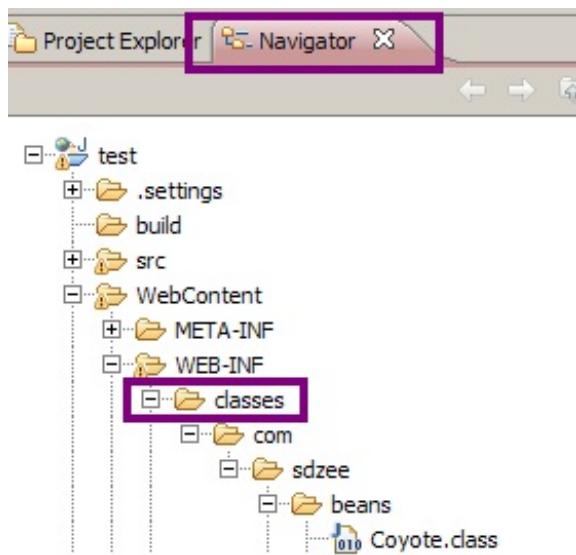


Configuration du build path d'un projet sous Eclipse - Étape 2

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 et vos JSP !

 Par défaut, Eclipse ne vous montre pas ce répertoire "classes" dans l'arborescence du 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**, comme indiqué à la figure suivante.



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

```

...
import com.sdzee.beans.Coyote;

...
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

```

<%@ page pageEncoding="UTF-8" %>
<!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 leurs 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 !** 😊

- Un bean est un objet Java réutilisable qui représente une entité, et dont les données sont représentées par des propriétés.
- Un bean est une classe publique et doit avoir au moins un constructeur par défaut, public et sans paramètres.
- Une propriété d'un bean est un champ non public, qui doit être accessible à travers un couple de getter/setter.
- Il faut configurer le build-path d'un projet web sous Eclipse pour qu'il y dépose automatiquement les classes compilées depuis les codes sources Java de vos objets.
- Un bean peut par exemple être transmis d'une servlet vers une page JSP (ou une autre servlet) en tant qu'attribut de requête.

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 préféré 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>
```

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 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 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 cette balise `page`, comme le `contentType` ou l'activation de la session. Toutes ont des valeurs par défaut, et je ne vais pas m'attarder sur les détails de chacune d'elles ici. Vous ne vous en servirez que dans des cas très spécifiques que nous découvrirons au cas par cas dans ce cours. Voici 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="..."
    pageEncoding="..."
    isErrorPage="true | false"
%>
```

Vous retrouvez ici celle que je vous ai fait utiliser depuis la mise en place de votre première JSP : le `pageEncoding`. C'est à travers cette option que vous pouvez spécifier l'encodage qui va être précisé dans l'en-tête de la réponse HTTP envoyée par votre page JSP.

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 clarifier, 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 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 l'inclusion ne se fera que lors de l'exécution. Ainsi fatallement, 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 les unes des autres et doivent pouvoir être compilées séparément. Ce 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'applications 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 ça 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.



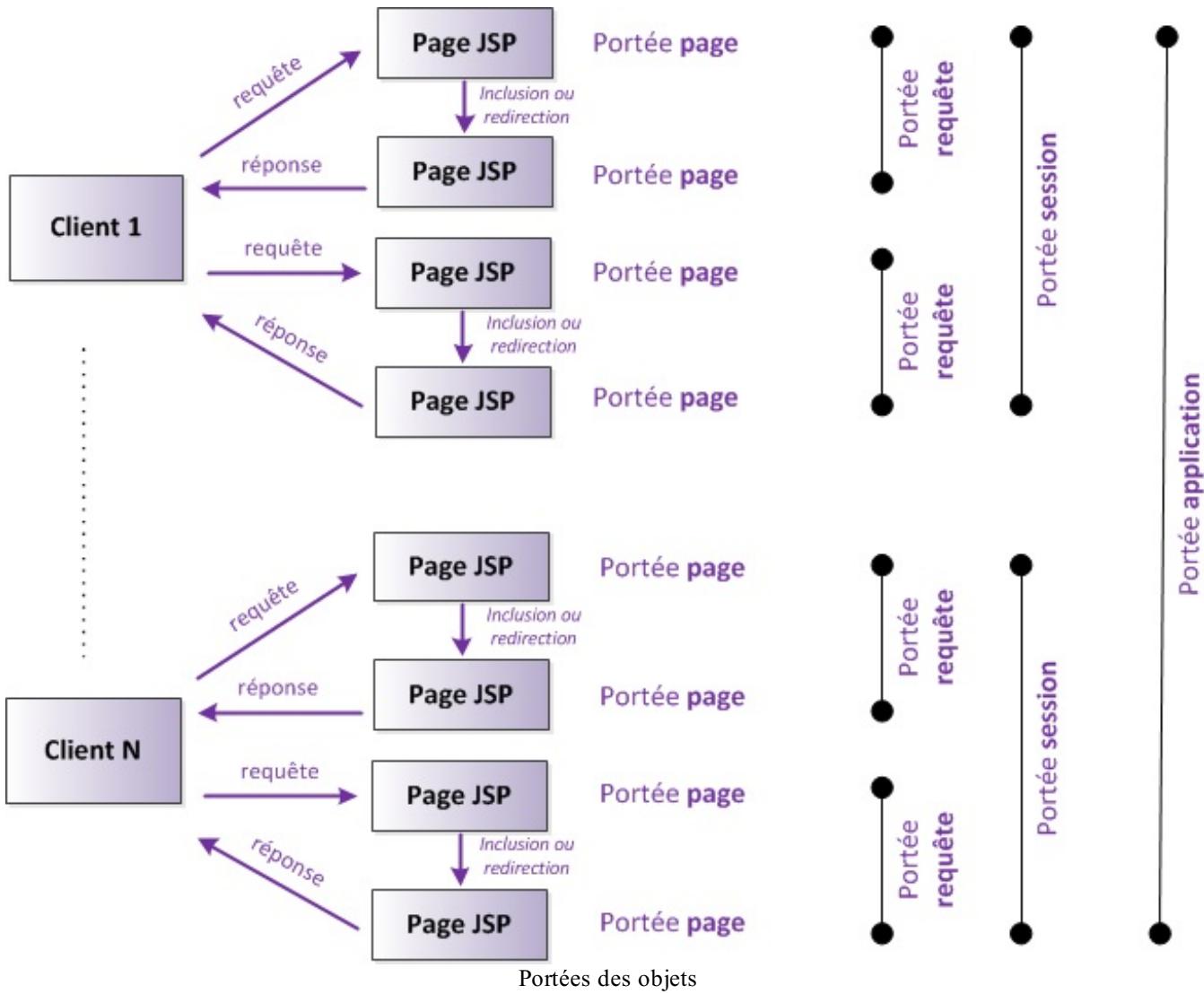
Qu'est-ce qu'une session ?

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 à la figure suivante un schéma regroupant les différentes portées existantes.



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 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.valueOf( 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 de la JSP utilisée :

Code : JSP

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

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

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

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 détail 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 !

- Les commentaires compris entre <%-- et --%> ne sont pas visibles dans la page finale générée.
- L'insertion directe de code Java dans une JSP est possible mais très déconseillée.
- Les directives se placent en début de fichier et permettent de configurer une JSP sous différents angles.
- Il existe 4 portées d'objets différentes, représentant 4 durées de vie différentes : page, request, session et application.
- Une session suit un visiteur de son arrivée sur le site jusqu'à son départ.
- Les actions standard permettent pour la plupart de manipuler des objets au sein d'une JSP, mais sont aujourd'hui de l'histoire ancienne.

La technologie JSP (2/2)

Nous allons dans ce chapitre terminer l'apprentissage de la technologie JSP, à travers la découverte des expressions 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 avertis 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 cela sans nécessiter l'utilisation de code ni de script Java** ! La maintenance de vos pages JSP, en fournissant des notations simples et surtout standard, est ainsi grandement facilitée.

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 expression EL* : je parle tout simplement de ce qui est situé entre les accolades ! 😊

La réalisation de tests

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

- opérateurs arithmétiques, applicables à des nombres : +, -, *, /, % ;
- opérateurs logiques, applicables à des booléens : &&, ||, ! ;
- 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 pour l'occasion 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 que les résultats obtenus correspondent bien aux commentaires que j'ai placés sur chaque ligne.

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 expression 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 ! Pas de panique, vous comprendrez où je veux en venir dans la partie suivante, lorsque nous pratiquerons la JSTL ! 😊

Attention ici aux **opérateurs relationnels** :

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

Autrement dit, pas besoin de vous casser la tête pour un objet de type *String* ou *Integer*, pour lesquels tout est déjà prêt nativement, mais pour des objets de votre propre création et/ou 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 officielle pour plus d'informations. Les applications que nous verrons dans ce cours ne mettront pas en jeu d'expressions EL très 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 ? si oui : 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 -->
```

Pour terminer, sachez enfin que la valeur renournée par une expression 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>
```

La manipulation d'objets

Toutes ces fonctionnalités semblent intéressantes, mais ne nous serviraient pas à grand-chose si elles ne pouvaient s'appliquer qu'à des valeurs écrites en dur dans le code de nos pages, comme nous l'avons fait à l'instant dans nos exemples. La vraie puissance des expressions EL, leur véritable intérêt, c'est le fait qu'elles permettent de manipuler des objets et de leur appliquer tous ces tests ! Quels types d'objets ? Voyons cela au cas par cas...

Des beans

Sous la couverture, la technologie EL est basée sur les spécifications des JavaBeans. Qu'est-ce que cela signifie concrètement ? Eh bien tout simplement qu'il est possible via une expression EL d'accéder directement à une propriété d'un bean ! Pour illustrer cette fonctionnalité sans trop compliquer notre exemple, nous allons utiliser les actions standard que nous avons découvertes dans le chapitre précédent. Éditez le fichier `test_el.jsp` que nous avons mis en place, 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>

```

Grâce aux deux premières actions (lignes 10 et 12), la base de notre exemple est posée : nous créons un bean de type **Coyote** dans notre page JSP, et initialisons sa propriété **prénom** 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 l'expression EL 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 **prénom** du bean nommé **coyote**. Remarquez bien la syntaxe et la convention employées :

- **coyote** est le nom du bean, que nous avions ici défini dans l'attribut **id** de l'action **<jsp:useBean>** ;
- **prénom** 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.

Ainsi de manière générale, il suffit d'écrire `${ bean.propriete }` pour accéder à une propriété d'un bean. Simple et efficace !

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.getAttribute( "coyote" );
if ( bean != null ) {
    String prenom = bean.getPrenom();
    if ( prenom != null ) {
        out.print( prenom );
    }
}

```

Peu importe que nous la comparions avec une scriptlet Java ou avec une action standard, l'expression EL **simplifie l'écriture de manière frappante** ! Ci-dessous, je vous propose d'étudier quelques exemples des utilisations et erreurs de syntaxe les plus courantes :

Code : JSP

```

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

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

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

```

Voilà pour la syntaxe à employer pour accéder à des objets. Maintenant comme je vous l'annonçais en début de paragraphe, la vraie puissance de la technologie EL réside non seulement dans le fait qu'elle permet de manipuler des beans, mais également dans le fait qu'elle permet de les faire intervenir au sein de tests ! Voici quelques exemples d'utilisations, toujours basés sur la propriété **prenom** du bean **coyote** :

Code : JSP

```

<!-- Comparaison d'égalité entre la propriété prenom et la chaîne
"Jean-Paul" -->
${ coyote.prenom == "Jean-Paul" }

<!-- Vérification si la propriété prenom est vide ou nulle -->
${ empty coyote.prenom }

<!-- Condition ternaire qui affiche la propriété prénom si elle
n'est ni vide ni nulle, et la chaîne "Veuillez préciser un prénom"
sinon -->
${ !empty coyote.prenom ? coyote.prenom : "Veuillez préciser un
prénom" }

```

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

<!-- L'action suivante affiche "null" si la propriété "prenom" 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="prenom" />

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

```

Des collections

Les beans ne sont pas les seuls objets manipulables dans une expression EL, il est également possible d'accéder aux collections

au sens large. Cela inclut donc tous les objets de type `java.util.List`, `java.util.Set`, `java.util.Map`, etc.

Nous allons, pour commencer, reprendre notre page `test_el.jsp`, et remplacer son code par cet exemple illustrant l'accès aux éléments d'une liste :

Code : JSP - /test_el.jsp

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>Test des expressions EL</title>
  </head>
  <body>
    <p>
      <%
        /* Création d'une liste de légumes et insertion de quatre
           éléments */
        java.util.List<String> legumes = new
        java.util.ArrayList<String>();
        legumes.add( "poireau" );
        legumes.add( "haricot" );
        legumes.add( "carotte" );
        legumes.add( "pomme de terre" );
        request.setAttribute( "legumes" , legumes );
      %>

      <!-- Les quatre syntaxes suivantes retournent le deuxième
          élément de la liste de légumes -->
      ${ legumes.get(1) }<br />
      ${ legumes[1] }<br />
      ${ legumes['1'] }<br />
      ${ legumes["1"] }<br />
    </p>
  </body>
</html>
```

Toujours afin de ne pas compliquer l'exemple, j'initialise directement une liste de légumes à l'aide d'une scriptlet Java. Rappelez-vous bien que je procède ainsi uniquement pour gagner du temps dans la mise en place de notre exemple, et que lors du développement d'une vraie application, il est hors de question d'écrire une page JSP aussi sale.

Vous découvrez aux lignes 20 à 23 quatre nouvelles notations :

- l'appel à une méthode de l'objet **legumes**. En l'occurrence notre objet est une liste, et nous appelons sa méthode `get()` en lui passant l'indice de l'élément voulu ;
- l'utilisation de crochets à la place de l'opérateur *point*, contenant directement l'indice de l'élément voulu ;
- l'utilisation de crochets à la place de l'opérateur *point*, contenant l'indice de l'élément entouré d'apostrophes ;
- l'utilisation de crochets à la place de l'opérateur *point*, contenant l'indice de l'élément entouré de guillemets.

Chacune d'elles est valide, et retourne bien le second élément de la liste de légumes que nous avons initialisée (souvenez-vous, l'indice du premier élément d'une collection est 0 et non pas 1). Rendez-vous sur la page http://localhost:8080/test/test_el.jsp et constatez par vous-mêmes !

Remplacez ensuite le code de l'exemple par le suivant, illustrant l'accès aux éléments d'un tableau :

Code : JSP - /test_el.jsp

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

```

<%
/* Cr ation d'un tableau */
String[] animaux = {"chien", "chat", "souris", "cheval"};
request.setAttribute("animaux" , animaux);
%>

<!-- Les trois syntaxes suivantes retournent le troisi me
 l ment du tableau --&gt;
${ animaux[2] }&lt;br /&gt;
${ animaux['2'] }&lt;br /&gt;
${ animaux["2"] }&lt;br /&gt;
&lt;/p&gt;
&lt;/body&gt;
&lt;/html&gt;
</pre>

```

Rendez-vous  nouveau sur la page de tests depuis votre navigateur, et vous constaterez que les trois notations avec les crochets, appliqu es   une liste dans le pr c dent exemple, fonctionnent de la m me mani re avec un tableau.



Sachez par ailleurs qu'il est impossible d'utiliser directement l'op rateur *point* pour acc der   un  l ment d'une liste ou d'un tableau. Les syntaxes \${entiers.1} ou \${animaux.2} enverront une exception   l'ex cution.

Enfin, remplacez le code de l'exemple par le suivant, illustrant l'acc s aux  l ments d'une Map :

Code : JSP - /test_el.jsp

```

<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Test des expressions EL</title>
    </head>
    <body>
        <p>
            <%
                /* Cr ation d'une Map */
                java.util.Map<String, Integer> desserts = new
                java.util.HashMap<String, Integer>();
                desserts.put("cookies", 8);
                desserts.put("glaces", 3);
                desserts.put("muffins", 6);
                desserts.put("tartes aux pommes", 2);

                request.setAttribute("desserts" , desserts);
            %>

            <!-- Les quatre syntaxes suivantes retournent la valeur
            associ e   la cl  "cookies" de la Map de desserts --&gt;
            ${ desserts.cookies }&lt;br /&gt;
            ${ desserts.get("cookies") }&lt;br /&gt;
            ${ desserts['cookies'] }&lt;br /&gt;
            ${ desserts["cookies"] }&lt;br /&gt;

            &lt;%
                /* Cr ation d'une cha ne nomm e "element" et contenant le
                mot "cookies" */
                String element = "cookies";
                request.setAttribute("element",element);
            %&gt;
            <!-- Il est  galement possible d'utiliser un objet au lieu
            d'initialiser la cl  souhait e directement dans l'expression --&gt;
            ${ desserts[element] }&lt;br /&gt;
        &lt;/p&gt;
        &lt;/body&gt;
    &lt;/html&gt;
</pre>

```

Rendez-vous une nouvelle fois sur la page de tests depuis votre navigateur, et remarquez deux choses importantes :

- la notation avec l'opérateur *point* fonctionne, à la ligne 21, de la même manière que lors de l'accès à une propriété d'un bean ;
- une expression peut en cacher une autre ! En l'occurrence à la ligne 32, lorsque le conteneur va analyser l'expression EL il va d'abord récupérer l'attribut de requête **element**, puis utiliser son contenu en guise de clé afin d'accéder à la valeur associée dans la Map de desserts.

Par ailleurs, faites bien attention à la syntaxe `${desserts[element]}` employée dans ce dernier cas :

- il est impératif de ne pas entourer le nom de l'objet d'apostrophes ou de guillemets au sein des crochets. En effet, écrire `${desserts["element"]}` reviendrait à essayer d'accéder à la valeur associée à la clé nommée "element", ce qui n'est pas le comportement souhaité ici ;
- il ne faut pas entourer l'expression contenue dans l'expression englobante avec des accolades. Autrement dit, il ne faut pas écrire `${desserts[${element}]}`, cette syntaxe n'est pas valide. Une seule paire d'accolades suffit ! 😊



Puisque les notations avec l'opérateur *point* et avec les crochets fonctionnent toutes deux dans le cas d'une Map, quelle est la notation recommandée ?

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

Toutefois, ce n'est pas une obligation et vous êtes libres de suivre ou non cette bonne pratique dans vos projets, je ne vous conseille rien de particulier à ce sujet. Personnellement j'utilise dès que j'en ai l'occasion la notation avec l'opérateur *point*, car je trouve qu'elle a tendance à moins surcharger visuellement les expressions EL dans le code de mes pages JSP.



Pour clore sur ce sujet, je ne vous l'ai pas dit lorsque nous avons découvert la manipulation des beans dans une expression EL, mais sachez que l'opérateur *point* n'est pas la seule manière d'accéder à une propriété d'un bean, il est également possible d'utiliser la notation avec les crochets : `${bean["propriété"]}`.

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 interprétées, 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**. Éditez le fichier et insérez-y en première ligne la directive. Enregistrez la modification et actualisez alors l'affichage de la page dans votre navigateur. Vous observerez que vos expressions ne sont plus évaluées, elles sont cette fois simplement affichées telles quelles.

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 de l'API servlet utilisé par votre application :

- si la version 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 savoir si une application permet d'utiliser les expressions EL ou non ?

Pour commencer, il faut s'assurer de la version de l'API servlet supportée par le conteneur qui fait tourner l'application. En l'occurrence, nous utilisons Tomcat 7 et celui-ci implémente la version 3.0 : tout va bien de ce côté, notre conteneur est capable de gérer les expressions EL.

Ensuite, il faut s'assurer que l'application est déclarée correctement pour utiliser cette version. Cela se passe au niveau de la balise **<web-app>** du fichier **web.xml**. Lorsque nous avions mis en place celui de notre projet, je vous avais dit de ne pas vous soucier des détails et de vous contenter d'écrire :

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

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
    version="3.0">

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

La mise en place de tous ces attributs implique que notre application va se baser sur la version 3.0 de l'API servlet.

Ainsi, puisque notre conteneur le supporte et que notre application est correctement configurée, nous pouvons en déduire que les expressions EL seront interprétées par défaut dans notre application.

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 supportée par le conteneur utilisé soit différente. Si par exemple vous travaillez sur une application un peu ancienne tournant sur un conteneur Tomcat 6, alors la version maximum de l'API servlet supportée est 2.5. Voici alors comment vous devrez déclarer votre application :

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

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    version="2.5">

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

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

```
<%@ page pageEncoding="UTF-8" %>
<!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>
```

```
</p>
</body>
</html>
```

Je vous avais alors fait remarquer qu'à la ligne 13, nous avions directement utilisé l'objet **out** sans jamais l'avoir instancié auparavant. De même, à la ligne 12 nous accédions 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 rappellez-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 qui 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 ;
- à 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	<code>PageContext</code>	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	<code>ServletContext</code>	Il permet depuis une page JSP d'obtenir ou de modifier des informations relatives à l'application dans laquelle elle est exécutée.
session	<code>HttpSession</code>	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	<code>HttpServletRequest</code>	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	<code>HttpServletResponse</code>	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	<code>Throwable</code>	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	<code>JspWriter</code>	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	<code>ServletConfig</code>	Il permet depuis une page JSP d'obtenir les éventuels paramètres d'initialisation disponibles.
page	objet <code>this</code>	Il est l'équivalent de la référence <code>this</code> 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 peut vous paraître compliqué, mais en réalité c'est très simple. C'est un outil incontournable en Java, et nous venons d'en manipuler une lorsque nous avons découvert les expressions EL. Mais si jamais 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 comme nous venons de le découvrir, 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. En guise de rappel, continuons notre exemple avec la liste d'aliments, et créons une page **test_map.jsp**, dans laquelle nous allons implémenter rapidement cette 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. Comme je vous l'ai annoncé un peu plus tôt, j'utilise la notation avec l'opérateur *point* - ici dans les lignes 18 à 21 - pour accéder aux valeurs contenues dans la Map, mais il est tout à fait possible d'utiliser la notation avec les crochets.



D'accord, avec des expressions EL, nous pouvons accéder au contenu d'objets de type Map. Mais ça, nous le savions



déjà... Quel est le rapport avec les objets implicites EL ?

Le rapport, c'est que tous ces objets sont des Map, et que par conséquent nous sommes capables d'y accéder depuis des expressions EL, de la même manière que nous venons de parcourir notre Map d'aliments ! 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 session, 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 reconnaisserez 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](http://localhost:8080/test/test_obj_im [...] r&article=782) pour que votre navigateur vous affiche ceci (voir la figure suivante).

← → C localhost:8080/test/test_obj_impl.jsp?langue=fr&article=782

Langue : fr
Article : 782

Cherchez maintenant, dans le tableau fourni précédemment, 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 ? 😊

Tout simplement en précisant plusieurs fois un paramètre d'URL avec des valeurs différentes ! Par exemple, accédez cette fois à la page de tests avec l'URL [http://localhost:8080/test/test_obj_im \[...\] 782&langue=zh](http://localhost:8080/test/test_obj_im [...] 782&langue=zh). Cette fois, la technologie EL va vous mettre à disposition un objet dont le contenu peut être représenté ainsi :

Nom du paramètre (Clé)	Valeur du paramètre (Valeur)
langue	[fr,zh]
article	782

La Map permettant d'accéder aux valeurs du paramètre **langue** n'est plus une Map<String, String>, mais une Map<String, String[]>. Si vous ne modifiez pas le code de l'expression EL dans votre page JSP, alors vous ne pourrez qu'afficher la première valeur du tableau des langues, renommée par défaut lorsque vous utilisez l'expression \${param.langue}.

Afin d'afficher la seconde valeur, il faut cette fois non plus utiliser l'objet implicite **param**, mais utiliser l'objet implicite nommé **paramValues**. Remplacez à la ligne 9 de votre fichier **test_obj_impl.jsp** l'expression \${param.langue} par l'expression \${paramValues.langue[1]}. Actualisez alors la page dans votre navigateur, et vous verrez alors s'afficher la valeur **zh** !

 Le principe est simple : alors qu'auparavant en écrivant \${param.langue} vous accédiez directement à la String associée au paramètre **langue**, cette fois en écrivant \${paramValues.langue} vous accédez non plus à une String, mais à un tableau de String. Voilà pourquoi il est nécessaire d'utiliser la notation avec crochets pour accéder aux différents éléments de ce tableau !

En l'occurrence, puisque seules deux langues ont été précisées dans l'URL, il n'existe que les éléments d'indices 0 et 1 dans le tableau, contenant les valeurs **fr** et **zh**. Si vous essayez d'accéder à un élément non défini, par exemple en écrivant \${paramValues.langue[4]}, alors l'expression EL détectera une valeur nulle et n'affichera rien. De même, vous devez **obligatoirement** cibler un des éléments du tableau ici. Si vous n'écrivez que \${paramValues.langue}, alors l'expression EL vous affichera la référence de l'objet Java contenant votre tableau...

Par ailleurs, sachez qu'il existe d'autres cas impliquant plusieurs valeurs pour un 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="multiple">
      <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="multiple"**, il est tout à fait possible de sélectionner plusieurs valeurs pour le seul paramètre nommé **pays**. Eh bien dans ce genre de cas, l'utilisation de l'objet implicite **paramValues** est nécessaire également : 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 pour un seul paramètre d'un en-tête HTTP, celles-ci sont la plupart du temps séparées par de simples points-virgules et concaténées dans une seule 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, l'important pour le moment est que vous compreniez bien leur mode de fonctionnement. Ne vous inquiétez pas si vous ne saisissez pas l'utilité de chacun d'entre eux, c'est tout à fait normal, certains concepts vous sont encore inconnus. La pratique vous fera prendre de l'aisance, et j'apporterai de plus amples explications au cas par cas dans les exemples de ce cours. Avant de passer à la suite, un petit avertissement quant au 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 des 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 !

- La technologie EL est fondée sur les JavaBeans et sur les collections Java, et existe depuis la version 2.4 de l'API Servlet.
- Les expressions EL remplacent les actions standard de manipulation des objets.
- Une expression EL permet d'effectuer des tests, interprétés à la volée lors de l'exécution de la page.
- L'interprétation des expressions EL peut être désactivée via une section dans le fichier web.xml.
- Un objet implicite n'est pas géré par le développeur, mais par le conteneur de servlets.
- Chaque objet implicite JSP donne accès à un objet mis à disposition par le conteneur.
- Chaque objet implicite EL est un raccourci vers des données de l'application.

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

```
<%@ page pageEncoding="UTF-8" %>
<!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

```
<%@ page pageEncoding="UTF-8" %>
<!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 %>
    </p>
  </body>
</html>
```

```

        <% 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" />
```

Ici, 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 attributs et paramètres depuis la requête...

Avec des EL

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

```

<%@ page pageEncoding="UTF-8" %>
<!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}
        ${param.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 !



Pourquoi est-ce que nous n'utilisons aucun objet implicite pour accéder aux attributs **test** et **coyote** ?

J'ai jusqu'à présent lâchement évité ce sujet pour ne pas vous embrouiller, mais cette question mérite effectivement d'être posée : comment notre expression EL est-elle capable de trouver nos différents objets ? Quand nous avions découvert les actions standard, le problème ne se posait pas, car il était nécessaire de préciser dans quelle portée nous souhaitions récupérer un objet, sans quoi l'action cherchait par défaut dans la portée **page**...

Eh bien le fonctionnement de la technologie EL est fondamentalement différent de celui des actions standard. Dans une expression EL, lorsque vous accédez à un objet présent dans une des quatre portées de votre application, vous n'avez pas besoin de spécifier l'objet implicite (c'est-à-dire la portée) auquel vous souhaitez accéder. D'ailleurs vous le voyez bien dans notre exemple, nous n'avons pas écrit `${request.test}` pour accéder à l'objet **test** présent dans la portée **request**, ni `${request.coyote.prenom}` pour accéder au bean **coyote** présent lui aussi dans la portée **request**.

D'ailleurs, si vous aviez fait ainsi... ça n'aurait pas fonctionné ! N'oubliez pas que l'objet implicite **request** ne représente pas la portée **request**, mais directement l'objet requête en cours d'utilisation. En réalité, le mécanisme de la technologie EL est un peu évolué : une expression est capable de réaliser d'elle-même un parcours automatique des différentes portées accessibles à la recherche d'un objet portant le nom précisé, de la plus petite à la plus grande portée.



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 ...

Concrètement, lorsque vous écrivez par exemple l'expression `${test}` dans votre JSP, derrière les rideaux le conteneur va se rendre compte qu'il ne s'agit pas d'un objet implicite mais bien d'un objet de votre création, et va appeler la méthode `findAttribute()` de l'objet **PageContext**. Ensuite, cette méthode va à son tour parcourir chacune des portées - **page**, puis **request**, puis **session** et enfin **application** - pour retourner le premier objet nommé "test" trouvé !



La bonne pratique de développement est de ne jamais donner le même nom à des objets existant dans des portées différentes ! Par exemple, si vous écrivez l'expression `${test}` pour cibler un objet nommé "test" que vous avez enregistré en session, alors qu'il existe un autre objet nommé "test" en requête, puisque la portée **request** sera toujours parcourue avant la portée **session** lors de la recherche automatique du mécanisme EL, c'est l'objet enregistré dans la requête qui vous sera renvoyé...



Dans ce cas, comment éviter ce parcours automatique et cibler directement une portée ?

Pour cela, il faut utiliser les objets implicites fournis par la technologie EL donnant accès aux attributs existant : **pageScope**, **requestScope**, **sessionScope** et **applicationScope**. Ainsi, dans notre précédent exemple nous aurions très bien pu écrire `${requestScope.test}` à la place de `${test}`, et cela aurait fonctionné tout aussi bien. Lors de l'analyse de l'expression EL, le conteneur aurait ainsi reconnu l'objet implicite **requestScope**, et n'aurait pas effectué le parcours des portées : il aurait

directement recherché un objet nommé "test" **au sein de la portée request uniquement.**



La bonne pratique veut qu'en complément de la rigueur dans le nommage des objets conseillée précédemment, le développeur précise toujours la portée qu'il souhaite cibler dans une expression EL. Ainsi pour accéder à un objet présent par exemple dans la portée **request**, cette pratique recommande non pas d'écrire `${test}`, mais `${requestScope.test}`. En procédant ainsi, le développeur s'assure qu'aucun objet ne sera ciblé par erreur.

En ce qui me concerne dans la suite de ce cours, je prendrai toujours garde à ne jamais donner le même nom à deux objets différents. Je me passerai donc de préciser la portée dans chacune des expressions EL que j'écrirai dans mes exemples, afin de ne pas les alourdir. Il va de soi que lors du développement d'une vraie application web, je vous recommande de suivre les bonnes pratiques que je vous ai énoncées à l'instant.



Qu'en est-il du paramètre "auteur" ?

Notez bien que lorsque vous souhaitez cibler un objet qui n'est pas présent dans une des quatre portées, il est bien entendu nécessaire d'expliciter l'objet implicite qui permet d'y accéder au sein de l'expression EL. Voilà pourquoi pour accéder au paramètre de requête **auteur**, nous devons bien préciser `${param.auteur}`. Si nous avions simplement écrit `${auteur}` cela n'aurait pas fonctionné, car le mécanisme de recherche automatique aurait tenté de trouver un objet nommé "auteur" dans une des quatre portées - **page, request, session** puis **application** - et n'aurait logiquement rien trouvé. Rappelez-vous bien qu'un paramètre de requête est différent d'un attribut de requête ! 😊

Enfin, comprenez bien que je prends ici l'exemple d'un paramètre de requête pour illustrer le principe, mais que ceci est valable pour tout objet implicite différent des quatre portées : **param** comme nous venons de le voir, mais également **header, cookie, paramValues**, etc.

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

Reprendons 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 nous 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 pageEncoding="UTF-8" %>
<%@ 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}
            ${param.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

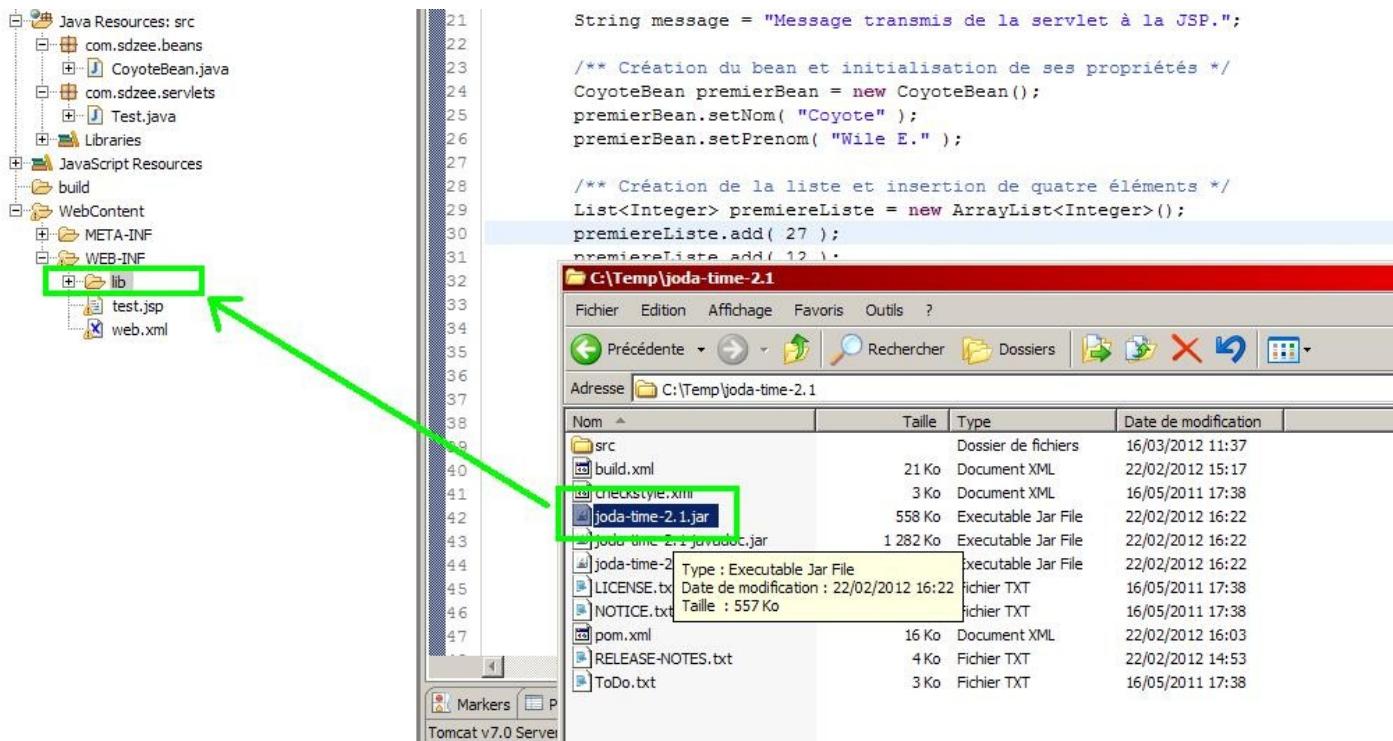


dates : JodaTime.

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 la figure suivante).



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." );

/* Crédit 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 librairie 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 proposés, c'est d'une simplicité impressionnante.

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

```

<%@ page pageEncoding="UTF-8" %>
<%@ 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}
            ${param.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>

```

```
</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 posé 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 rendu 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 ; 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-mêmes 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 sont légion sur le web, notamment au sujet de la plate-forme Java EE** qui est 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.

- Les expressions EL remplacent élégamment scriptlets et actions standard.
- La technologie EL ne répond malheureusement pas à tous nos besoins.
- La documentation est indispensable, à condition qu'elle soit à jour.

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 en partant de zéro dans votre environnement, et y créer vos différents fichiers. Le second objectif est que vous soyez à 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 d'un client

À travers notre petite application, l'utilisateur doit 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 une 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 d'une commande

L'utilisateur doit également pouvoir créer une commande, en saisissant des données depuis un formulaire, et visualiser la fiche en résultant. De même, puisque vous n'avez pas encore découvert les formulaires, je vais vous fournir une 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 utiliser ces deux formulaires comme base pour votre application. Vous les placerez directement à la racine de votre application, sous le répertoire **WebContent** d'Eclipse.

Création d'un client

Code : JSP - /creerClient.jsp

```
<%@ page pageEncoding="UTF-8" %>
<!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">Prénom </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 d'une commande

Code : JSP - /creerCommande.jsp

```

<%@ page pageEncoding="UTF-8" %>
<!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" />

```

```
name="nomClient" value="" size="20" maxlength="20" />
<br />

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

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

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

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

    <label for="modePaiementCommande">Mode de
paiement *</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></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

```
/* 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")` ;
- 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 aurez donc à 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.

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 devrez donc récupérer directement la date courante depuis votre servlet et l'enregistrer au format `String` dans le bean.

Les vues

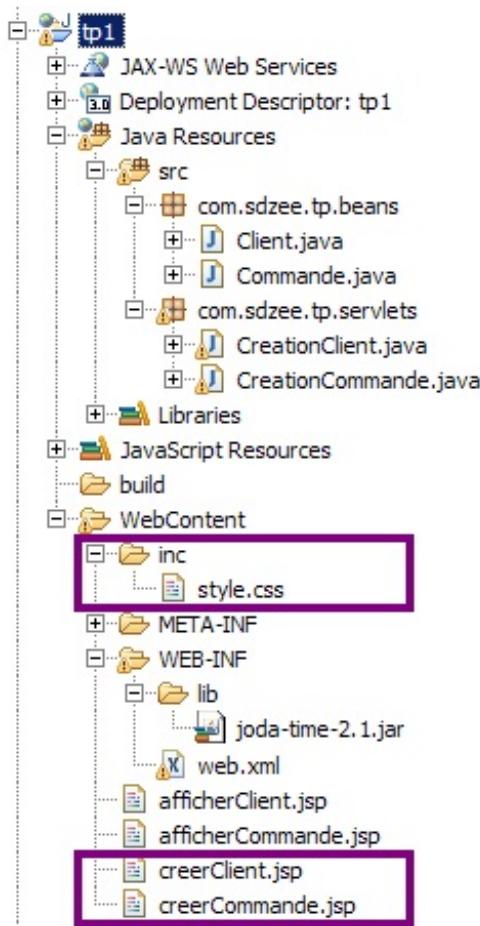
Je vous fournis les deux pages JSP contenant les formulaires de saisie des données. Les seules vues que vous avez à développer sont celles qui affichent les données saisies, après validation du formulaire. Vous pouvez par exemple les nommer **afficherClient.jsp** et **afficherCommande.jsp**. Elles vont 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 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 en partant 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.

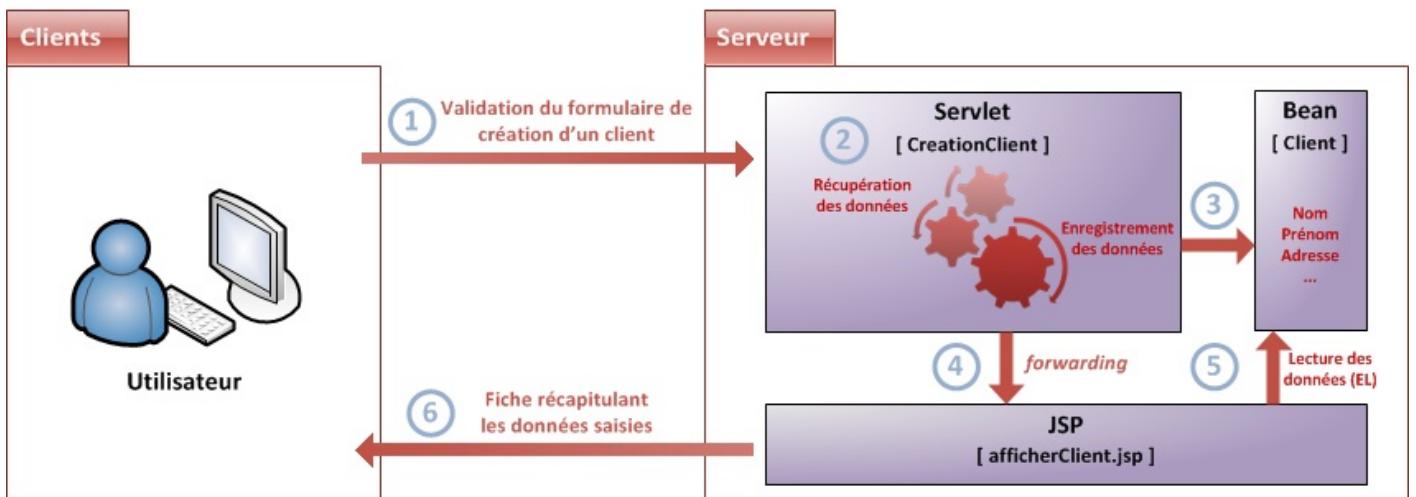
Pour conclure, voici à la figure suivante ce à quoi est supposée ressembler l'architecture de votre projet fini si vous avez suivi mes conseils et exemples de nommage.



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

Illustration du comportement attendu

À la figure suivante, 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 (voir les figures suivantes).

localhost:8080/tp1/creerClient.jsp

Informations client

Nom *	DUPOND
Prénom	Marcel
Adresse de livraison *	12 rue de la marmotte, Avoriaz
Numéro de téléphone *	0412345678
Adresse email	marcel.dupond@mail.com

Valider Remettre à zéro

Saisie de données valides dans le formulaire

localhost:8080/tp1/creationClient?nomClient=DUP

Client créé avec succès !

Nom : DUPOND

Affichage du message de

Prénom : Marcel

Adresse : 12 rue de la marmotte, Avoriaz

Numéro de téléphone : 0412345678

Email : marcel.dupond@mail.com

succès et des données

Avec erreur (voir les figures suivantes).

Informations client

Nom *	DUPOND
Prénom	Marcel
Adresse de livraison *	12 rue de la marmotte, Avoriaz
Numéro de téléphone *	
Adresse email	marcel.dupond@mail.com

Oubli d'un champ obligatoire

Valider **Remettre à zéro**

dans le formulaire

← → C localhost:8080/tp1/creationClient?nomClient=DUP ☆ ⚡ 🔍

*Erreur - Vous n'avez pas rempli tous les champs obligatoires.
Cliquez ici pour accéder au formulaire de création d'un client.*

Nom : DUPOND

Affichage du message d'erreur

Prénom : Marcel

Adresse : 12 rue de la marmotte, Avoriaz

Numéro de téléphone :

Email : marcel.dupond@mail.com

et des données

Création d'une commande

Avec succès (voir les figures suivantes).

Informations client

Nom *	DUPOND
Prénom	Marcel
Adresse de livraison *	12 rue de la marmotte, Avoriaz
Numéro de téléphone *	0412345678
Adresse email	marcel.dupond@mail.com

Informations commande

Date *	
Montant *	499.90
Mode de paiement *	Chèque
Statut du paiement	
Mode de livraison *	48H chrono
Statut de la livraison	

Saisie de données valides dans

Valider **Remettre à zéro**

le formulaire



Commande créée avec succès !

Client

Nom : DUPOND

Prénom : Marcel

Adresse : 12 rue de la marmotte, Avoriaz

Numéro de téléphone : 0412345678

Email : marcel.dupond@mail.com

Commande

Date : 14/06/2012 10:37:16

Montant : 499.9

Mode de paiement : Chèque

Statut du paiement :

Mode de livraison : 48H chrono

Statut de la livraison :

Affichage du message de succès et des données

Avec erreur (voir les figures suivantes).



Informations client

Nom *	DUPOND
Prénom	Marcel
Adresse de livraison *	12 rue de la marmotte, Avoriaz
Numéro de téléphone *	
Adresse email	marcel.dupond@mail.com

Informations commande

Date *	
Montant *	abcdef
Mode de paiement *	Chèque
Statut du paiement	
Mode de livraison *	
Statut de la livraison	

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



Client

Nom : DUPOND

Prénom : Marcel

Adresse : 12 rue de la marmotte, Avoriaz

Numéro de téléphone :

Email : marcel.dupond@mail.com

Commande

Date : 14/06/2012 10:40:14

Montant : -1.0

Mode de paiement : Chèque

Statut du paiement :

Mode de livraison :

Statut de la livraison :

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 le début, puis de chercher et de 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

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

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 d'un 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 d'une 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

Page d'affichage d'un client :

Code : JSP - afficherClient.jsp

```

<%@ page pageEncoding="UTF-8" %>
<!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>

```

```

</head>
<body>
    <%-- Affichage de la chaîne "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>Prénom : ${ 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

```

<%@ page pageEncoding="UTF-8" %>
<!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 chaîne "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>Prénom : ${ 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. Ils vous serviront de base pour les prochaines étapes du fil rouge !

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, vous allez découvrir 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 couches 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 des 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, d'abord en Java (à base de scriptlet donc), puis via des balises JSTL. 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 ; elle ne fait plus intervenir d'attributs et de 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, moins vous aurez à écrire de code, moins vous serez susceptibles 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 des 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 inconvénients des scriptlets sont les suivants :

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'avez 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. 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 manipuler ? Sans aucune hésitation, lorsque les balises JSTL sont utilisées, la taille des pages est fortement réduite. La compréhension et la 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 se 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 maîtrisent pas aussi bien, voire pas du tout.

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 la plate-forme Java EE 6, avec un conteneur JSP 2.1 ou 3.0 (*ex: Tomcat 6 et 7*).

Les différences entre ces versions résident principalement dans le conteneur JSP nécessaire. Le changement majeur à retenir dans le passage de la première version à la seconde version de ce conteneur, c'est la gestion de la technologie EL. Le conteneur JSP 1.2 sur lequel est basée la JSTL 1.0 ne gérait pas les expressions EL, cette dernière proposait donc deux implémentations pour pallier ce manque : une les interprétant et l'autre non. Ceci se traduisait alors par l'utilisation d'adresses différentes lors de la déclaration des bibliothèques, nous allons revenir sur cela un petit peu plus loin.

La version 1.1 est basée sur le conteneur JSP 2.0, qui intègre nativement un interpréteur d'expressions 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 souhaitez 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'avérera 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 (voir la figure suivante).

```
<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> Unknown tag (c:out). Press 'F2' for focus
```

Warning Eclipse : balise inconnue !

Il ne connaît 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... Ça vous revient en mémoire ? 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 travaillez 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", j'aurais alors dû 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 (voir la figure suivante) !

```
<%@ 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>
```

Erreur Eclipse : bibliothèque introuvable !



Effectivement, nouvelle erreur. Pourquoi ?

Eh bien cette fois, c'est Tomcat qui est en cause ! 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 pas, par défaut, 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**, comme indiqué à la figure suivante.



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 : je ne suis pas vraiment certain de la raison pour laquelle Eclipse ne met pas directement ses avertissements à jour. 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 à vérifier que tout se passe bien, en accédant à votre JSP depuis votre navigateur via l'adresse <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 !

- la JSTL est composée de cinq bibliothèques de balises standard ;
- elle permet d'éviter l'utilisation de code Java dans les pages JSP ;
- elle permet de réduire la quantité de code à écrire ;
- elle rend le code des pages JSP plus lisible ;
- sous Tomcat, il faut placer son fichier .jar sous **/WEB-INF/lib** pour qu'elle soit correctement intégrée.

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 --%>
```

À celui-ci s'ajoutent deux attributs optionnels :

- **default** : permet de définir une valeur affichée par défaut si le contenu de l'expression évaluée est vide ;
- **escapeXml** : permet de remplacer les caractères de scripts `<, >, ", ' et &` par leurs équivalents en code html `<, >, ", ', &`. 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 : --%>
<p>Je suis un &#039;paragraphe&#039;.</p>

<%-- 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'elles 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}" />" />
```

Nous le découvrirons plus tard, mais sachez que 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. Ne vous posez pas trop de questions au sujet de cet exemple, nous reviendrons en détail 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 corollaire 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 est 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 "message" de la requête : --%>
<c:set var="message" value="Salut les zéros !" scope="request" />

<%-- Et est l'équivalent du scriplet Java suivant : --%>
<% request.setAttribute( "message", "Salut les zéros !" ); %>
```

L'attribut **var** contient le nom de la variable que l'on veut stocker, **value** sa valeur, et **scope** la portée de cette variable. 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 "message" de la
   requête --%>
<c:out value="${requestScope.message}" />
```

 Vous remarquerez que nous utilisons ici dans l'expression 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="${ message }"/>`. 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 **message**, et nous le renvoie !

 N'oubliez pas : la bonne pratique veut que vous ne donnez 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 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 utilisé des objets implicites. Nous y reviendrons de toute manière quand 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 balises `<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 par rapport au 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, initialisée 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 pas 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** : cet 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 autorisée : 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'une 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 associera le titre d'une news à 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<Map<String, String>> maListe = new ArrayList<Map<String, String>>();
    Map<String, String> news = new HashMap<String, String>();
    news.put("titre", "Titre de ma première news");
    news.put("contenu", "corps de ma première news");
    maListe.add(news);
    news = new HashMap<String, String>();
    news.put("titre", "Titre de ma seconde news");
    news.put("contenu", "corps de ma seconde news");
    maListe.add(news);
    pageContext.setAttribute("maListe", maListe);
%>

<c:forEach items="${maListe}" var="news">
<div class="news">
    <div class="titreNews">
        <c:out value="${news['titre']}" />
    </div>
    <div class="corpsNews">
        <c:out value="${news['contenu']}" />
    </div>
</div>

```

```
</c:forEach>
```

Je sens que certains vont m'attendre au tournant... Eh oui, j'ai 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 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 trouvent les JSP s'appelle la couche de présentation.

Revenons à notre boucle : ici, je n'ai 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` nommée `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 titres et contenus de nos news, il suffit, via la notation avec crochets, de préciser les éléments visés dans notre Map, ici aux lignes 19 et 22. Nous aurions très bien pu utiliser à la place des crochets l'opérateur `point` : `${news.titre}` et `${news.contenu}`.

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 mis entre parenthèses 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. Cependant, volontairement, je n'aborderai 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 ne 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, utilisable 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
<code>begin</code>	La valeur de l'attribut begin.
<code>end</code>	La valeur de l'attribut end.
<code>step</code>	La valeur de l'attribut step.
<code>first</code>	Booléen précisant si l'itération courante est la première.
<code>last</code>	Booléen précisant si l'itération courante est la dernière.
<code>count</code>	Compteur d'itérations (commence à 1).
<code>index</code>	Index d'itérations (commence à 0).
<code>current</code>	Élément courant de la collection parcourue.

Reprendons l'exemple utilisé précédemment, mais cette fois-ci en 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 utilisé ici la propriété **count** de l'attribut **varStatus**, affichée 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 nommée... **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+!">
    ${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. Par conséquent, il est impossible la plupart du temps 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 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, une 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="

```

Reprendons 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 assez intuitive :

Code : JSP

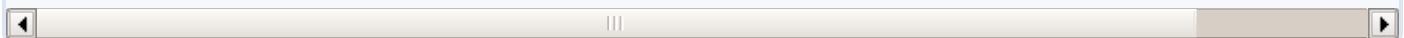
```
<c:url value="/monSiteWeb/countZeros.jsp">
  <c:param name="nbZeros" value="${countZerosBean.nbZeros}" />
  <c:param name="date" value="22/06/2010" />
</c:url>
```

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 transparente du point de vue de l'utilisateur (l'URL dans la barre de navigation du navigateur n'est pas modifiée).

La raison de cette différence de comportement est simple : le *forwarding* se fait côté serveur, contrairement à la redirection qui est effectuée par le navigateur. Cela limite par conséquent la portée de l'action de *forwarding* qui, puisque exécutée côté serveur,

est limitée aux pages présentes dans le contexte de la servlet utilisée. La redirection étant 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 appelée via l'URL d'origine, ou par une autre page vers laquelle la première servlet ou JSP appelée a effectué un *forwarding* ! Ceci peut donc poser problème si l'utilisateur rafraîchit la page par exemple, puisque cela appellera à nouveau la servlet ou JSP d'origine... 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 page="/monSiteWeb/erreur.jsp">

<%-- Redirection avec la balise redirect --%>
<c:redirect url="http://www.siteduzero.com"/>

<%-- Les attributs valables pour <c:url/> le sont aussi pour la
redirection.
Ici par exemple, l'utilisation de paramètres --%>
<c:redirect url="http://www.siteduzero.com">
    <c:param name="mascotte" value="zozor"/>
    <c:param name="langue" value="fr"/>
</c:redirect>

<%-- Redirigera vers --%>
http://www.siteduzero.com?mascotte=zozor&langue=fr
```

Imports

La balise `<c:import>` est en quelque sorte un équivalent à `<jsp:include>`, 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 détail 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 stocké 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 par rapport au *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>
```

Les autres bibliothèques de la JSTL

Au terme de ce chapitre, vous devez être capables 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 !
- On affiche le contenu d'une variable ou d'un objet avec `<c:out>`.
- On effectue un test avec `<c:if>` ou `<c:choose>`.

- On réalise une boucle sur une collection avec `<c:forEach>`.
- On génère un lien de manière automatique avec `<c:url>`.
- On redirige vers une autre page avec `<c:redirect>`.
- On importe une autre page avec `<c:import>`.

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 entraîner !

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 les pages JSP !

Une fois que c'est 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

```
<%@ page pageEncoding="UTF-8" %>
<!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" placeholder="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 visités, en les séparant par une virgule.

Voici à la figure suivante le rendu, rempli avec des données de test.

The screenshot shows a web browser window with the URL `localhost:8080/test/initForm.jsp`. The page contains the following form fields:

- A text input field labeled "Entrez ici votre nom de famille :" containing the value "Coyote".
- A text input field labeled "Entrez ici votre prénom :" containing the value "Wile E.".
- A dropdown menu labeled "Dans quel(s) pays avez-vous déjà voyagé ?" with the following options:
 - France
 - Espagne
 - Italie
 - Royaume-UniThe option "Italie" is currently selected.
- A text input field labeled "Entrez ici les autres pays que vous avez visités, séparés par une virgule :" containing the value "Laponie de l'Ouest, Nouvelle-Zélande, Guyane".
- At the bottom are two buttons: "Valider" and "Remettre à zéro".

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. Tout ce que vous avez besoin de savoir ici, c'est que les données saisies par le client dans les champs du formulaire seront accessibles dans votre JSP à travers les **paramètres de requêtes**, autrement dit l'objet implicite **param**. Avec la JSTL et les expressions 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, l'intérêt est de vous familiariser avec le développement de pages et la manipulation de données en utilisant la JSTL. Ne vous préoccupez pas de l'architecture factice mise en place, et ne vous intéressez pas conséquent 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 rien de bien compliqué. La page devra également être placée à la racine du projet, sous le répertoire **WebContent**, et sera donc accessible après validation du formulaire de la page http://localhost:8080/jstl_exo1/initForm.jsp. Elle devra simplement afficher :

1. une liste récapitulant le nom de chaque champ du formulaire et les informations qui y ont été saisies ;
2. le nom et le 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 :

Code : JSP

```
<%@ page pageEncoding="UTF-8" %>
<%@ 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 paramètre pays. S'il existe on le
traite,
            sinon on affiche un message par défaut.--%>

            <%-- Teste l'existence du paramètre autre. Si des données
existent on les traite,
            sinon 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 :

Code : JSP - initProcess.jsp

```
<%@ page pageEncoding="UTF-8" %>
<%@ 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 paramètre pays. S'il existe on le
traite,
sinon 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>
    </body>
</html>
```

```

<%-- Teste l'existence du paramètre autre. Si des données existent
on les traite,
sinon 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 à la figure suivante le rendu avec les données de test.



Vous avez renseigné les informations suivantes :

- **prenom :**
Wile E.
- **autre :**
Laponie de l'Ouest, Nouvelle-Zélande, Guyane
- **pays :**
France Italie Chine
- **nom :**
Coyote Rendu du formulaire du TP Core.

Vous vous nommez :

Coyote Wile E.

Vous avez visité les pays suivants :

France
Italie
Chine
Laponie de l'Ouest
Nouvelle-Zélande
Guyane

J'ai utilisé ici des tests conditionnels et différentes boucles afin de vous faire pratiquer, et mis en jeu différents objets implicites. J'aurais 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 cette simple récupération de données, vous devez identifier les points qui vous ont posé problème et revoir le cours plus attentivement !

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 moments un peu plus expéditif. 😊

La syntaxe XPath

Pour vous permettre de comprendre simplement les notations que j'utiliserais 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, 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 noeuds dans un document XML. Il fournit une syntaxe permettant de cibler directement un fragment du document traité, comme un ensemble de noeuds ou encore un attribut d'un noeud 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 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 du fichier XML de test que j'utiliserais dans les quelques exemples illustrant ce paragraphe :

Code : XML - monDocument.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>
```

La syntaxe XPath

Plutôt que de paraphraser, voyons directement comment sélectionner diverses portions de ce document via des expressions XPath, à travers des exemples commentés :

Code : XML

```

<!-- Sélection du nœud racine -->
/
<!-- Sélection des nœuds 'article' enfants des nœuds 'news' -->
/news/article

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

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

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

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

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

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

<!-- Sélection des nœuds 'article' enfants de 'news' dont l'attribut
id vaut '12' -->
/news/article[@id='12']

```

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 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 vieillissants, 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 parseé, 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 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 accédera à ces fichiers externes lors du parseage 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 s'agit.

Afficher une expression

Les noms des balises que nous allons maintenant aborder devraient vous être familiers : ils trouvent leurs équivalents dans la bibliothèque *Core* que vous avez découverte dans le chapitre précédent. Alors que les balises de type *Core* accédaient à des données de l'application en utilisant des EL, les balises de la bibliothèque *xml* vont 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 noeuds "auteur" du document !
Où sont passés Paul et Jacques ? Où est l'erreur ?

Hé hé... Eh bien, à vrai dire il n'y a aucune erreur ! En effet, l'expression XPath renvoie bel et bien un ensemble de noeuds, en l'occurrence les noeuds "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 noeuds, et n'affiche que le premier noeud 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 noeuds, 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 noeuds 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 contenu 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 des 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 cette 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 bibliothèque *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 boucles 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 transformations

Transformations

La 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:transform>**. Commençons par un petit exemple, afin de comprendre comment elle fonctionne. J'utiliserais ici le même fichier XML que pour les exemples précédents, ainsi que la feuille de style XSL suivante :

Code : XML - test.xml

```
<?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>
```

```

    </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 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

```

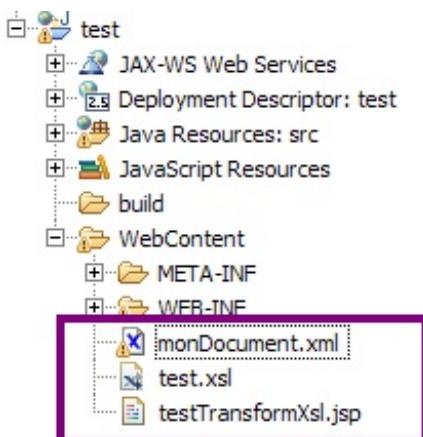
<x:transform xslt="${xslFile}">
  <news>
    <article id="1">
      <auteur>Pierre</auteur>
      <titre>Foo...</titre>
      <contenu>...bar !</contenu>
    </article>
    <article id="27">
      <auteur>Paul</auteur>
      <titre>Bientôt un LdZ J2EE !</titre>
      <contenu>Woot ?</contenu>
    </article>
    <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, 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. Observez plutôt les figures suivantes.



Arborescence sous Eclipse avec XSLT.

localhost:8080/test/testTransformXsl.jsp			
Id	Auteur	Titre	Contenu
1	Pierre	Foo...	...bar !
27	Paul	Bientôt un LdZ J2EE !	Woot ?
102	Jacques	Coyote court toujours	Bip bip !

Rendu transformation XSLT.

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.

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

- La bibliothèque XML de la JSTL s'appuie sur la technologie XPath.
- On analyse une source XML avec `<x:parse>`.
- On affiche le contenu d'une variable ou d'un noeud avec `<x:out>`.
- On réalise un test avec `<x:if>` ou `<x:choose>`.
- On réalise une boucle avec `<x:forEach>`.
- On applique une transformation XSLT avec `<x:transform>`.

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 entraîner !

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 les pages JSP !

Une fois que c'est fait, créez pour commencer un document XML à la racine de votre projet, sous le répertoire **WebContent**. 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>Développez 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>Découvrez 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 à 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 prenez 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 existants 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.

Cette page devra également être placée à la racine du projet, sous le répertoire **WebContent**, et sera donc accessible via l'URL http://localhost:8080/jstl_exo2/rapportInventaire.jsp.

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 :

Code : JSP - rapportInventaire.jsp

```
<%@ page pageEncoding="UTF-8" %>
<%@ 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 nœud "livre". --%>
                <x:forEach var="livre" select="$doc/inventaire/livre">
                    <%-- Affichage du titre du livre récupéré. --%>
                    <li><x:out select="$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 nœud "livre" dont le contenu du nœud "stock" est inférieur au contenu du nœud "minimum". --%>
                <x:forEach var="livre" select="$doc/inventaire/livre[stock < minimum]">
                    <%-- Affichage des titres, stocks et minimaux du livre récupéré. --%>
                    <li><x:out select="$livre/titre" /> : <x:out select="$livre/stock" /> livres en stock (limite avant alerte : </li>
                </x:forEach>
            </ul>
        </div>
    </body>

```

```
<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 d'un 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 !

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.

Reprendons 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 pageEncoding="UTF-8" %>
<%@ 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}
            ${param.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

```
<%@ page pageEncoding="UTF-8" %>
<%@ 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}
            ${param.auteur}
        </p>
        <p>
            Récupération du bean :
            ${coyote.prenom}
            ${coyote.nom}
        </p>
        <p></p>
    </body>
</html>
```

```

        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 type primitifs en dur au sein de votre code, et de les regrouper sous forme de constantes en début de classe afin d'y centraliser les données.

Reprendons par exemple notre servlet d'exemple, où vous pouvez voir aux lignes 42 à 45 et 48 des String initialisées 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 librairie 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 );
}
}
}

```

Les lignes 20, 24 à 25 et 29 à 32, bien qu'elles contiennent des `String` et `int` en dur, correspondent simplement à l'initialisation des données d'exemple que nous transmettons à notre JSP : ce sont des données "externes". Dans le cas d'une application réelle, 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 dans cet exemple.

En ce qui concerne les `String` initialisées en dur, 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 librairie 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 constante 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 bibliothèque 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 : ce sera fait de manière transparente ! 😊

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



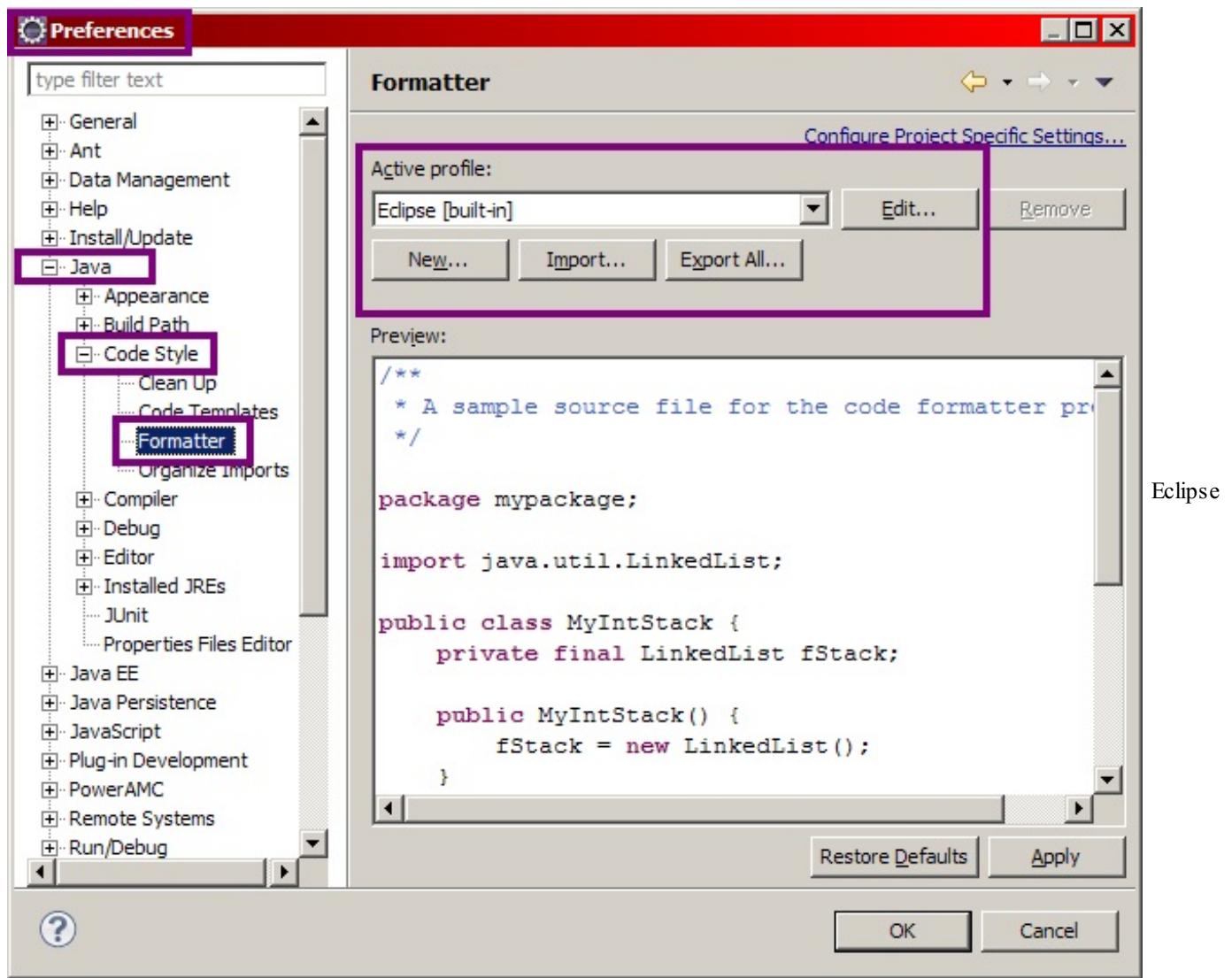
Nous n'en avons pas l'utilité ici, mais sachez qu'il est possible, avec ce même système, d'inclure automatiquement un fichier en fin de page : il faut pour cela préciser le fichier à inclure au sein d'une balise **<include-coda>**, et non plus **<include-prelude>** comme nous l'avons fait dans notre 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 à la figure suivante.

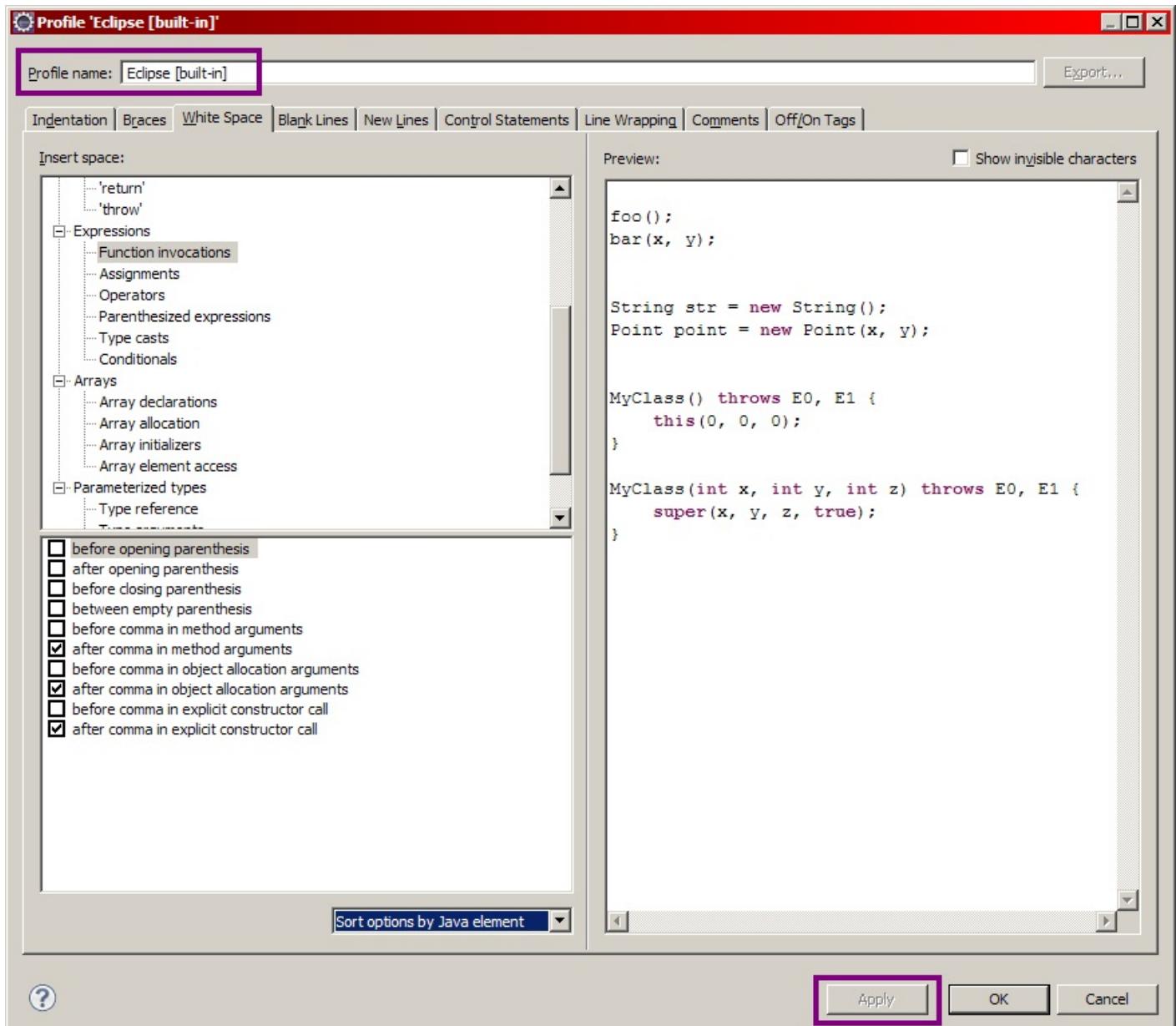


format tool.

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 (voir la figure suivante).



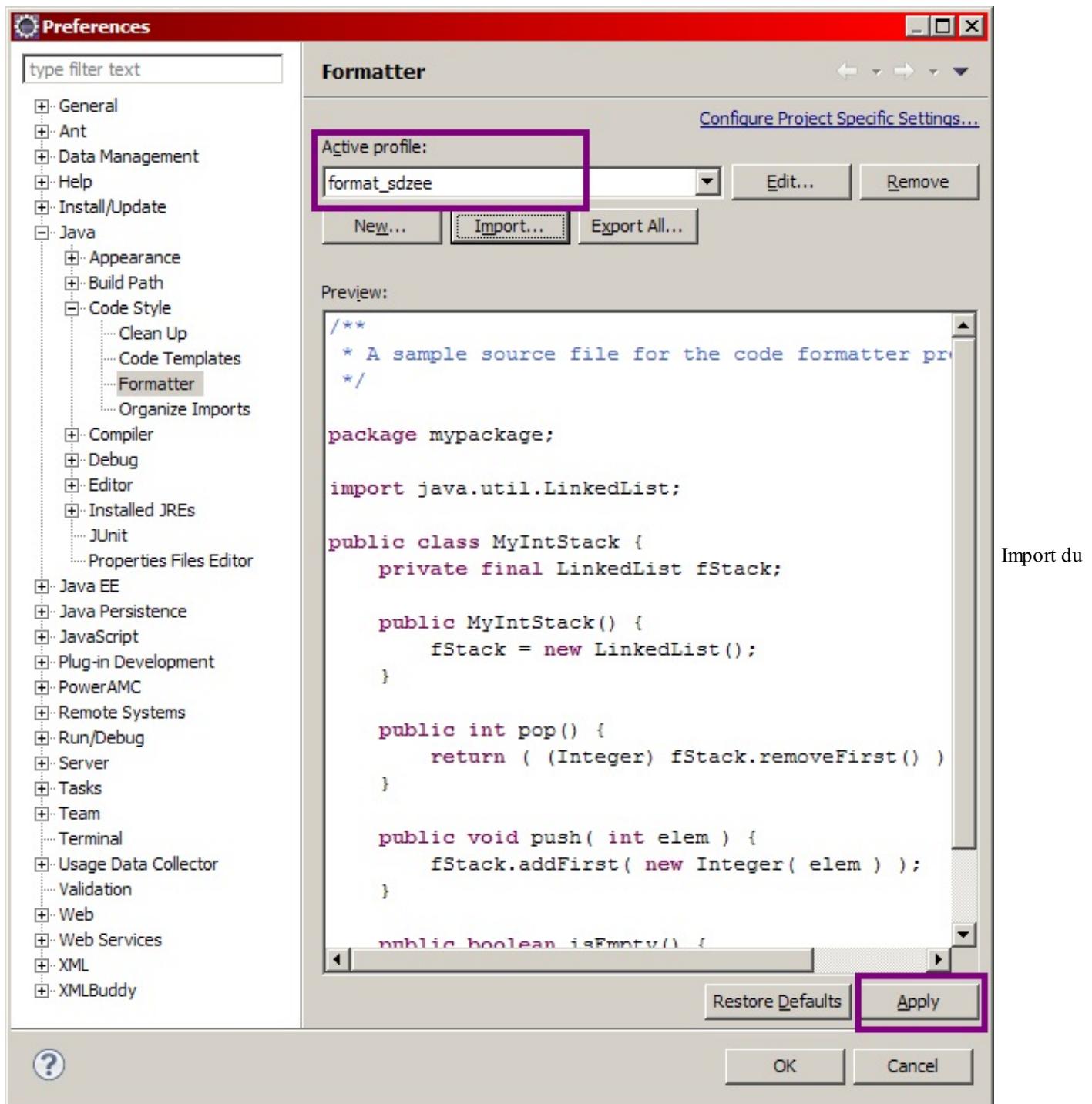
Options de formatage du code.

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, comme indiqué à la figure suivante.



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 utilisez tout simplement le raccourci clavier **Ctrl + Maj + F**. Prenons pour exemple le code de notre servlet **Test** (voir la figure suivante).

```

package com.eduvia.services.servGener;
import java.io.IOException;
public class Test extends HttpServlet {
    public static final String ATT_MESSAGE = "mess";
    public static final String ATT_NOM = "nom";
    public static final String ATT_LISTE = "list";
    public static final String ATT_JOUR = "jour";
    public static final String ATT_DATE = "date";
    public void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        /* Cohérence et initialisation du message. */
        String message = "Message transmis de la servlet à la JSP." ;
        /* Cohérence du nom et initialisation de ses propriétés */
        HttpSession session = new HttpSession();
        premierJour.set("Dagen") ;
        premierJour.set("Midi E.") ;
        /* Cohérence de l'ordre d'insertion de quatre éléments */
        List<Date> premiereListe = new ArrayList<Date>();
        premiereListe.add(27.32);
        premiereListe.add(17.17);
        premiereListe.add(128.14);
        premiereListe.add(6.11);
        /* On utilise une liste Date pour manipuler les dates, pour deux
        raisons : - c'est vraiment plus simple et pratique que de travailler
        avec des objets Date ou Calendar ; - c'est (probablement) un bon
        standard de l'RFI Java.
        */
        Date date = new Date();
        Integer jourNum = dt.getDays(date));
        /* Stockage du message, du bean et de la liste dans l'objet request */
        request.setAttribute(ATT_MESSAGE, message);
        request.setAttribute(ATT_NOM, nom);
        request.setAttribute(ATT_LISTE, premiereListe);
        request.setAttribute(ATT_JOUR, jourNum);
        /* Transmission de la paire d'objets request/response à notre JSP */
        this.getRequestDispatcher("index.jsp").forward(request, response);
    }
}

```

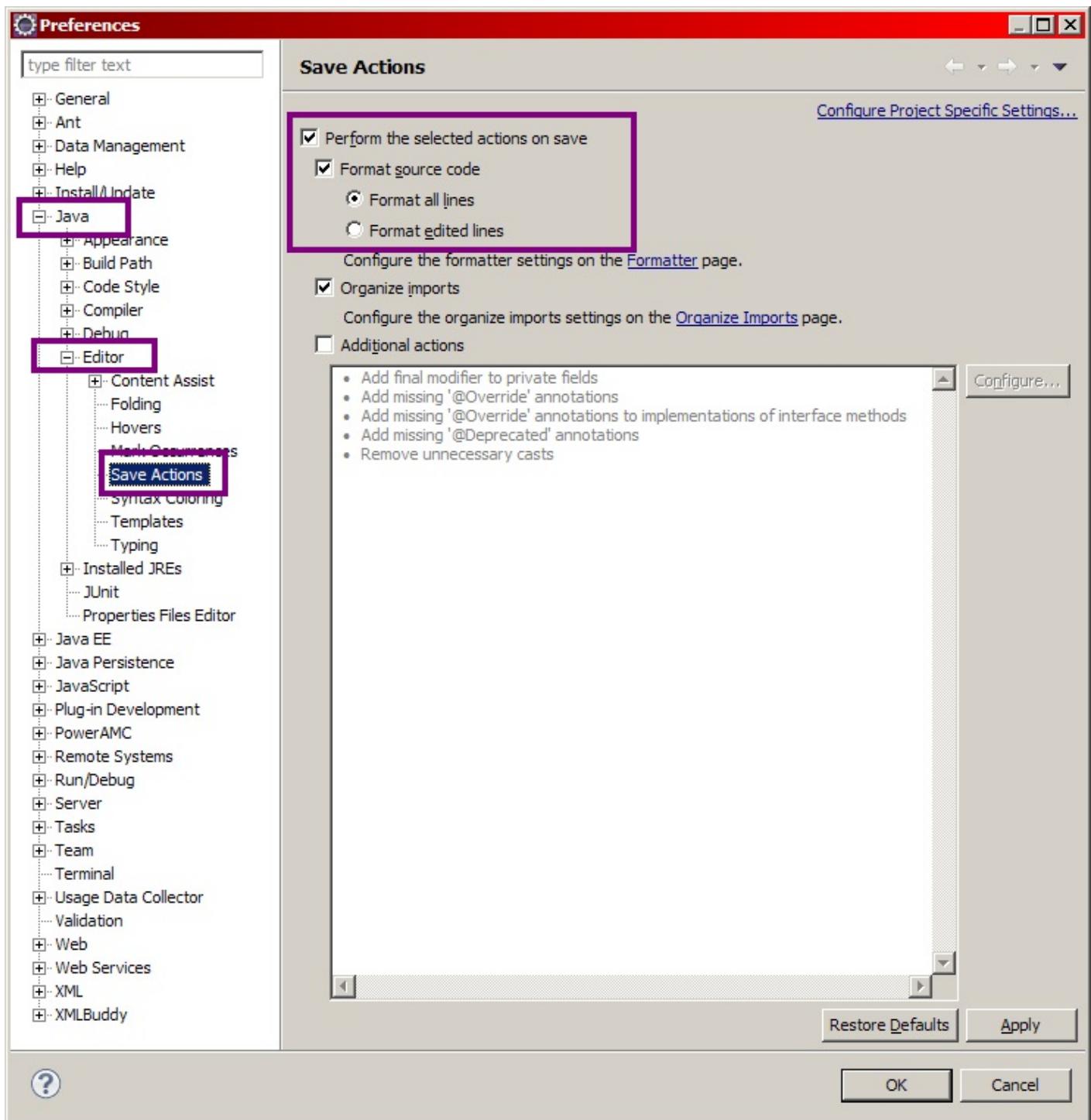
Rendu du formatage de la source.

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 chaque fois que nous effectuons des modifications dans le code source, afin de conserver un formatage parfait. Comme nous sommes fainéants, nous allons demander à Eclipse d'y penser pour nous ! Rendez-vous dans le menu Window > Preferences > Java > Editor > Save Actions, comme c'est indiqué sur la figure suivante.



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 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-mêmes 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 sont 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.

- La JSTL nous ouvre la porte aux fonctionnalités jusque là uniquement réalisables avec des scriptlets.
- Mettre en place des constantes permet de clarifier le code d'une classe et de simplifier sa maintenance.
- Eclipse peut prendre en charge pour vous le formatage et l'indentation de votre code, ainsi que la gestion automatique des imports.
- Cette prise en charge est automatisable, vous permettant ainsi de vous libérer de cette contrainte et de vous concentrer sur l'utile.
- La documentation est indispensable, à condition qu'elle soit à jour.

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 soyez à l'aise avec le système des balises de la JSTL. Je vous demande de :

- sécuriser l'affichage des données saisies par l'utilisateur contre les failles XSS, 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 ;
- isoler la partie du formulaire responsable de la création d'un client dans une page JSP à part, et modifier les deux formulaires existants pour qu'ils incluent tous deux ce fichier à la place du code actuellement dupliqué ;
- 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 d'un client avec erreurs (figure suivante).

The screenshot shows a web browser window with the URL `localhost:8080/tp2/creationClient?nomClient=`. Below the URL bar, there is a blue rectangular box containing two purple underlined links: Créer un nouveau client and Créer une nouvelle commande. At the bottom of the page, there is an error message in yellow text: *Erreur - Vous n'avez pas rempli tous les champs obligatoires. Cliquez ici pour accéder au formulaire de création d'un client.*

Création d'un client sans erreur (figure suivante).

The screenshot shows a browser window with the URL `localhost:8080/tp2/creationClient?nomClient=`. The page contains two links: [Créer un nouveau client](#) and [Créer une nouvelle commande](#). Below these links, a message in orange text says *Client créé avec succès !*. Underneath the message, there are five input fields with placeholder text: Nom : Coyote, Prénom :, Adresse : Pékin, Chine, Numéro de téléphone : 123456789, and Email :.

Création d'une commande avec erreurs (figure suivante).

The screenshot shows a browser window with the URL `localhost:8080/tp2/creationCommande?nomC`. The page contains two links: [Créer un nouveau client](#) and [Créer une nouvelle commande](#). Below these links, an error message in orange text reads *Erreur - Vous n'avez pas rempli tous les champs obligatoires.* and *Cliquez ici pour accéder au formulaire de création d'une commande.*

Création d'une commande sans erreur (figure suivante).

The screenshot shows a browser window with the URL `localhost:8080/tp2/creationCommande?nomC`. The page contains two links: [Créer un nouveau client](#) and [Créer une nouvelle commande](#). Below these links, a message in orange text says *Commande créée avec succès !*. Underneath the message, there are ten input fields with placeholder text: Client, Nom : Coyote, Prénom :, Adresse : Pékin, Chine, Numéro de téléphone : 123456789, Email :, Commande, Date : 21/06/2012 15:12:30, Montant : 123.45, Mode de paiement : CB, Statut du paiement :, Mode de livraison : La poste, and Statut de la livraison :.

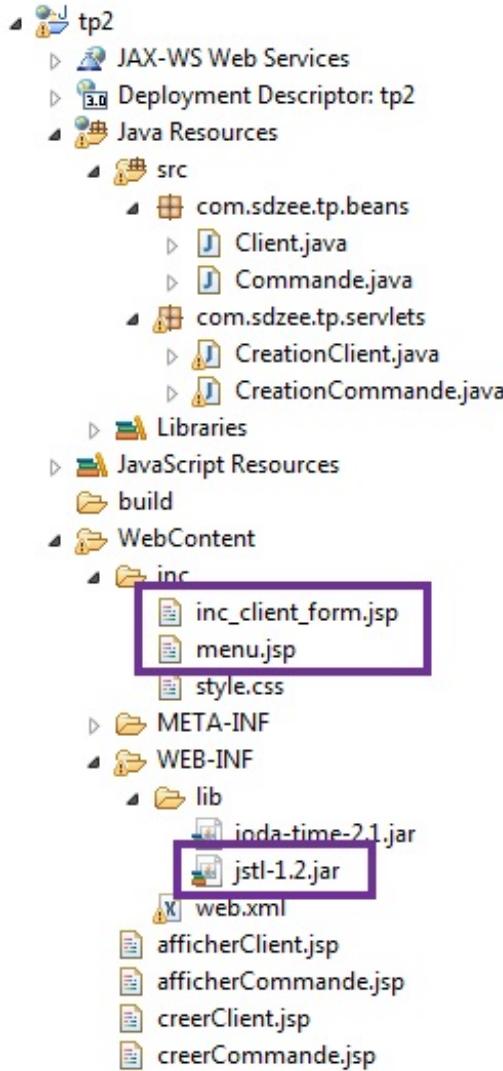
Conseils

Utilisation de la JSTL

Je vous donne tout de même quelques pistes pour vous guider, mais vous devriez être capables de vous en sortir 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 l'isolement du formulaire de création d'un client, même solution : il vous suffit de déplacer le code dans une page JSP que vous pouvez par exemple nommer `inc_client_form.jsp` et placer dans le répertoire `/inc`, et d'utiliser la balise `<c:import>` pour l'inclure aux deux formulaires existant ;
- 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 à la figure suivante l'allure de l'arborescence que vous devriez obtenir une fois le TP terminé.



Vous pouvez remarquer en encadré les trois nouveaux fichiers intervenant dans votre projet : le fichier jar de la JSTL, la page `menu.jsp` et la page `inc_client_form.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 de 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

Servlet gérant le formulaire de création d'un 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 d'une 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";
    ...
}

```

```
"emailClient";\n\n    public static final String CHAMP_DATE =\n\"dateCommande\";\n    public static final String CHAMP_MONTANT =\n\"montantCommande\";\n    public static final String CHAMP_MODE_PAIEMENT =\n\"modePaiementCommande\";\n    public static final String CHAMP_STATUT_PAIEMENT =\n\"statutPaiementCommande\";\n    public static final String CHAMP_MODE_LIVRAISON =\n\"modeLivraisonCommande\";\n    public static final String CHAMP_STATUT_LIVRAISON =\n\"statutLivraisonCommande\";\n\n    public static final String ATT_COMMANDE = \"commande\";\n    public static final String ATT_MESSAGE = \"message\";\n    public static final String ATT_ERREUR = \"erreur\";\n\n    public static final String FORMAT_DATE = \"dd/MM/yyyy\nHH:mm:ss\";\n\n    public static final String VUE =\n\"/afficherCommande.jsp\";\n\n    public void doGet( HttpServletRequest request,\nHttpServletRequest response ) throws ServletException, IOException {\n        /*\n        * Récupération des données saisies, envoyées en tant que paramètres\n        de\n        * la requête GET générée à la validation du formulaire\n        */\n        String nom = request.getParameter( CHAMP_NOM );\n        String prenom = request.getParameter( CHAMP_PRENOM );\n        String adresse = request.getParameter( CHAMP_ADRESSE );\n        String telephone = request.getParameter( CHAMP_TELEPHONE );\n        String email = request.getParameter( CHAMP_EMAIL );\n\n        /* Récupération de la date courante */\n        DateTime dt = new DateTime();\n        /* Conversion de la date en String selon le format choisi\n        */\n        DateTimeFormatter formatter = DateTimeFormat.forPattern(\nFORMAT_DATE );\n        String date = dt.toString( formatter );\n        double montant;\n        try {\n            /* Récupération du montant */\n            montant = Double.parseDouble( request.getParameter(\nCHAMP_MONTANT ) );\n        } catch ( NumberFormatException e ) {\n            /* Initialisation à -1 si le montant n'est pas un\n            nombre correct */\n            montant = -1;\n        }\n        String modePaiement = request.getParameter(\nCHAMP_MODE_PAIEMENT );\n        String statutPaiement = request.getParameter(\nCHAMP_STATUT_PAIEMENT );\n        String modeLivraison = request.getParameter(\nCHAMP_MODE_LIVRAISON );\n        String statutLivraison = request.getParameter(\nCHAMP_STATUT_LIVRAISON );\n\n        String message;\n        boolean erreur;\n        /*\n        * Initialisation du message à afficher : si un des champs\n        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

Page de création d'un client :

Code : JSP - /creerClient.jsp

```

<%@ page pageEncoding="UTF-8" %>
<%@ 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="" />
    </head>

```

```

</head>
<body>
    <c:import url="/inc/menu.jsp" />
    <div>
        <form method="get" action="

```

Page de création d'une commande :

Code : JSP - /creerCommande.jsp

```

<%@ page pageEncoding="UTF-8" %>
<%@ 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="" />
    </head>
    <body>
        <c:import url="/inc/menu.jsp" />
        <div>
            <form method="get" action="

```

```

        <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 contenant le fragment de formulaire :

Code : JSP - /inc/inc_client_form.jsp.jsp

```

<%@ page pageEncoding="UTF-8" %>
<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">Prénom </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 />

```

Page d'affichage d'un client :

Code : JSP - /afficherClient.jsp

```

<%@ page pageEncoding="UTF-8" %>
<%@ 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="" />
</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="${ client.nom }"/></p>
            <p>Prénom : <c:out value="${ 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>
        </c:if>
    </div>
</body>
</html>

```

Page d'affichage d'une commande :

Code : JSP - /afficherCommande.jsp

```

<%@ page pageEncoding="UTF-8" %>
<%@ 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="" />
    </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="${ commande.client.nom }" /></p>
                <p>Prénom : <c:out value="${ 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>
            </c:if>
        </div>
    </body>

```

```
</html>
```

Page contenant le nouveau menu :

Code : JSP - /inc/menu.jsp

```
<%@ page pageEncoding="UTF-8" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<div id="menu">
    <p><a href="
```

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 aimerais 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 application réelle.

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**.

N'oubliez pas les changements à effectuer sur le build-path et le serveur de déploiement, et pensez à ajouter le .jar de la JSTL à notre projet, ainsi que de créer 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 l'adresse mail de l'utilisateur ;
- un champ texte recueillant son mot de passe ;
- un champ texte recueillant la confirmation de son mot de passe ;
- un champ texte recueillant son nom d'utilisateur (optionnel).

Voici à la figure suivante un aperçu du design que je vous propose de mettre en place.

Inscription

Vous pouvez vous inscrire via ce formulaire.

Adresse email *	<input type="text"/>
Mot de passe *	<input type="password"/>
Confirmation du mot de passe *	<input type="password"/>
Nom d'utilisateur	<input type="text"/>
<input type="button" value="Inscription"/>	

Formulaire d'inscription

Si vous avez été assidus lors de vos premiers pas, vous devez vous souvenir que, dorénavant, nous placerons 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

```
<%@ page pageEncoding="UTF-8" %>
<!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="email">Adresse email <span
class="requis">*</span></label>
        <input type="text" id="email" name="email" value=""
size="20" maxlength="60" />
        <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="nom">Nom d'utilisateur</label>
        <input type="text" id="nom" name="nom" value=""
size="20" maxlength="20" />
        <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 rendre invisible ce qu'il contient pour 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 fatallement, 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 prétraitements depuis une servlet comme nous le faisons jusqu'à présent pour nos pages JSP. Nous pouvons donc rendre les fichiers CSS accessibles directement aux navigateurs en les plaçant dans un répertoire public de l'application.

En l'occurrence, ici j'ai placé cette feuille 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 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 pas par l'URL mais dans le corps d'une requête HTTP POST, l'utilisateur ne les verra donc pas dans la barre d'adresses de son navigateur.



Malgré leur invisibilité apparente, les données envoyées via la méthode POST restent aisément accessibles, et ne sont donc pas plus sécurisées qu'avec la méthode GET : 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 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 l'URL permettant de joindre cette servlet, c'est-à-dire l'URL que vous allez spécifier 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 du 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 indiqué à la figure suivante dans votre navigateur.

Inscription

Vous pouvez vous inscrire via ce formulaire.

Adresse email *

Mot de passe *

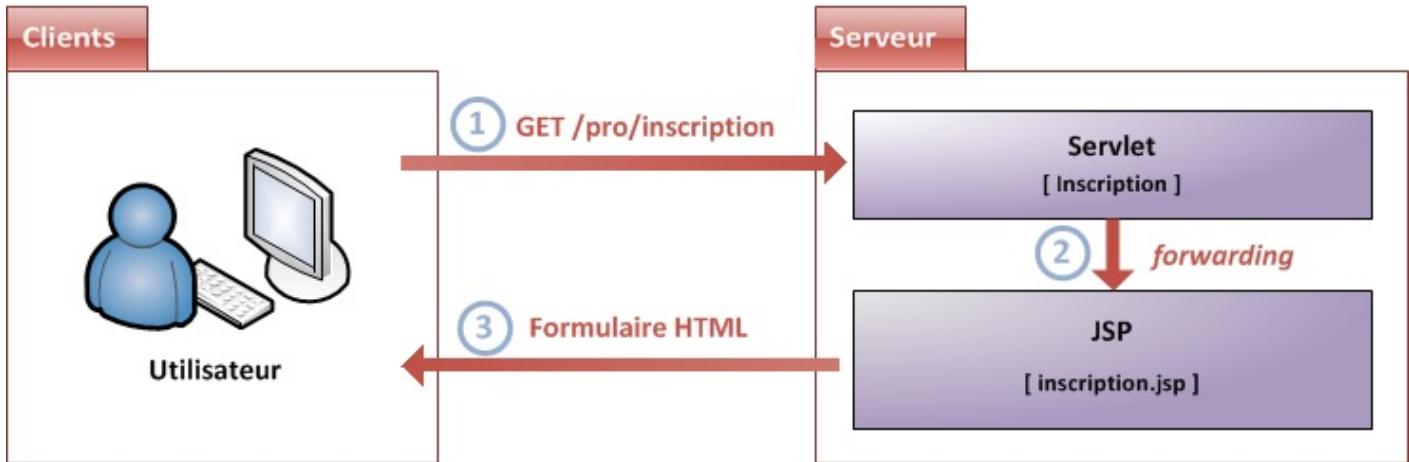
Confirmation du mot de passe *

Nom d'utilisateur

Inscription

Formulaire d'inscription

Et voici sous forme d'un schéma ce que nous venons de réaliser (voir la figure suivante).



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, puis cliquez sur le bouton d'inscription. Voici à la figure suivante la page que vous obtenez.

Etat HTTP 405 - La méthode HTTP POST n'est pas supportée par cette URL

type Rapport d'état

message La méthode HTTP POST n'est pas supportée par cette URL

description La méthode HTTP spécifiée n'est pas autorisée pour la ressource demandée (La méthode HTTP POST n'est pas supportée par cette URL).

Apache Tomcat/7.0.20

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 modifié de notre servlet :

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 Incription 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 envoyer les données du formulaire, il n'y aura plus d'erreur HTTP !

Par contre, la méthode `doPost()` étant vide, nous obtenons bien évidemment une 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 **email** n'est pas vide et qu'il contient une adresse mail valide ;
- 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 **nom**, s'il est rempli, contient au moins 3 caractères.

Nous allons confier ces tâches à trois méthodes distinctes :

- une méthode `validationEmail()`, chargée de valider l'adresse mail saisie ;
- une méthode `validationMotsDePasse()`, chargée de valider les mots de passe saisis ;
- une méthode `validationNom()`, chargée de valider le nom d'utilisateur saisi.

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 Incription extends HttpServlet {
    public static final String VUE = "/WEB-INF/inscription.jsp";
    public static final String CHAMP_EMAIL = "email";
    public static final String CHAMP_PASS = "motdepasse";
    public static final String CHAMP_CONF = "confirmation";
    public static final String CHAMP_NOM = "nom";

    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 email = request.getParameter( CHAMP_EMAIL );
        String motDePasse = request.getParameter( CHAMP_PASS );
        String confirmation = request.getParameter( CHAMP_CONF );
        String nom = request.getParameter( CHAMP_NOM );

        try {
            validationEmail( email );
            validationMotsDePasse( motDePasse, confirmation );
            validationNom( nom );
        } catch (Exception e) {
            /* Gérer les erreurs de validation ici. */
        }
    }

    private void validationEmail( String email ) throws Exception{}
    private void validationMotsDePasse( String motDePasse, String
confirmation ) throws Exception{}
    private void validationNom( String nom ) throws Exception{}
}

```

La partie en charge de la récupération des champs du formulaire se situe aux lignes 24 à 27 : il s'agit tout simplement d'appels à la méthode `request.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.Incription

```

...
/** 
 * Valide l'adresse mail saisie.
*/
private void validationEmail( String email ) throws Exception {
    if ( email != null && email.trim().length() != 0 ) {
        if ( !email.matches(
"([^.@]+)(\\.[^.@]+)*@[^.@]+\\.)+([^.@]+)" ) ) {
            throw new Exception( "Merci de saisir une adresse mail
valide." );
        }
    } else {
        throw new Exception( "Merci de saisir une adresse mail." );
    }
}

```

```

    }

    /**
     * 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.trim().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 le nom d'utilisateur saisi.
     */
    private void validationNom( String nom ) throws Exception {
        if ( nom != null && nom.trim().length() < 3 ) {
            throw new Exception( "Le nom d'utilisateur doit contenir au
moins 3 caractères." );
        }
    }
}

```

 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.

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 :

 Que faire de ces exceptions envoyées ?

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 Incription extends HttpServlet {
    public static final String VUE = "/WEB-INF/inscription.jsp";
    public static final String CHAMP_EMAIL = "email";
    public static final String CHAMP_PASS = "motdepasse";
    public static final String CHAMP_CONF = "confirmation";
    public static final String CHAMP_NOM = "nom";
    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 email = request.getParameter( CHAMP_EMAIL );
    String motDePasse = request.getParameter( CHAMP_PASS );
    String confirmation = request.getParameter( CHAMP_CONF );
    String nom = request.getParameter( CHAMP_NOM );

    /* Validation du champ email. */
    try {
        validationEmail( email );
    } catch ( Exception e ) {
        erreurs.put( CHAMP_EMAIL, 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 nom. */
    try {
        validationNom( nom );
    } catch ( Exception e ) {
        erreurs.put( CHAMP_NOM, 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 );
}

...
}

```

Analysez bien les modifications importantes du code, afin de bien comprendre ce qui intervient dans ce processus de gestion des exceptions :

- chaque appel à une méthode de validation d'un champ est entouré 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 **email**, **motdepasse**, **confirmation** et **nom** :

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

```

<%@ page pageEncoding="UTF-8" %>
<!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="email">Adresse email <span
                    class="requis">*</span></label>
                <input type="email" id="email" name="email" value=""
                    size="20" maxlength="60" />
                <span class="erreur">${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">${erreurs['motdepasse']}</span>
                <br />
            </fieldset>
        </form>
    </body>
</html>

```

```

        <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="nom">Nom d'utilisateur</label>
        <input type="text" id="nom" name="nom" value="" size="20" maxlength="20" />
        <span class="erreur">${erreurs['nom']}</span>
        <br />

        <input type="submit" value="Inscription" class="sansLabel" />
        <br />
    </fieldset>
</form>
</body>
</html>

```

Vous retrouvez aux lignes 17, 22, 27 et 32 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. 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 maintenant faire le test : remplissez votre formulaire avec des données erronées (une adresse mail invalide, un nom d'utilisateur trop court ou des mots de passe différents, par exemple) et contemplez le résultat ! À la figure suivante, le rendu attendu lorsque vous entrez un nom d'utilisateur trop court.

Inscription

Vous pouvez vous inscrire via ce formulaire.

Adresse email *

Mot de passe *

Confirmation du mot de passe *

Nom d'utilisateur

Le nom d'utilisateur doit contenir au moins 3 caractères.

Inscription

Erreur de validation du formulaire

2. Réafficher 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 réafficher 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 ?

Bien... puisque vous semblez amnésiques et sceptiques, faisons comme si de rien n'était, et réaffichons 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**, aux lignes 16 et 31 :

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

```

<%@ page pageEncoding="UTF-8" %>
<!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="email">Adresse email <span
                    class="requis">*</span></label>
                    <input type="email" id="email" name="email"
                    value="${param.email}" size="20" maxlength="60" />
                    <span class="erreur">${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">${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="nom">Nom d'utilisateur</label>
                    <input type="text" id="nom" name="nom"
                    value="${param.nom}" size="20" maxlength="20" />
                    <span class="erreur">${erreurs['nom']}</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, ainsi que vous pouvez le constater à la figure suivante.

Inscription

Vous pouvez vous inscrire via ce formulaire.

Adresse email *	<input type="text" value="test@test.com"/>
Mot de passe *	<input type="password"/>
Confirmation du mot de passe *	<input type="password"/>
Nom d'utilisateur	<input type="text" value="te"/> Le nom d'utilisateur doit contenir au moins 3 caractères.
<input type="button" value="Inscription"/>	

Erreur de validation du formulaire avec affichage des données saisies

En apparence ça tient la route, mais je vous ai lourdement avertis : **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 (voir la figure suivante).

Inscription

Vous pouvez vous inscrire via ce formulaire.

Adresse email *	<input type="text" value="test@test.com"/>
Mot de passe *	<input type="password"/>
Confirmation du mot de passe *	<input type="password"/>
Nom d'utilisateur	<input maxlength="20" size="20" type="text" value="Bip bip !"/> Bip bip !" size="20" maxlength="20">
<input type="button" value="Inscription"/>	

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 puis 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. 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, observez bien les lignes 17 et 32 :

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

```
<%@ page pageEncoding="UTF-8" %>
<%@ 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="email">Adresse email <span
class="requis">*</span></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 />

                <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="nom">Nom d'utilisateur</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 />

                <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 la 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 le résultat affiché à la figure suivante.

Inscription

Vous pouvez vous inscrire via ce formulaire.

Adresse email *	<input type="text" value="test@test.com"/>
Mot de passe *	<input type="password"/>
Confirmation du mot de passe *	<input type="password"/>
Nom d'utilisateur	">Bip bip !
	<input type="button" value="Inscription"/>

Erreurs 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, à la ligne 39 dans le code suivant :

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

```
<%@ page pageEncoding="UTF-8" %>
<%@ 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="email">Adresse email <span class="requis">*</span></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 />
                <label for="motdepasse">Mot de passe <span class="requis">*</span></label>
            </fieldset>
        </form>
    </body>
</html>
```

```

<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="nom">Nom d'utilisateur</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 />

<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**, sinon on utilise le style **erreur**. En effet, j'en ai 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 aux figures suivantes le résultat final en cas de succès et d'erreur.

Inscription

Vous pouvez vous inscrire via ce formulaire.

Adresse email *	<input type="text" value="test@test.com"/>
Mot de passe *	<input type="password"/>
Confirmation du mot de passe *	<input type="password"/>
Nom d'utilisateur	<input type="text" value="test"/>

<input type="button" value="Inscription"/>
--

Succès de l'inscription.

Succès de la validation du formulaire

Inscription

Vous pouvez vous inscrire via ce formulaire.

Adresse email *

Mot de passe *

Confirmation du mot de passe *

Nom d'utilisateur

Le nom d'utilisateur doit contenir au moins 3 caractères.

Échec de l'inscription.

Erreurs dans la validation du formulaire

- Les pages placées sous /WEB-INF ne sont par défaut pas accessibles au public par une URL.
- La méthode d'envoi des données d'un formulaire se définit dans le code HTML via `<form method="...">`.
- Les données envoyées via un formulaire sont accessibles côté serveur via des appels à `request.getParameter("nom_du_champ")`.
- La validation des données, lorsque nécessaire, peut se gérer simplement avec les exceptions et des interceptions via des blocs `try / catch`.
- Le renvoi de messages à l'utilisateur peut se faire via une simple Map placée en tant qu'attribut de requête.
- L'affichage de ces messages côté JSP se fait alors via de simples et courtes expressions EL.
- Il ne faut **jamais** afficher du contenu issu d'un utilisateur sans le sécuriser, afin d'éliminer le risque de failles XSS.
- Placer la validation et les traitements sur les données dans la servlet n'est pas une bonne solution, il faut trouver mieux.

Formulaires : à la mode MVC

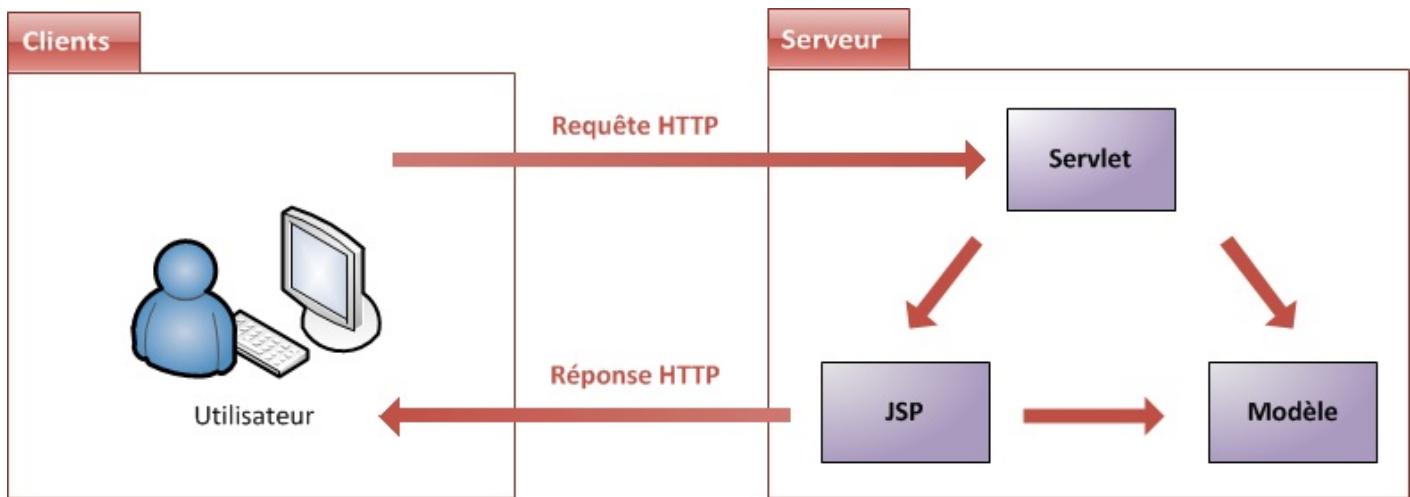
Le modèle MVC est très clair sur ce point : c'est le modèle qui doit s'occuper de traiter les données. Le contrôleur doit avoir pour unique but d'aiguiller les requêtes entrantes et d'appeler les éventuels traitements correspondants. Nous devons donc nous pencher sur la conception que nous venons de mettre en place afin d'en identifier les défauts, et de la rectifier dans le but 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...

Voici à la figure suivante le schéma représentant ce à 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` : `email`, `motDePasse` et `nom`. 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 email;
    private String motDePasse;
    private String nom;

    public void setEmail(String email) {
```

```

    this.email = email;
}
public String getEmail() {
    return email;
}

public void setMotDePasse(String motDePasse) {
    this.motDePasse = motDePasse;
}
public String getMotDePasse() {
    return motDePasse;
}

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

```

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. Un utilisateur final possède un mot de passe et point barre : la confirmation est une information temporaire propre à l'étape d'inscription uniquement ; il n'y a par conséquent aucun intérêt à la stocker dans le 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. Dans notre cas, cet objet va contenir :

1. les constantes identifiant les champs du formulaire ;
2. la chaîne **resultat** et la Map **erreurs** que nous avions mises en place dans la servlet ;
3. la logique de validation que nous avions utilisée dans la méthode `doPost()` de la servlet ;
4. 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.

1. Pour commencer, nous allons nommer ce nouvel objet **InscriptionForm**, le placer dans un nouveau package `com.sdzee.forms`, et y inclure les constantes dont nous allons avoir besoin :

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 IncriptionForm {
    private static final String CHAMP_EMAIL = "email";
    private static final String CHAMP_PASS = "motdepasse";
    private static final String CHAMP_CONF = "confirmation";
    private static final String CHAMP_NOM = "nom";

    ...
}

```

2. Nous devons ensuite y ajouter la chaîne **resultat** et la Map **erreurs** :

Code : Java - com.sdzee.forms.InscriptionForm

```
...  
  
private String resultat;  
private Map<String, String> erreurs = new HashMap<String,  
String>();  
  
public String getResultat() {  
    return resultat;  
}  
  
public Map<String, String> getErreurs() {  
    return erreurs;  
}  
  
...
```

3. Nous ajoutons alors la méthode principale, contenant la logique de validation :

Code : Java - com.sdzee.forms.InscriptionForm

```
...  
  
public Utilisateur inscrireUtilisateur( HttpServletRequest request )  
{  
    String email = getValeurChamp( request, CHAMP_EMAIL );  
    String motDePasse = getValeurChamp( request, CHAMP_PASS );  
    String confirmation = getValeurChamp( request, CHAMP_CONF );  
    String nom = getValeurChamp( request, CHAMP_NOM );  
  
    Utilisateur utilisateur = new Utilisateur();  
  
    try {  
        validationEmail( email );  
    } catch ( Exception e ) {  
        setErreur( CHAMP_EMAIL, e.getMessage() );  
    }  
    utilisateur.setEmail( email );  
  
    try {  
        validationMotsDePasse( motDePasse, confirmation );  
    } catch ( Exception e ) {  
        setErreur( CHAMP_PASS, e.getMessage() );  
        setErreur( CHAMP_CONF, null );  
    }  
    utilisateur.setMotDePasse( motDePasse );  
  
    try {  
        validationNom( nom );  
    } catch ( Exception e ) {  
        setErreur( CHAMP_NOM, e.getMessage() );  
    }  
    utilisateur.setNom( nom );  
  
    if ( erreurs.isEmpty() ) {  
        resultat = "Succès de l'inscription.";  
    } else {  
        resultat = "Échec de l'inscription.";  
    }  
}
```

```

    return utilisateur;
}

...

```

4. Pour terminer, nous mettons en place les méthodes de validation et les deux méthodes utilitaires nécessaires au bon fonctionnement de la logique que nous venons d'écrire :

Code : Java - com.sdzee.forms.InscriptionForm

```

...
private void validationEmail( String email ) throws Exception {
    if ( email != null ) {
        if ( !email.matches(
            "([^.@]+)(\\.[^.@]+)*@[^.@]+\\" ) ) {
            throw new Exception( "Merci de saisir une adresse mail valide." );
        }
    } else {
        throw new Exception( "Merci de saisir une adresse mail." );
    }
}

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 validationNom( String nom ) throws Exception {
    if ( nom != null && nom.length() < 3 ) {
        throw new Exception( "Le nom d'utilisateur doit contenir au moins 3 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.trim();
    }
}

```

Encore une fois, prenez bien le temps d'analyser 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 4, 14 et 26 du dernier code 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` du troisième code, 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 ;
- toujours dans le troisième code, 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 16, 24 et 31).

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 au traitement et à la 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 un 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, malheureusement très courante sur le web. Dans énormément d'exemples de servlets, vous pourrez trouver ce genre de code :

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, quel que 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, ce 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 accédons aux données saisies par l'utilisateur via l'objet implicite **param**.

Maintenant, la servlet transmet le bean et l'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

```
<%@ page pageEncoding="UTF-8" %>
<%@ 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="email">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 />

                <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="nom">Nom d'utilisateur</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>
            </fieldset>
        </form>
    </body>
</html>
```

```

<br />
<input type="submit" value="Inscription"
class="sansLabel" />
<br />

<p class="${empty form.erreurs ? 'succes' :
'erreur'}">${form.resultat}</p>
</fieldset>
</form>
</body>
</html>

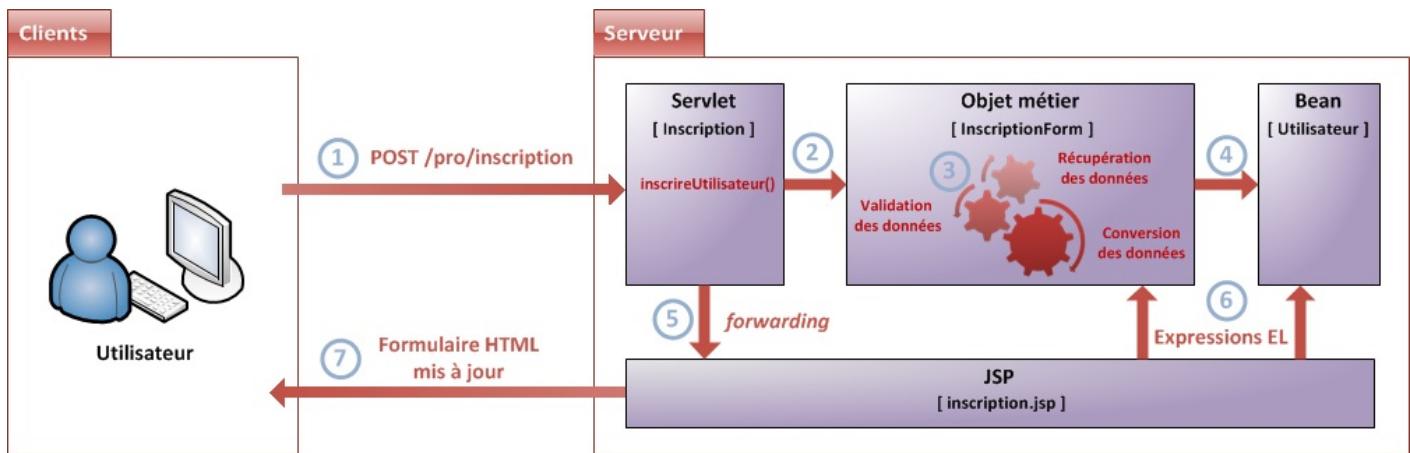
```

Les modifications apportées semblent donc mineures :

- 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 à la figure suivante un schéma de ce que nous avons réalisé.



Nous avons ainsi avec succès mis en place une architecture MVC pour le traitement de notre formulaire :

- les données saisies et envoyées par le client arrivent à la méthode `doPost()` de la servlet ;
 - celle-ci ordonne alors le contrôle des données reçues en appelant la méthode `inscrireUtilisateur()` de l'objet métier **InscriptionForm** ;
 - l'objet **InscriptionForm** effectue les traitements de validation de chacune des données reçues ;
 - il les enregistre par la même occasion dans le bean **Utilisateur** ;
 - la méthode `doPost()` récupère enfin les deux objets du modèle, et les transmet à la JSP via la portée requête ;
 - 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 ;
 - pour finir, la JSP met à jour l'affichage du formulaire en se basant sur les nouvelles données.
- Il faut utiliser un bean pour stocker les données du formulaire.
 - Il faut déplacer la validation et le traitement des données dans un objet métier.
 - La servlet ne fait alors plus qu'aiguiller les données : contrôle > appels aux divers traitements > renvoi à la JSP.
 - La méthode `doGet()` s'occupe des requêtes GET, la méthode `doPost()` des requêtes POST. Tout autre usage est fortement déconseillé.
 - La page JSP accède dorénavant aux données directement à travers les objets du modèle mis en place, et non plus depuis la requête.

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 servlets 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. Au passage, vous allez en profiter pour appliquer la pratique que je vous avais communiquée lorsque nous avions découvert MVC : vous allez déplacer toutes vos JSP sous le répertoire **/WEB-INF**, et gérer leur accès entièrement depuis vos servlets. Je ne vous l'avais pas fait faire dans les premières étapes pour ne pas vous embrouiller, mais le moment est venu !

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 :

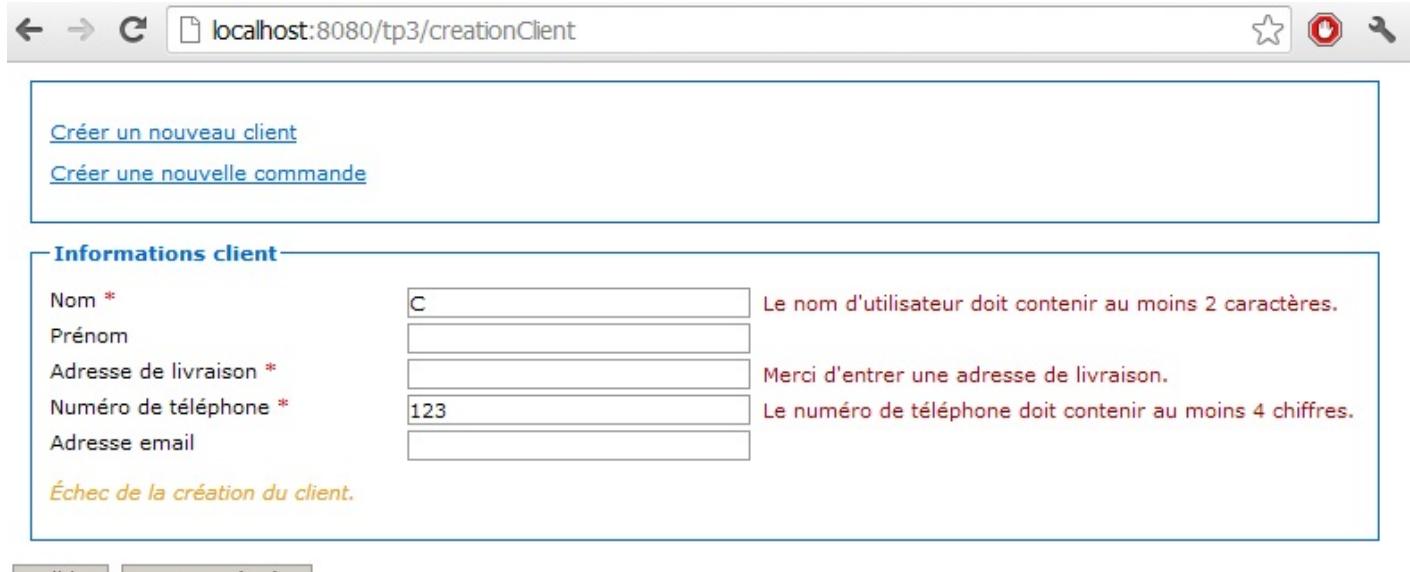
- en cas d'erreur lors de la validation (champs manquants ou erronés), vous devrez faire retourner l'utilisateur au formulaire de saisie en lui réaffichant - sans faille XSS ! - les données qu'il a saisies, et en précisant un message signalant les erreurs sur chaque champ qui pose problème ;
- en cas 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

Voici aux figures suivantes quelques exemples de rendu.

Création d'un client avec erreurs :



The screenshot shows a browser window with the URL `localhost:8080/tp3/creationClient`. The page title is "Créer un nouveau client". Below it is a link "Créer une nouvelle commande". The main section is titled "Informations client". It contains the following fields with validation errors:

- Nom ***: An empty input field with the error message "Le nom d'utilisateur doit contenir au moins 2 caractères".
- Prénom**: An empty input field.
- Adresse de livraison ***: An empty input field with the error message "Merci d'entrer une adresse de livraison".
- Numéro de téléphone ***: An input field containing "123" with the error message "Le numéro de téléphone doit contenir au moins 4 chiffres".
- Adresse email**: An empty input field.

A yellow box at the bottom left of the form area contains the text "Échec de la création du client."

At the bottom of the page are two buttons: "Valider" and "Remettre à zéro".

Création d'un client sans erreur :

The screenshot shows a browser window with the URL `localhost:8080/tp3/creationClient`. Inside the page, there are two blue links: "Créer un nouveau client" and "Créer une nouvelle commande". Below these links, a message in orange text says "Succès de la création du client.". Underneath, there are five input fields with the following values: Nom : Coyote, Prénom : , Adresse : Pékin, Chine, Numéro de téléphone : 123456789, and Email : .

Création d'une commande avec erreurs :

The screenshot shows a browser window with the URL `localhost:8080/tp3/creationCommande`. It features two blue links: "Créer un nouveau client" and "Créer une nouvelle commande". Below these, there are two sections: "Informations client" and "Informations commande".

Informations client:

Nom *	Coyote
Prénom	
Adresse de livraison *	Pékin
Numéro de téléphone *	123456789A
Adresse email	

Validation errors are shown in red: "L'adresse de livraison doit contenir au moins 10 caractères." and "Le numéro de téléphone doit uniquement contenir des chiffres."

Informations commande:

Date *	15/08/2012 09:28:19
Montant *	123.45
Mode de paiement *	
Statut du paiement	
Mode de livraison *	La poste
Statut de la livraison	

A validation error message "Merci d'entrer un mode de paiement." is displayed next to the Mode de paiement field.

An orange message at the bottom left says "Échec de la création de la commande."

At the bottom, there are two buttons: "Valider" and "Remettre à zéro".

Création d'une commande sans erreur :

The screenshot shows a web browser window with the URL `localhost:8080/tp3/creationCommande`. The page displays a success message: "Succès de la création de la commande." Below this, it lists the details of the created command and its associated client. The client information includes: Nom : Coyote, Prénom : , Adresse : Pékin, Chine, Numéro de téléphone : 123456789, Email : . The command details include: Date : 21/06/2012 14:52:12, Montant : 123.45, Mode de paiement : CB, Statut du paiement : , Mode de livraison : La poste, and Statut de la livraison : . There are also links to "Créer un nouveau client" and "Créer une nouvelle commande".

Succès de la création de la commande.

Client

Nom : Coyote

Prénom :

Adresse : Pékin, Chine

Numéro de téléphone : 123456789

Email :

Commande

Date : 21/06/2012 14:52:12

Montant : 123.45

Mode de paiement : CB

Statut du paiement :

Mode de livraison : La poste

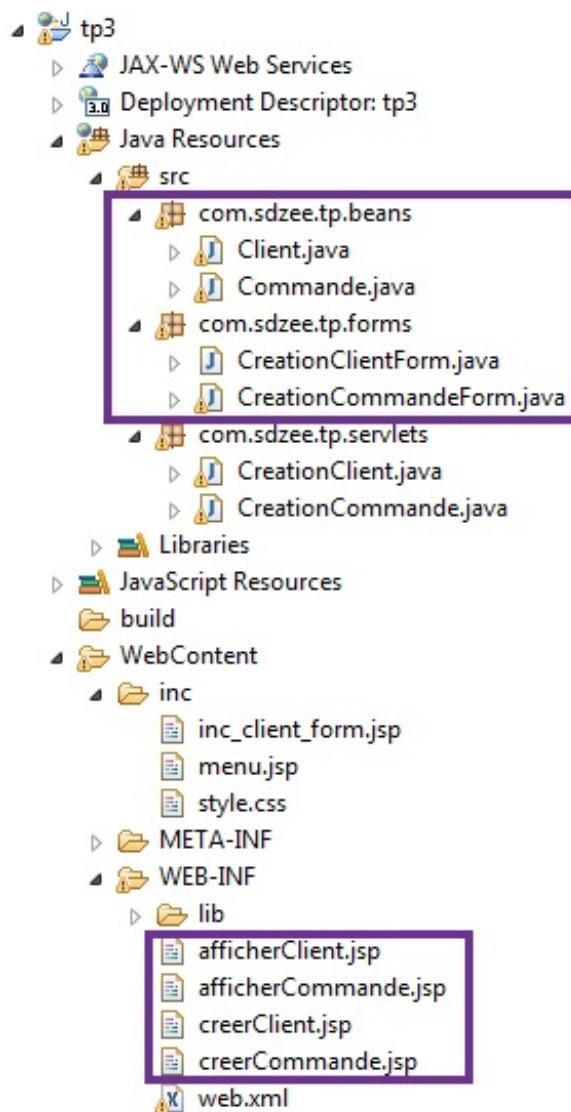
Statut de la livraison :

Conseils

Concernant le changement de méthode d'envoi de GET vers POST et le déplacement des JSP sous /WEB-INF, il vous suffit de bien penser à ce que cela va impliquer dans vos formulaires, dans vos servlets et dans votre menu ! Typiquement, vous allez devoir faire en sorte que vos servlets affichent les formulaires en cas de réception d'une requête GET, et traitent les données envoyées en cas de réception d'une requête POST. Côté formulaire, vous allez devoir modifier un attribut de la balise <form>... Et dans votre menu, vous allez devoir remplacer les URL des deux pages JSP par celles de leurs servlets respectives.

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 vérification 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" !

Voici à la figure suivante l'arborescence que vous êtes censés créer.



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 de 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

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() );
        }
    }

    private String getValeurChamp( HttpServletRequest request, String champ ) {
        String valeur = request.getParameter( champ );
        if ( valeur == null || valeur.trim().length() == 0 ) {
            return null;
        }
        return valeur;
    }

    private void validationNom( String nom ) {
        if ( nom == null || nom.trim().length() < 2 ) {
           erreurs.put( CHAMP_NOM, "Le nom doit contenir au moins 2 caractères" );
        }
    }

    private void validationPrenom( String prenom ) {
        if ( prenom == null || prenom.trim().length() < 2 ) {
            erreurs.put( CHAMP_PRENOM, "Le prénom doit contenir au moins 2 caractères" );
        }
    }

    private void validationAdresse( String adresse ) {
        if ( adresse == null || adresse.trim().length() < 5 ) {
            erreurs.put( CHAMP_ADRESSE, "L'adresse doit contenir au moins 5 caractères" );
        }
    }

    private void validationTelephone( String telephone ) {
        if ( telephone == null || telephone.trim().length() < 10 ) {
            erreurs.put( CHAMP_TELEPHONE, "Le numéro de téléphone doit contenir au moins 10 caractères" );
        }
    }

    private void validationEmail( String email ) {
        if ( email == null || !email.matches( "[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\.[a-zA-Z]{2,}" ) ) {
            erreurs.put( CHAMP_EMAIL, "L'e-mail n'est pas valide" );
        }
    }

    private void setErreur( String champ, String message ) {
        erreurs.put( champ, message );
    }
}
```

```
        }
        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 chiffres." );
            }
        } 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 =
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 ! À 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'erreurs 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_PAITEMENT );
    String statutPaiement = getValeurChamp( request,
CHAMP_STATUT_PAITEMENT );
    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_PAITEMENT, e.getMessage() );
    }
}
```

```
        commande.setModePaiement( modePaiement );

    try {
        validationStatutPaiement( statutPaiement );
    } catch ( Exception e ) {
        setErreur( CHAMP_STATUT_PAIMENT, e.getMessage() );
    }
    commande.setStatutPaiement( statutPaiement );

    try {
        validationModeLivraison( modeLivraison );
    } catch ( Exception e ) {
        setErreur( CHAMP_MODE_LIVRAISON, e.getMessage() );
    }
    commande.setModelLivraison( 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
paiement." );
    }
}

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

Servlet gérant la création d'un 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 = "/WEB-INF/afficherClient.jsp";
    public static final String VUE_FORM = "/WEB-INF/creerClient.jsp";

    public void doGet( HttpServletRequest request,
HttpServletResponse response ) throws ServletException, IOException
{
    /* À la réception d'une requête GET, simple affichage du formulaire */
    this.getServletContext().getRequestDispatcher( VUE_FORM )
).forward( request, response );
}

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 d'une 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 = "/WEB-INF/afficherCommande.jsp";
    public static final String VUE_FORM = "/WEB-INF/creerCommande.jsp";

    public void doGet( HttpServletRequest request,
HttpServletResponse response ) throws ServletException, IOException
{
    /* À la réception d'une requête GET, simple affichage du formulaire */
    this.getServletContext().getRequestDispatcher( VUE_FORM )
).forward( request, response );
}

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

Page contenant le menu :

Code : JSP - /inc/menu.jsp

```

<%@ page pageEncoding="UTF-8" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<div id="menu">
    <p><a href=<c:url value="/creationClient"/>">Créer un nouveau client</a></p>
    <p><a href=<c:url value="/creationCommande"/>">Créer une nouvelle commande</a></p>
</div>

```

Page contenant le fragment de formulaire :

Code : JSP - /inc/inc_client_form.jsp

```
<%@ page pageEncoding="UTF-8" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<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']}

```

Page de création d'un client :

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

```
<%@ page pageEncoding="UTF-8" %>
<%@ 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="" />
    </head>
    <body>
        <c:import url="/inc/menu.jsp" />
        <div>
            <form method="post" action="">
                <fieldset>
                    <legend>Informations client</legend>
                    <c:import url="/inc/inc_client_form.jsp" />
                </fieldset>
            </form>
        </div>
    </body>
</html>
```

```

        <p class="titre" style="text-align: center;">Résultat de la recherche
        <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 - /WEB-INF/creerCommande.jsp

```

<%@ page pageEncoding="UTF-8" %>
<%@ 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="" />
    </head>
    <body>
        <c:import url="/inc/menu.jsp" />
        <div>
            <form method="post" action="

```

```

name="modePaiementCommande" value="
class="erreur">>${form.erreurs['modePaiementCommande']}</span>
<br />

<label for="statutPaiementCommande">Statut du
paiement</label>
<input type="text" id="statutPaiementCommande"
name="statutPaiementCommande" value="
class="erreur">>${form.erreurs['statutPaiementCommande']}</span>
<br />

<label for="modeleLivraisonCommande">Mode de
livraison <span class="requis">*</span></label>
<input type="text" id="modeleLivraisonCommande"
name="modeleLivraisonCommande" value="
class="erreur">>${form.erreurs['modeleLivraisonCommande']}</span>
<br />

<label for="statutLivraisonCommande">Statut de
la livraison</label>
<input type="text" id="statutLivraisonCommande"
name="statutLivraisonCommande" value="
class="erreur">>${form.erreurs['statutLivraisonCommande']}</span>
<br />
</fieldset>
<p class="info">${ form.resultat }</p>
<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

```

<%@ page pageEncoding="UTF-8" %>
<%@ 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="" />
  </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>

```

```
</div>
</body>
</html>
```

Page d'affichage d'une commande :

Code : JSP - /afficherCommande.jsp

```
<%@ page pageEncoding="UTF-8" %>
<%@ 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="" />
    </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

```
<%@ page pageEncoding="UTF-8" %>
<%@ 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=">" size="20"
maxlength="60" />
                    <span class="erreur">${form.erreurs['email']}

```

Nous reprenons la même architecture que pour le système d'inscription : notre JSP exploite un objet **form** contenant les éventuels messages d'erreur, 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();

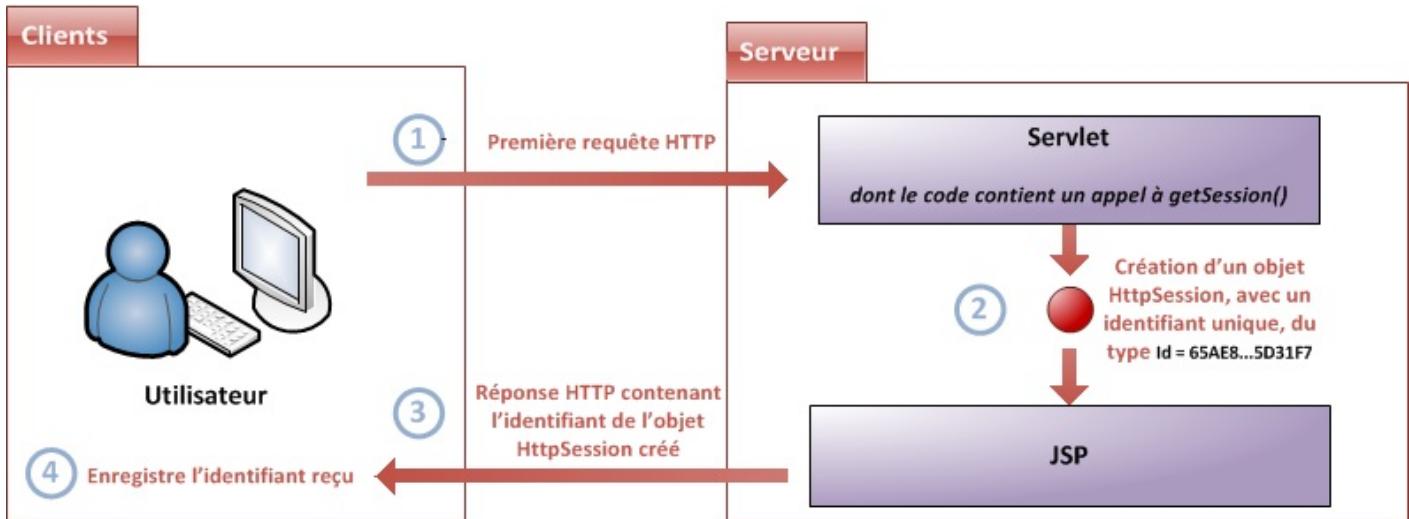
/* Mise en session d'une chaîne de caractères */
String exemple = "abc";
```

```

        session.setAttribute( "chaine", exemple );
        /* Récupération de l'objet depuis la session */
        String chaine = (String) session.getAttribute( "chaine" );
    
```

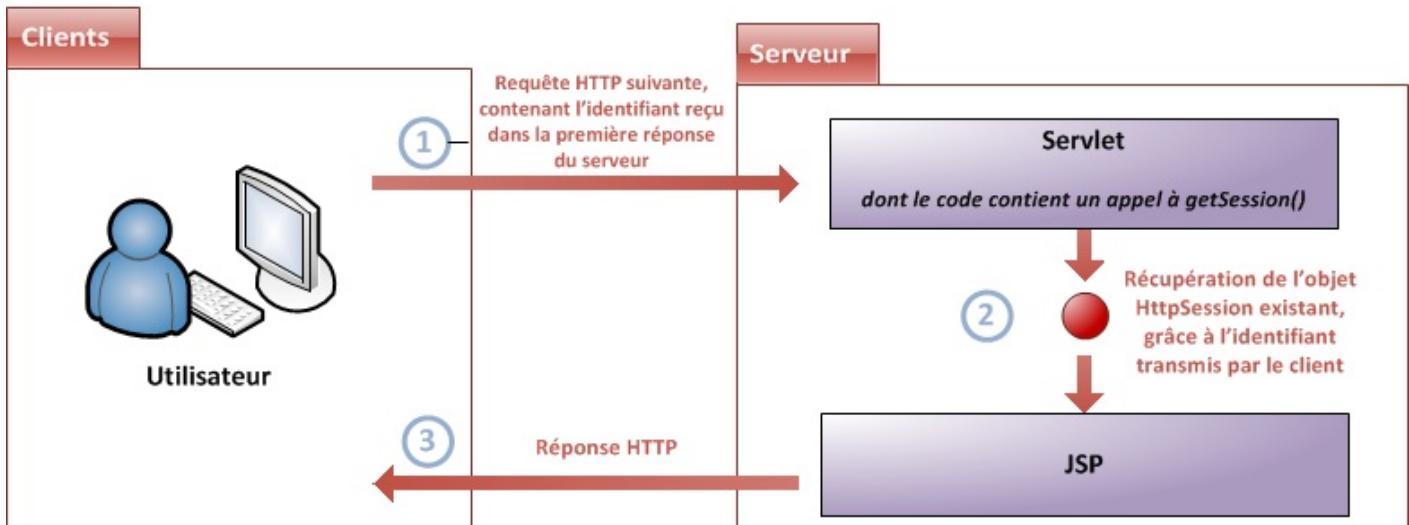
C'est tout ce que nous avons besoin de savoir pour le moment.

Observez sur la figure suivante 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 à la figure suivante ce qui se passe lors des prochaines visites.



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 capables 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 : pour 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 assidus, 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
 */
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 précédemment citées et renvoie un bean **Utilisateur**.

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 un 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 :

- 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

```

...
<servlet>
    <servlet-name>Connexion</servlet-name>
    <servlet-class>com.sdzee.servlets.Connexion</servlet-class>
</servlet>

...
<servlet-mapping>
    <servlet-name>Connexion</servlet-name>
    <url-pattern>/connexion</url-pattern>
</servlet-mapping>

...

```

Une fois tout ce code bien en place et votre serveur redémarré, vous pouvez accéder à la page de connexion via votre navigateur en vous rendant sur l'URL <http://localhost:8080/pro/connexion>. À la figure suivante, le résultat attendu.

Connexion

Vous pouvez vous connecter via ce formulaire.

Adresse email *	<input type="text"/>
Mot de passe *	<input type="password"/>
<input type="button" value="Connexion"/>	

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... Aux figures suivantes, le résultat actuel respectivement lors d'un échec et d'un succès de la validation.

Connexion

Vous pouvez vous connecter via ce formulaire.

Adresse email *	<input type="text" value="test@test"/>	Merci de saisir une adresse mail valide.
Mot de passe *	<input type="password"/>	Le mot de passe doit contenir au moins 3 caractères.
<input type="button" value="Connexion"/>		

Échec de la connexion.

Échec de la connexion

Connexion

Vous pouvez vous connecter via ce formulaire.

Adresse email *	<input type="text" value="test@test.com"/>
Mot de passe *	<input type="password"/>
<input type="button" value="Connexion"/>	

Succès 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

```
<%@ page pageEncoding="UTF-8" %>
<%@ 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']}

```

Dans cette courte modification, aux lignes 31 à 35, 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 33 et 34) 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 à la figure suivante le résultat attendu après succès de la connexion.

Connexion

Vous pouvez vous connecter via ce formulaire.

Adresse email *

Mot de passe *

Succès de la connexion.

Vous êtes connecté(e) avec l'adresse : test@test.com

Ensuite, réaffichez 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, simplement entrez à nouveau l'URL dans le même onglet de votre navigateur, ou bien ouvrez un nouvel onglet. Voici à la figure suivante le résultat attendu.

Connexion

Vous pouvez vous connecter via ce formulaire.

Adresse email *

Mot de passe *

Vous êtes connecté(e) avec l'adresse : test@test.com

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 reconnus : 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 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 regardez à la figure suivante le résultat renvoyé.

Connexion

Vous pouvez vous connecter via ce formulaire.

Adresse email *

Mot de passe *

Le mot de passe doit contenir au moins 3 caractères.

Échec de la connexion.

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 reconnus : 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 au-delà de laquelle la session est 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 est détruite. Je l'ai ici abaissé 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 oubliés : 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 et 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, dans notre servlet j'ai fait en sorte que lorsque vous vous déconnectez, vous êtes redirigés 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 l'application :

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

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

...
<servlet-mapping>
    <servlet-name>Deconnexion</servlet-name>
    <url-pattern>/deconnexion</url-pattern>
</servlet-mapping>

...
```

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.

Difference 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 );
}
}
}

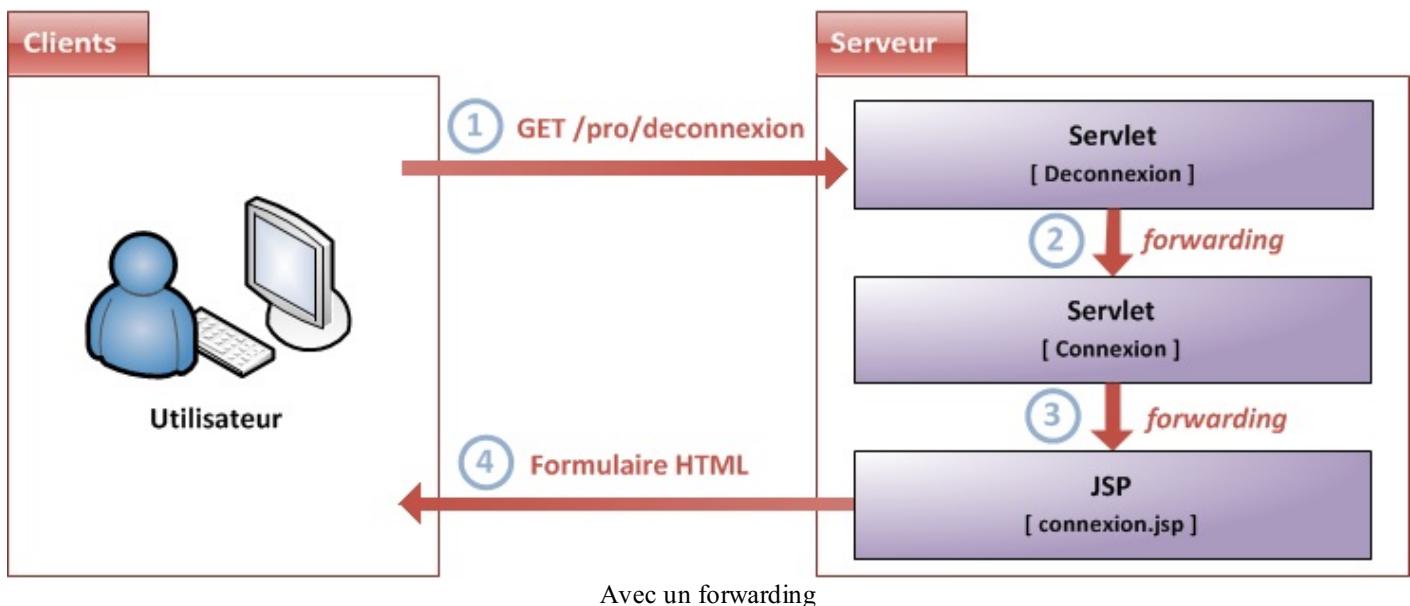
```

Nous avons ici simplement mis en place un *forwarding* vers la servlet de connexion : une fois déconnectés, vous allez visualiser le formulaire de connexion dans votre navigateur. Oui, mais voyez plutôt ce qu'indique la figure suivante !



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'adresses de votre navigateur n'a pas changé ! Pourtant, côté serveur, a été effectué un petit enchaînement de *forwardings*, comme on peut le voir à la figure suivante.

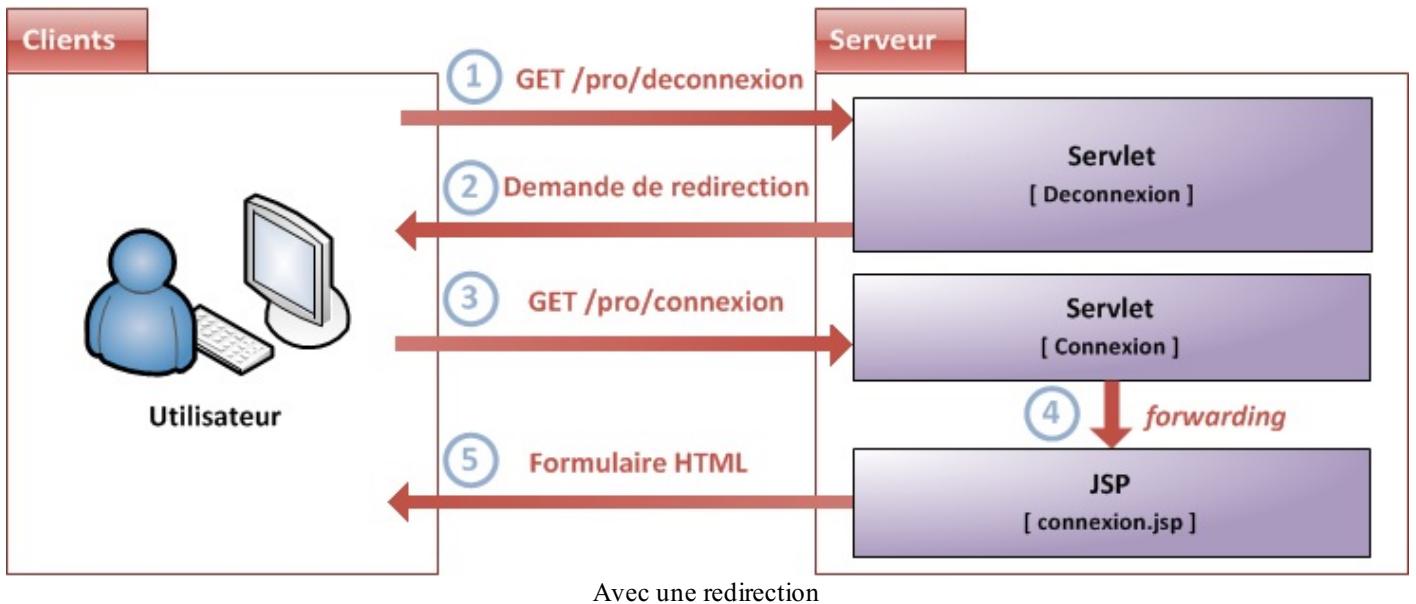


1. l'utilisateur accède à la page de déconnexion depuis son navigateur ;
2. la servlet de déconnexion transfère la requête vers la servlet de connexion via un *forwarding* ;
3. la servlet de connexion transfère la requête vers la JSP du formulaire de connexion via un *forwarding* ;
4. la JSP renvoie 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'adresses : 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 (voir la figure suivante).



- l'utilisateur accède à la page de déconnexion depuis son navigateur ;
- la servlet de déconnexion envoie une demande de redirection au navigateur vers la servlet de connexion, via un `sendRedirect("/pro/connexion")` ;
- le navigateur de l'utilisateur exécute alors la redirection et effectue alors une nouvelle requête vers la servlet de connexion ;
- la servlet de connexion transfère la requête vers la JSP du formulaire de connexion via un *forwarding* ;
- 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'adresses du navigateur est bien celle de la page de connexion, et l'utilisateur n'est pas dérouté.

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 troubistant, 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*, 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*, 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 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 que sont exactement ces cookies, et 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. Donc, chaque fois qu'il crée une session pour un nouveau client, le serveur va envoyer son identifiant au navigateur de celui-ci.



Comment est géré ce cookie ?

Je vous l'ai déjà dit, nous allons y revenir plus en détail 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 `HttpSession` 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'aura alors accès à la précédente session : le conteneur de servlets en créera une nouvelle.

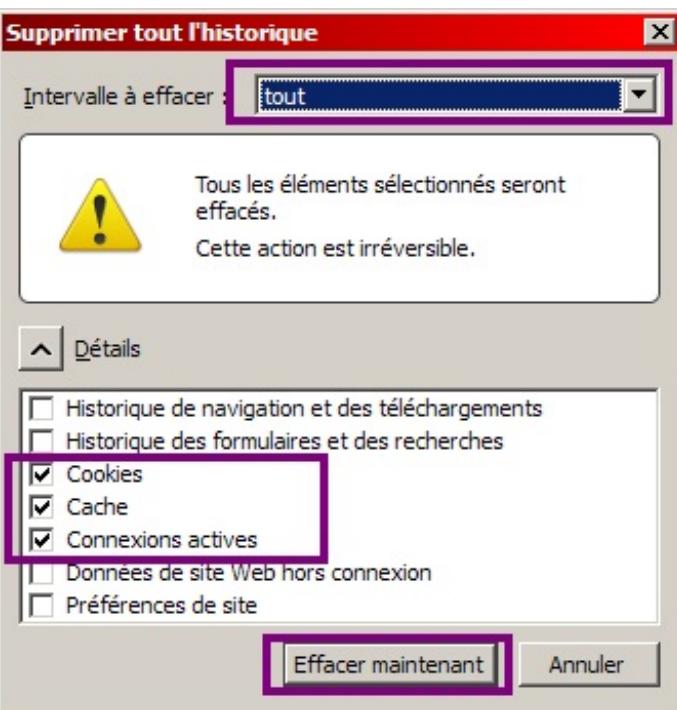
Côté client, le cookie de session a une durée de vie également, qui par défaut est limitée au temps durant lequel le navigateur reste ouvert. Lorsque le client ferme son navigateur, le cookie est donc détruit côté client. Si le client ouvre à nouveau son navigateur, le cookie associé à la précédente 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 les légendes 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...

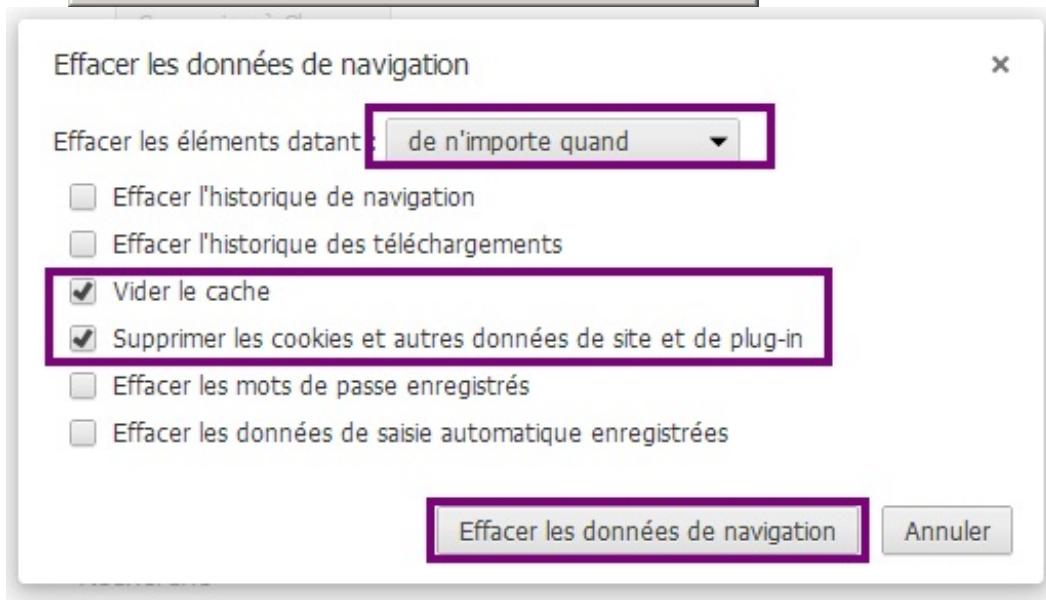
La pratique : scrutons nos requêtes et réponses

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 (voir les figures suivantes).



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.

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 (voir les figures suivantes).

The screenshot shows a Firefox browser window with a login form titled "Connexion". The URL bar contains "localhost:8080/pro/connexion". The form has fields for "Adresse email *" and "Mot de passe *", and a "Connexion" button. Below the browser is the Firebug extension interface. The Network tab is selected, showing a list of requests. The first request, "GET connexion", is expanded, showing its headers. The "Set-Cookie" header is highlighted with a purple box, containing the value "JSESSIONID=9FDE6962675D1CF4EFC32C5C7010CA19; Path=/pro/; HttpOnly". The Request section also lists various client headers.

URL	Statut	Domaine	Poids	Remote IP	Chronologie
GET connexion	200 OK	localhost:8080	1.4 KB	127.0.0.1:8080	

En-têtes **Réponse** **Cache** **HTML**

Réponse voir le code source

Content-Length 1421
Content-Type text/html; charset=ISO-8859-1
Date Wed, 30 May 2012 08:32:22 GMT
Server Apache-Coyote/1.1
Set-Cookie JSESSIONID=9FDE6962675D1CF4EFC32C5C7010CA19; Path=/pro/; HttpOnly

Requête voir le code source

Accept text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Encoding gzip, deflate
Accept-Language fr,fr-fr;q=0.8,en-us;q=0.6,en;q=0.4,zh-cn;q=0.2
Connection keep-alive
Host localhost:8080
User-Agent Mozilla/5.0 (Windows NT 5.1; rv:12.0) Gecko/20100101 Firefox/12.0

+ GET form.css 200 OK localhost:8080 758 B 127.0.0.1:8080 ||
2 requests 2.1 KB 63ms (onload: 122ms)

Cookie dans la réponse avec l'outil Firebug

The screenshot shows a web browser window with a login form titled "Connexion". The URL in the address bar is "localhost:8080/pro/connexion". The form contains fields for "Adresse email *" and "Mot de passe *", and a "Connexion" button. Below the browser is the Chrome DevTools Network tab. The "Headers" section shows a request for "connexion" with a status of 200 OK. The "Response Headers" section shows a Set-Cookie header with the value "JSESSIONID=1B2E4C2B24BA28A8113151859D897018; Path=/pro/; HttpOnly". The "Cookies" section also lists this cookie.

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 la figure suivante.

The screenshot shows a web browser window with a login form titled "Connexion". The URL in the address bar is "localhost:8080/pro/connexion". The form contains fields for "Adresse email *" and "Mot de passe *", and a "Connexion" button. Below the browser is a screenshot of the developer tools Network tab. The "Headers" tab is selected, showing the request headers for a GET request to "http://localhost:8080/pro/connexion". A purple box highlights the "Cookie" header, which contains the value "JSESSIONID=1B2E4C2B24BA28A8113151859D897018". Other headers listed include Accept, Accept-Charset, Accept-Encoding, Accept-Language, Cache-Control, Connection, Host, User-Agent, Content-Length, Content-Type, Date, and Server.

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 la figure suivante.

The screenshot shows a browser window with a login form titled "Connexion". The URL is "localhost:8080/pro/connexion". The form has fields for "Adresse email *" and "Mot de passe *", and a "Connexion" button. Below the browser is a Network traffic analysis tool. The "Headers" tab is selected. In the request headers, the "Cookie" field contains "JSESSIONID=1B2E4C2B24BA28A8113151859D897018". In the response headers, the "Set-Cookie" field contains "JSESSIONID=226821971C90981DD10CEB66FFA09D0E; Path=/pro/; HttpOnly".

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 correspond 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()`. Il nous faut donc créer une page JSP pour l'occasion, que nous allons nommer **accesPublic.jsp** et placer directement à la racine de notre application, sous le répertoire **WebContent** :

Code : JSP - /accesPublic.jsp

```
<%@ page pageEncoding="UTF-8" %>
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Accès public</title>
    </head>
    <body>
        <p>Bienvenue sur la page d'accès public.</p>
    </body>
</html>
```

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 la figure suivante.

Bienvenue sur la page d'accès public.

Network traffic analysis tool screenshot showing the request and response headers for the accessed page:

- Request Headers:**
 - Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
 - Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3
 - Accept-Encoding: gzip,deflate,sdch
 - Accept-Language: fr-FR,fr;q=0.8,en-US;q=0.6,en;q=0.4
 - Connection: keep-alive
 - Host: localhost:8080
 - User-Agent: Mozilla/5.0 (Windows NT 5.1) AppleWebKit/536.5 (KHTML, like Gecko) Chrome/19.0.1084.52 Safari/536.5
- Response Headers:**
 - Content-Length: 274
 - Content-Type: text/html; charset=ISO-8859-1
 - Date: Thu, 31 May 2012 01:32:26 GMT
 - Server: Apache-Coyote/1.1
 - Set-Cookie: JSESSIONID=BB569C5F0777ADCC1853937A5C565E93; Path=/pro/; HttpOnly

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'existe 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 œil 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 toute 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 la figure suivante.

Bienvenue sur la page d'accès public.

Name	Path
accesPublic.jsp	/pro

Request URL: http://localhost:8080/pro/accesPublic.jsp
 Request Method: GET
 Status Code: 200 OK

Request Headers

- Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
- Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3
- Accept-Encoding: gzip,deflate,sdch
- Accept-Language: fr-FR,fr;q=0.8,en-US;q=0.6,en;q=0.4
- Cache-Control: max-age=0
- Connection: keep-alive
- Host: localhost:8080
- User-Agent: Mozilla/5.0 (Windows NT 5.1) AppleWebKit/536.5 (KHTML, like Gecko) Chrome/19.0.1084.52 Safari/536.5

Response Headers

- Content-Length: 276
- Content-Type: text/html;charset=ISO-8859-1
- Date: Thu, 31 May 2012 01:55:44 GMT
- Server: Apache-Coyote/1.1

1 requests | 424B transferred | 1.

All | Documents | Stylesheets | Images | Scripts | XHR | Fonts | WebSockets | Other |

Désactivation de la session dans la JSF

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.

Ce genre d'optimisation n'a absolument aucun impact sur une application de petite envergure, vous pouvez alors 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 disponible : 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.

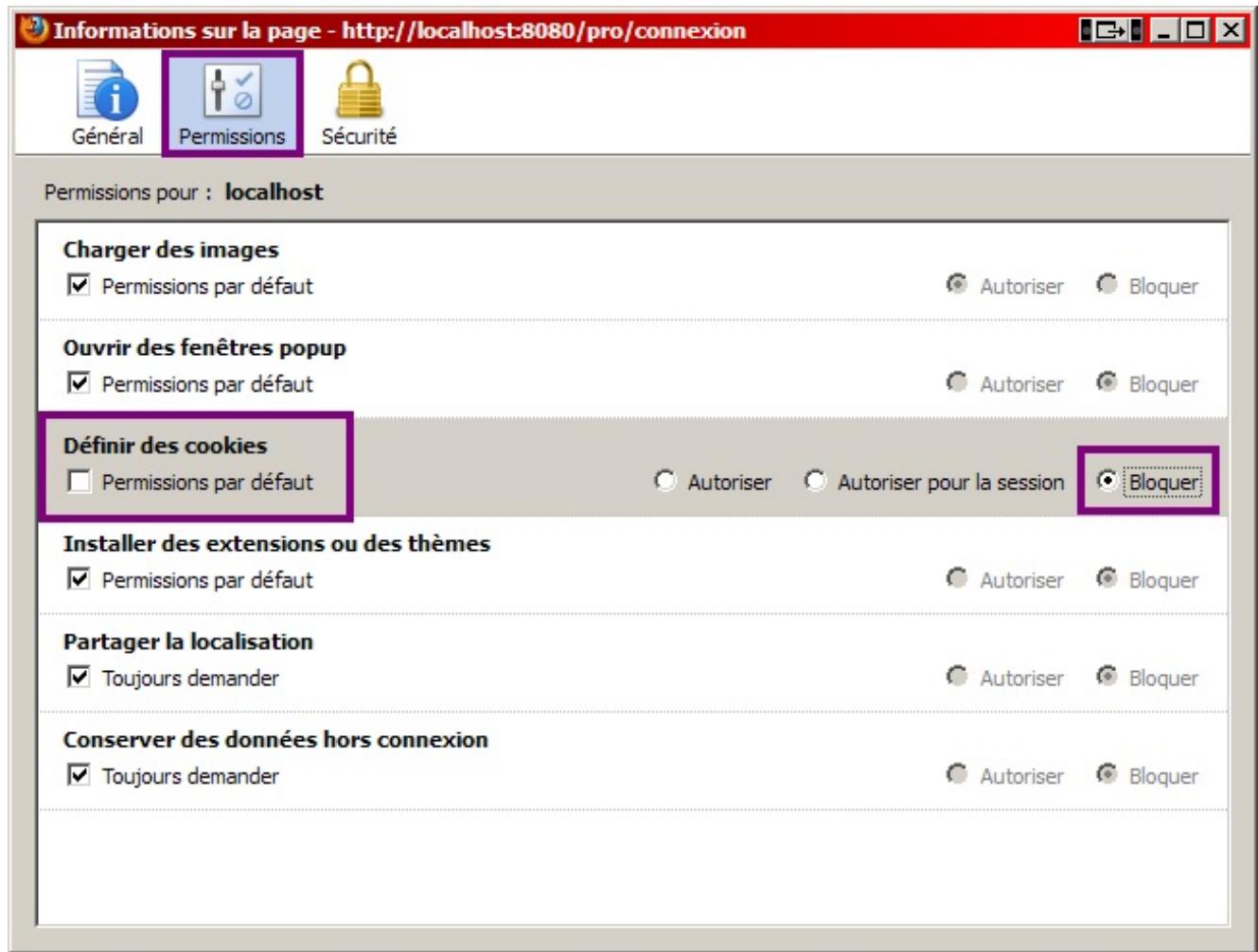
L'accès à une page sans cookie

Dernier scénario 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**.

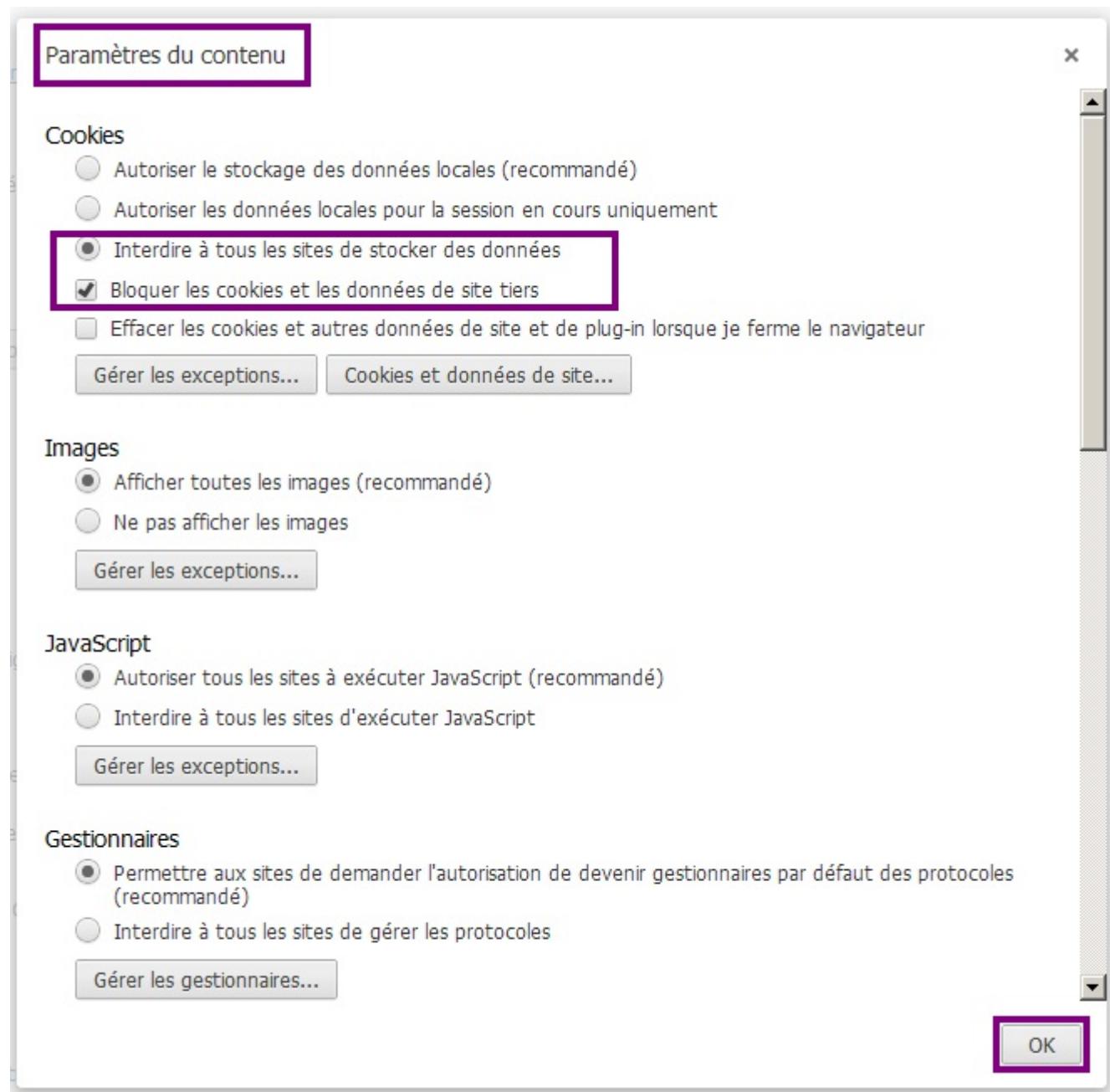
4. Sous **Définir des cookies**, décochez **Permissions par défaut** et cochez **Bloquer**, comme indiqué à la figure suivante.



5. Fermez 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 comme indiqué à la figure suivante.



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 ! 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.



Bien, 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 la figure suivante.

Connexion

Vous pouvez vous connecter via ce formulaire.

Adresse email *	<input type="text" value="test@test.com"/>
Mot de passe *	<input type="password"/>
<input type="button" value="Connexion"/>	

Succès de la connexion.

Vous êtes connecté(e) avec l'adresse : test@test.com

2. Ouvrez alors un nouvel onglet, et rendez-vous à nouveau sur la page de connexion. Observez la figure suivante.

Connexion

Vous pouvez vous connecter via ce formulaire.

Adresse email *	<input type="text"/>
Mot de passe *	<input type="password"/>
<input type="button" value="Connexion"/>	

Lors de nos précédents tests, dans la partie sur les vérifications, le formulaire réaffichait l'adresse mail avec laquelle vous vous étiez connectés auparavant. Cette fois, aucune information n'est réaffichée 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="

```

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 le scénario de tests suivant.

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 la figure suivante.

The screenshot shows a browser window with the title "Connexion" and the URL "localhost:8080/pro/connexion". Below the browser is a developer tools interface with several tabs: Elements, Resources, Network, Scripts, Timeline, and a search bar. The Network tab is active, showing a list of requests. One request is selected, and its details are shown in the main pane. The "Headers" tab is selected, displaying the following information:

- Request URL: http://localhost:8080/pro/connexion
- Request Method: GET
- Status Code: 200 OK
- Request Headers:
 - Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
 - Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3
 - Accept-Encoding: gzip,deflate,sdch
 - Accept-Language: fr-FR,fr;q=0.8,en-US;q=0.6,en;q=0.4
 - Cache-Control: max-age=0
 - Connection: keep-alive
 - Host: localhost:8080
 - User-Agent: Mozilla/5.0 (Windows NT 5.1) AppleWebKit/536.5 (KHTML, like Gecko) Chrome/19.0.1084.52 Safari/536.5
- Response Headers:
 - Content-Length: 1470
 - Content-Type: text/html;charset=ISO-8859-1
 - Date: Thu, 31 May 2012 03:34:18 GMT
 - Server: Apache-Coyote/1.1

A purple box highlights the "Set-Cookie" header entry: "Set-Cookie: JSESSIONID=CD2B836CFA9DCF97ED1E0431F57D00FF; Path=/pro/; HttpOnly".

The "Response" tab shows the HTML code for the page:

```
7   <link type="text/css" rel="stylesheet" href="inc/form.css" />
8   </head>
9   <body>
10  <form method="post"
11    action="/pro/connexion;jsessionid=CD2B836CFA9DCF97ED1E0431F57D00FF">
12    <fieldset>
13      <legend>Connexion</legend>
14      <p>Vous pouvez vous connecter via ce formulaire.</p>
15
16      <label for="nom">Adresse email <span class="requis">*</span></label>
17      <input type="email" id="email" name="email" value="" size="20"
18        maxlength="60" />
19      <span class="erreur"></span>
20      <br />
```

Connectez-vous alors avec des données valides. Vous retrouverez alors, dans la barre d'adresses de votre navigateur, l'URL modifiée par la balise **<c:url>**, contenant l'identifiant de session passé par le serveur (voir la figure suivante).

The screenshot shows a browser window with the URL `localhost:8080/pro/connexion;jsessionid=CD2B836CFA9DCF97ED1E0431F57D00FF`. The page title is "Connexion". The form contains fields for "Adresse email *" (with value `test@test.com`) and "Mot de passe *". A "Connexion" button is present. Below the form, a green message says "Succès de la connexion." and "Vous êtes connecté(e) avec l'adresse : test@test.com".

Ouvrez un nouvel onglet, et copiez/collez l'URL dans la barre d'adresses pour y ouvrir à nouveau la page de connexion en conservant le **JSESSIONID**. Vous constaterez cette fois que le serveur vous a bien reconnus 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 (voir la figure suivante).

The screenshot shows a browser window with the URL `localhost:8080/pro/connexion;jsessionid=CD2B836CFA9DCF97ED1E0431F57D00FF`. The page title is "Connexion". The form contains fields for "Adresse email *" and "Mot de passe *". A "Connexion" button is present. Below the form, a green message says "Vous êtes connecté(e) avec l'adresse : test@test.com".

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 !

En résumé

- La session complète un manque du protocole HTTP, en permettant au serveur de reconnaître et tracer un visiteur ;
- une session est un espace mémoire alloué sur le serveur, et dont le contenu n'est accessible que depuis le serveur ;
- le serveur représente une session par l'objet `HttpSession`, initialisé par un simple appel à `request.getSession()` ;
- 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 - avec `<c:url>` par exemple - pour permettre une gestion continue des sessions, indépendante de l'acceptation ou non des cookies côté client ;
- une session expire et est détruite après le départ d'un visiteur (fermeture de son navigateur), après son inactivité prolongée ou après sa déconnexion manuelle ;
- pour effectuer une déconnexion manuellement, côté serveur il suffit d'appeler la méthode `session.invalidate()` ;

- la désactivation de la gestion des sessions sur une page en particulier est possible via la directive JSP <%@ page session="false" %>.

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 pages, 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 l'exemple basique que je vous propose :

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

```
<%@ page pageEncoding="UTF-8" %>
<!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>
```

Reprenez alors la page **accesPublic.jsp** créée dans le chapitre précédent et modifiez son code ainsi :

Code : JSP - /accesPublic.jsp

```
<%@ page pageEncoding="UTF-8" %>
<!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 je mettais en place un *forwarding* vers la page publique au lieu d'une redirection HTTP, alors l'URL dans le navigateur d'un utilisateur non connecté ne changerait pas lorsqu'il échoue à accéder à la page restreinte. Autrement dit, il serait redirigé vers la page publique de manière transparente, et l'URL de son navigateur lui suggèrerait donc qu'il se trouve sur la page restreinte, ce qui n'est évidemment pas le cas.

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 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 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-mêmes 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 attentifs 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

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

...
<servlet-mapping>
  <servlet-name>Restriction</servlet-name>
  <url-pattern>/restriction</url-pattern>
</servlet-mapping>

...
```

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) ;
3. cliquez alors sur le lien vers la page de connexion, entrez des informations valides et connectez-vous ;

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) ;
5. allez maintenant sur la page <http://localhost:8080/pro/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. La réponse à nos soucis s'appelle le **filtre**, et nous allons le découvrir dans le paragraphe suivant.

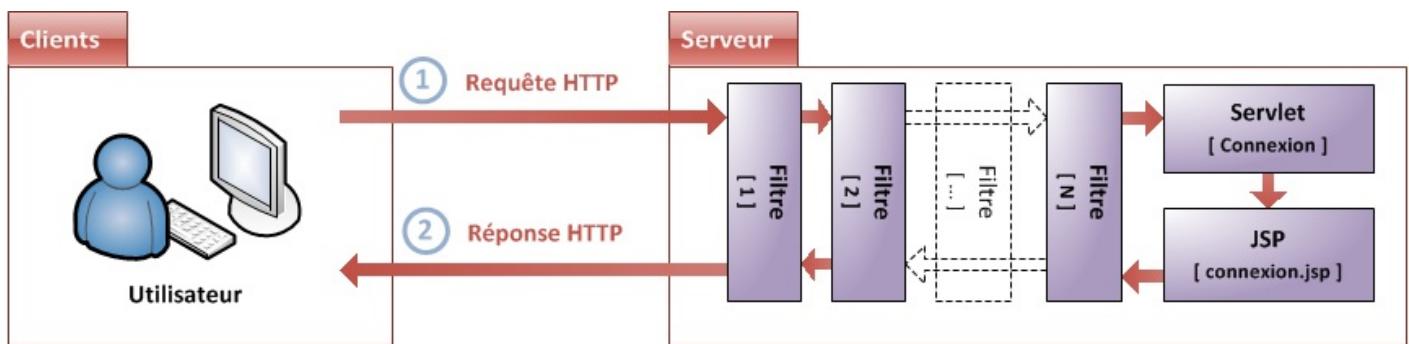
Le principe du filtre

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 à la figure suivante un schéma représentant le cas où plusieurs filtres seraient associés à notre servlet de connexion.



Vous pouvez dès lors 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, en cascade.



Quelle est la différence entre un filtre et une servlet ?

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

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



Quel est l'intérêt d'un filtre ?

Le filtre offre trois avantages majeurs, qui sont interdépendants :

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

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 œil 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 reconnaisserez 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 nommé **restreint** que nous allons placer à la racine de notre projet, dans lequel nous allons déplacer le fichier **accesRestreint.jsp** et y placer les deux fichiers suivants :

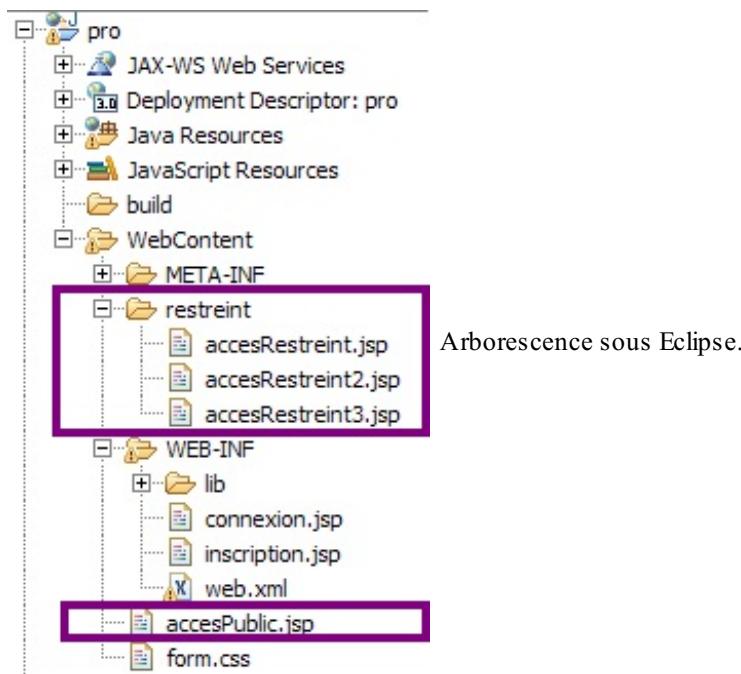
Code : JSP - /restreint/accesRestreint2.jsp

```
<%@ page pageEncoding="UTF-8" %>
<!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

```
<%@ page pageEncoding="UTF-8" %>
<!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 à la figure suivante un aperçu de l'arborescence que vous devez alors obtenir.



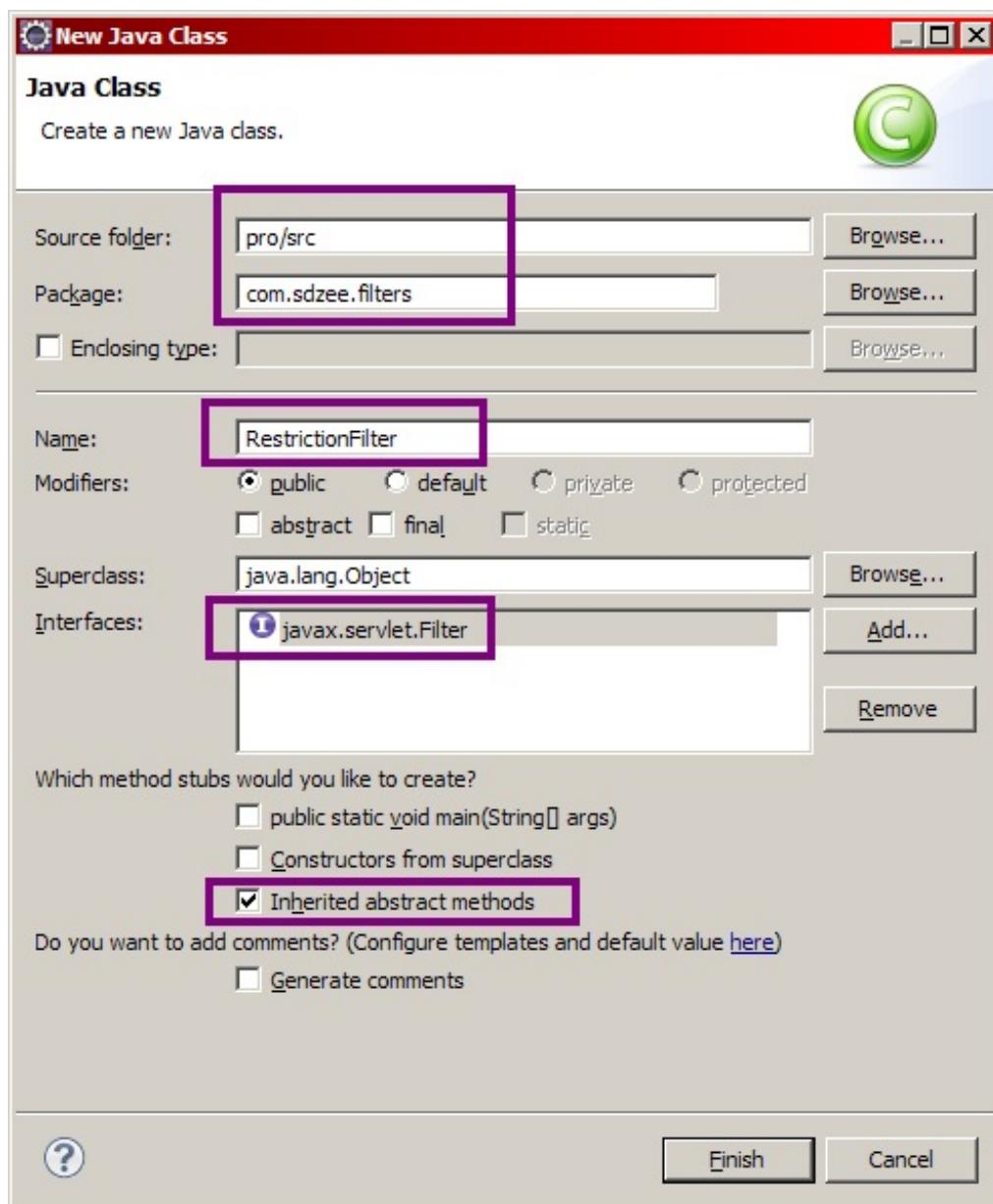
Arborescence sous Eclipse.

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 leurs URL respectives. Par exemple, vous pouvez vous rendre sur <http://localhost:8080/pro/restreint/accesRestreint.jsp> même sans être connectés, 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 à la figure suivante comment procéder après un Ctrl + N sous Eclipse.



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 arguments à notre méthode `doFilter()`. La raison en 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

```
...
<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>
...
...
```

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 ;
2. rendez-vous sur la page de connexion et connectez-vous avec des informations valides ;
3. essayez à nouveau d'accéder à la page <http://localhost:8080/pro/restreint/accesRestreint.jsp>, et constatez le succès de l'opération ;
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 ;
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.

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, c'est-à-dire un aller-retour avec le navigateur du client. Le client va donc 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 aux figures suivantes les messages affichés respectivement par Chrome et Firefox.

The screenshot shows a browser window with the URL `localhost:8080/pro/accesPublic.jsp`. The main content area displays the following message:

Cette page Web présente chrome une boucle de redirection.

La page Web à l'adresse <http://localhost:8080/pro/accesPublic.jsp> a déclenché trop de redirections. Pour résoudre le problème, effacez les cookies de ce site ou autorisez les cookies tiers. Si le problème persiste, il peut être dû à une mauvaise configuration du serveur et n'être aucunement lié à votre ordinateur.

Voici quelques suggestions :

- [Actualisez cette page Web ultérieurement.](#)
- [En savoir plus sur ce problème.](#)

Below the main message, there is a smaller note: "Erreur 310 (net::ERR_TOO_MANY_REDIRECTS) : Trop de redirections".

Échec de la restriction



La page n'est pas redirigée correctement

Firefox a détecté que le serveur redirige la demande pour cette adresse d'une manière qui n'aboutira pas.

- La cause de ce problème peut être la désactivation ou le refus des cookies.

[Réessayer](#)

Échec 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 le code de notre filtre, les changements intervenant aux lignes 16 et 37 :

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 le formulaire affiché à la figure suivante.

localhost:8080/pro/restreint/accesRestreint.jsp

Connexion

Vous pouvez vous connecter via ce formulaire.

Adresse email *

Mot de passe *

Connexion

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 superfiltre ! 😊



Comment régler ce problème ?

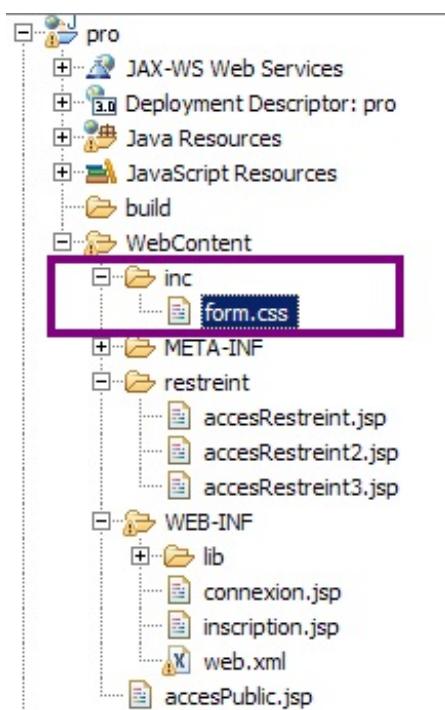
Il y a plusieurs solutions envisageables. Voici les deux plus courantes :

- 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 le fichier CSS, comme

indiqué à la figure suivante.



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 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>. Observez la figure suivante.

The screenshot shows a web browser window with the URL localhost:8080/pro/connexion in the address bar. The page title is "Connexion". The content of the page is:

Vous pouvez vous connecter via ce formulaire.

Adresse email *

Mot de passe *

Le résultat est parfait, ça 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>.

 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 dans votre dernière fenêtre. 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 `/reestreint/accesRestreint.jsp`.

De ce fait, lorsqu'il va silencieusement envoyer une requête au serveur pour récupérer la feuille CSS associée à la page de connexion, 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 `/reestreint` (puisque pour lui, le formulaire a été affiché par `/reestreint/accesRestreint.jsp`). Ainsi, le navigateur va demander au serveur de lui renvoyer la page `/reestreint/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="" />
```

Vous vous souvenez de ce que je vous avais expliqué à propos de la balise `<c:url>` ? Je vous avais dit qu'elle ajoutait automatiquement le contexte de l'application aux URL absolues qu'elle contenait. C'est exactement ce que nous souhaitons : dans ce cas, le rendu de la balise sera `/pro/inc/form.css`. Le navigateur reconnaîtra 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 (voir la figure suivante).

Connexion

Vous pouvez vous connecter via ce formulaire.

Adresse email *

Mot de passe *

Connexion

Finalement, nous y sommes : tout fonctionne comme prévu !



Si je vous fais mettre en place une telle restriction, c'est uniquement pour vous faire découvrir le principe. Dans une vraie application, il faudra bien prendre garde à ne pas restreindre des pages dont l'accès est supposé être libre !

Par exemple, dans votre projet il est dorénavant impossible pour un utilisateur de s'inscrire ! Eh oui, réfléchissez-bien : puisque le filtre est en place, dès lors qu'un utilisateur non inscrit (et donc non connecté) tente d'accéder à la page d'inscription, il est automatiquement redirigé vers la page de connexion ! Devoir se connecter avant même de pouvoir s'inscrire, admettez qu'on a connu plus logique... Pour régler le problème, il suffirait en l'occurrence d'ajouter une exception au filtre pour autoriser l'accès à la page d'inscription, tout comme nous l'avons fait pour le dossier /inc. Mais ce n'est pas sur cette correction en particulier que je souhaite insister, vous devez surtout bien réaliser que **lorsque vous appliquez des filtres avec un spectre très large, voire intégral, alors vous devez faire très attention et bien réfléchir à tous les cas d'utilisation de votre application.**

Désactiver le filtre

Une fois vos développements et tests terminés, pour plus de commodité dans les exemples à suivre, je vous conseille de désactiver ce filtre. Pour cela, commentez 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.

Modifier le mode de déclenchement d'un filtre

Je vous ai implicitement fait comprendre à travers ces quelques exemples qu'un filtre était déclenché lors de la réception d'une requête HTTP uniquement. Eh bien sachez qu'il s'agit là d'un comportement par défaut ! En réalité, un filtre est tout à fait capable de s'appliquer à un *forwarding*, mais il faut pour cela modifier sa déclaration dans le fichier **web.xml** :

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>
    <dispatcher>REQUEST</dispatcher>
    <dispatcher>FORWARD</dispatcher>
</filter-mapping>

```

Il suffit, comme vous pouvez l'observer, de rajouter un champ **<dispatcher>** à la fin de la section **<filter-mapping>**.

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

Code : XML

```
<dispatcher>INCLUDE</dispatcher>
```

Nous n'allons pas nous amuser à vérifier le bon fonctionnement de ces changements. Retenez simplement qu'il est bel et bien possible de filtrer les *forwardings* et inclusions en plus des requêtes directes entrantes, en modifiant au cas par cas les déclarations des filtres à appliquer. Enfin, n'oubliez pas que ces ajouts au fichier **web.xml** ne sont pris en compte qu'après un redémarrage du serveur.

Retour sur l'encodage UTF-8

Avant de passer à la suite, je souhaite vous faire découvrir un autre exemple d'utilisation d'un filtre. Vous ne l'avez peut-être pas encore remarqué, mais notre application ne sait toujours pas correctement gérer les caractères accentués, ni les caractères spéciaux et alphabets non latins...



Comment est-ce possible ? Nous avons déjà paramétré Eclipse et notre projet pour que tous nos fichiers soient encodés en UTF-8, il ne devrait plus y avoir de problème !

Je vous ai déjà expliqué, lorsque nous avons associé notre première servlet à une page JSP, que les problématiques d'encodage interviennent à deux niveaux : côté navigateur et côté serveur. Eh bien en réalité, ce n'est pas si simple. Avec ce que nous avons mis en place, le navigateur est bien capable de déterminer l'encodage des données envoyées par le serveur, mais le serveur quant à lui est incapable de déterminer l'encodage des données envoyées par le client, lors d'une requête GET ou POST. Essayez dans votre formulaire d'inscription d'entrer un nom qui contient un accent, par exemple (voir la figure suivante).

Inscription

Vous pouvez vous inscrire via ce formulaire.

Adresse email *	<input type="text"/>
Mot de passe *	<input type="password"/>
Confirmation du mot de passe *	<input type="password"/>
Nom d'utilisateur	<input type="text" value="cèpe"/>
<input type="button" value="Inscription"/>	

Si vous cliquez alors sur le bouton d'inscription, votre navigateur va envoyer une requête POST au serveur, qui va alors retourner le résultat affiché à la figure suivante.

Inscription

Vous pouvez vous inscrire via ce formulaire.

Adresse email *	<input type="text"/>	Merci de saisir une adresse mail.
Mot de passe *	<input type="password"/>	Merci de saisir et confirmer votre mot de passe.
Confirmation du mot de passe *	<input type="password"/>	
Nom d'utilisateur	cÃ“pe	
<input type="button" value="Inscription"/>		

Échec de l'inscription.

Vous devez ici reconnaître le problème que nous avions rencontrés à nos débuts ! Là encore, il s'agit d'une erreur d'interprétation : le serveur considère par défaut que les données qui lui sont transmises suivent l'encodage latin ISO-8859-1, alors qu'en réalité ce sont des données en UTF-8 qui sont envoyées, d'où les symboles bizarroïdes à nouveau observés...



Très bien, mais quel est le rapport avec nos filtres ?

Pour corriger ce comportement, il est nécessaire d'effectuer un appel à la méthode `setCharacterEncoding()` non plus depuis l'objet `HttpServletResponse` comme nous l'avons fait dans notre toute première servlet, mais depuis l'objet `HttpServletRequest` ! En effet, nous cherchons bien ici à préciser l'encodage des données de nos requêtes ; nous l'avons déjà appris, l'encodage des données de nos réponses est quant à lui assuré par la ligne placée en tête de chacune de nos pages JSP.

Ainsi, nous pourrions manuellement ajouter une ligne `request.setCharacterEncoding("UTF-8")` ; dans les méthodes `doPost()` de chacune de nos servlets, mais nous savons que dupliquer cet appel dans toutes nos servlets n'est pas pratique du tout. En outre, la documentation de la méthode précise qu'il faut absolument réaliser l'appel avant toute lecture de données, afin que l'encodage soit bien pris en compte par le serveur.

Voilà donc deux raisons parfaites pour mettre en place... un filtre ! C'est l'endroit idéal pour effectuer simplement cet appel à chaque requête reçue, et sur l'intégralité de l'application. Et puisque qu'une bonne nouvelle n'arrive jamais seule, il se trouve que nous n'avons même pas besoin de créer nous-mêmes ce filtre, Tomcat en propose déjà un nativement ! Il va donc nous suffire d'ajouter une déclaration dans le fichier `web.xml` de notre application pour que notre projet soit enfin capable de gérer correctement les requêtes POST qu'il traite :

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

```

...
<filter>
    <filter-name>Set Character Encoding</filter-name>
    <filter-
class>org.apache.catalina.filters.SetCharacterEncodingFilter</filter-
class>
    <init-param>
        <param-name>encoding</param-name>
        <param-value>UTF-8</param-value>
    </init-param>
    <init-param>
        <param-name>ignore</param-name>
        <param-value>false</param-value>
    </init-param>
</filter>
<filter-mapping>
    <filter-name>Set Character Encoding</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
...

```

Vous retrouvez la déclaration que vous avez apprise un peu plus tôt, via la balise `<filter>`. La seule nouveauté, c'est l'utilisation du filtre natif `SetCharacterEncodingFilter` issu du package `org.apache.catalina.filters`. Comme vous pouvez le constater, celui-ci nécessite deux paramètres d'initialisation dont un nommé `encoding`, qui permet au développeur de spécifier l'encodage à utiliser. Je vous laisse parcourir la documentation du filtre pour plus d'informations. 😊

De même, vous retrouvez dans la section `<filter-mapping>` l'application du filtre au projet entier, grâce au caractère * appliquée à la racine.

Une fois les modifications effectuées, il vous suffit alors de redémarrer Tomcat et d'essayer à nouveau de saisir un nom accentué. Vous remarquez cette fois la bonne gestion du mot accentué, qui est ré-affiché correctement. Bien évidemment, cela vaut également pour tout caractère issu d'un alphabet non latin : votre application est désormais capable de traiter des données écrites en arabe, en chinois, en russe, etc.

- Le forwarding s'effectue sur le serveur de manière transparente pour le client, alors que la redirection implique un aller/retour chez le client.
- Un filtre ressemble à une servlet en de nombreux aspects :
 - il agit sur le couple requête / réponse initié par le conteneur de servlets ;
 - il se déclare dans le fichier web.xml via deux sections `<filter>` et `<filter-mapping>` ;
 - il contient une méthode de traitement, nommée `doFilter()` ;
 - il peut s'appliquer à un pattern d'URL comme à une page en particulier.
- Plusieurs filtres sont applicables en cascade sur une même paire requête / réponse, dans l'ordre défini par leur déclaration dans le web.xml.
- La transition d'un filtre vers le maillon suivant de la chaîne s'effectue via un appel à la méthode `doFilter()` de l'objet `FilterChain`.

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'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 du serveur vers le 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 qui 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 : 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.

Voici pour commencer la reprise de la méthode `doGet()`, accompagnée des nouvelles constantes et méthodes nécessaires à son bon fonctionnement. Je n'ai volontairement pas inclus le code existant de la méthode `doPost()`, afin de ne pas compliquer la lecture. Lorsque vous reporterez ces modifications sur votre servlet de connexion, ne faites surtout pas un bête copier-coller du code suivant ! Prenez garde à modifier correctement le code existant, et à ne pas supprimer la méthode `doPost()` de votre servlet (que j'ai ici remplacée par "...") :

Code : Java - com.sdzee.servlets.Connexion

```
package com.sdzee.servlets;

import java.io.IOException;
import javax.servlet.*;
import org.joda.time.*;

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,
HttpServletRequest response ) throws ServletException, IOException
{
    /* Tentative de récupération du cookie depuis la requête */
    String derniereConnexion = getCookieValue( request,
COOKIE_DERNIERE_CONNEXION );
    /* Si le cookie existe, alors calcul de la durée */
    if ( derniereConnexion != 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 dtDerniereConnexion = formatter.parseDateTime(
derniereConnexion );
        /* Calcul de la durée de l'intervalle */
        Period periode = new Period( dtDerniereConnexion,
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 );
}

...
/***
* 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 **derniereConnexion** ;
 - j'ai mis en place une méthode nommée `getCookieValue()`, dédiée à la recherche d'un cookie donné dans une requête HTTP :
 - à la ligne 56, 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 57, elle vérifie si des cookies existent, c'est-à-dire si `request.getCookies()` n'a pas retourné **null** ;
 - à la ligne 58, elle parcourt le tableau de cookies récupéré ;
 - à la ligne 59, elle vérifie si un des éventuels cookies présents dans le tableau a le même nom que le paramètre **nom** passé en argument, récupéré par un appel à `cookie.getName()` ;
 - à la ligne 60, si un tel cookie est trouvé, elle retourne sa valeur via un appel à `cookie.getValue()`.
 - à la ligne 23, je teste si ma méthode `getCookieValue()` a retourné une valeur ou non ;
 - de la ligne 24 à la ligne 43, je traite les dates grâce aux méthodes de la bibliothèque JodaTime. Je vous recommande fortement d'aller vous-mêmes 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 **derniereConnexion** placé dans le navigateur le client ;
 - les lignes 27 et 28 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 30, 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 32 à 40, je crée un format d'affichage de mon choix à l'aide de l'objet `PeriodFormatterBuilder` ;
 - à la ligne 41 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 43, je transmets 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 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 **derniereConnexion** 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 **derniereConnexion** qui, éventuellement, existe déjà dans le navigateur du client.

Code : Java - com.sdzee.servlets.Connexion

```

public static final String CHAMP_MEMOIRE = "memoire";
public static final int COOKIE_MAX_AGE = 60 * 60 * 24
* 365; // 1 an

...

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 25 permet de déterminer si la case du formulaire, que j'ai choisi de nommer **memoire**, est cochée ;

- les lignes 29 et 30 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 50 ;
 - 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 35, 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 **derniereConnexion** 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 $60 \times 60 \times 24 \times 365 = 31\,536\,000$ 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 c'est **null**, la case n'est pas cochée ; sinon, 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.

Voici le code, modifié aux lignes 14 à 16 et 29 à 30 :

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="" />
  </head>
  <body>
    <form method="post" action="

```

```

        <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. Rappelez-vous de nos méthodes **doGet()** et **doPost()** : si le client n'a pas envoyé le cookie **derniereConnexion**, 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 s'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 à la figure suivante la réponse renvoyée par le serveur.

The screenshot shows a browser window titled "Connexion" at the URL "localhost:8080/pro/connexion;jsessionid=7D0DE489A8CBCD6B9A2286E26B888BA8". The form contains fields for "Adresse email *" (test@test.com), "Mot de passe *", and a checked "Se souvenir de moi" checkbox. A "Connexion" button is present. Below the form, a success message says "Succès de la connexion." and "Vous êtes connecté(e) avec l'adresse : test@test.com".

Below the browser is a screenshot of the Network tab in developer tools. The Headers section shows standard HTTP headers like Content-Length, Content-Type, and various cookies. The Response Headers section is highlighted with a purple box and shows the Set-Cookie header: "Set-Cookie: derniereConnexion="04/06/2012 15:51:35"; Version=1; Max-Age=31536000; Expires=Tue, 04-Jun-2013 07:51:35 GMT".

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 **derniereConnexion** qui :

- contient bien la date de connexion, formatée selon la convention choisie dans notre servlet ;
- expire bien dans 31 536 000 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 (voir

la figure suivante).

Connexion

Vous pouvez vous connecter via ce formulaire.

Adresse email *

Mot de passe *

Se souvenir de moi

Connexion

Vous êtes connecté(e) avec l'adresse : test@test.com

Déconnectez-vous alors en vous rendant sur <http://localhost:8080/pro/deconnexion>, puis rendez-vous à nouveau sur la page de connexion et observez la figure suivante.

The screenshot shows a browser window with the title "Connexion" and the URL "localhost:8080/pro/connexion". The page content is identical to the one above, but includes a yellow message box stating: "(Vous ne vous êtes pas connecté(e) depuis ce navigateur depuis 15 minutes et 55 secondes)". Below the form, the browser's developer tools Network tab is open, showing a request for "connexion /pro". The "Headers" tab is selected, displaying the following request headers:

```

Request URL: http://localhost:8080/pro/connexion
Request Method: GET
Status Code: 200 OK (from cache)
Request Headers
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3
Accept-Encoding: gzip,deflate,sdch
Accept-Language: fr-FR,fr;q=0.8,en-US;q=0.6,en;q=0.4
Connection: keep-alive
Cookie: JSESSIONID=DCED3DB3BEF101AF5E21D4844DDC8FC3; dernièreConnexion="04/06/2012 15:51:35"
Host: localhost:8080
User-Agent: Mozilla/5.0 (Windows NT 5.1) AppleWebKit/536.5 (KHTML, like Gecko) Chrome/19.0.1084.52 Safari/536.5

```

The "Cookies" tab shows the cookie "Cookie: JSESSIONID=DCED3DB3BEF101AF5E21D4844DDC8FC3; dernièreConnexion="04/06/2012 15:51:35"

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. 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 (voir la figure suivante).

The screenshot shows a browser window with a login form and a developer tools Network tab.

Browser Window:

- Address bar: localhost:8080/pro/connexion
- Form fields:
 - Adresse email *: test@test.com
 - Mot de passe *: (empty)
 - Se souvenir de moi:
 - Connexion button
- Success message: Succès de la connexion.
- Text: Vous êtes connecté(e) avec l'adresse : test@test.com

Developer Tools Network Tab:

- Request Headers (highlighted):
 - Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
 - Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3
 - Accept-Encoding: gzip,deflate,sdch
 - Accept-Language: fr-FR,fr;q=0.8,en-US;q=0.6,en;q=0.4
 - Cache-Control: max-age=0
 - Connection: keep-alive
 - Content-Length: 37
 - Content-Type: application/x-www-form-urlencoded
 - Cookie: JSESSIONID=4E4AD98523EC7EA3E62AA6AA4B506DF4; dernièreConnexion=11/06/2012 11:54:37
 - Host: localhost:8080
 - Origin: http://localhost:8080
 - Referer: http://localhost:8080/pro/connexion
 - User-Agent: Mozilla/5.0 (Windows NT 5.1) AppleWebKit/536.5 (KHTML, like Gecko) Chrome/19.0.1084.52 Safari/536.5
- Form Data (highlighted):
 - email: test@test.com
 - motdepasse: test
- Response Headers (highlighted):
 - Content-Length: 1597
 - Content-Type: text/html;charset=ISO-8859-1
 - Date: Mon, 11 Jun 2012 03:54:51 GMT
 - Server: Apache-Coyote/1.1
 - Set-Cookie: JSESSIONID=C9A886CE72BE8300C3FA3AE89E904FC9; Path=/pro/; HttpOnly, dernièreConnexion=""; Expires=Thu, 01-Jan-1970 00:00:10 GMT

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... 1^{er} 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 1^{er} 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, car 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 ! 😊

- Un cookie est un fichier texte de très petite taille stocké dans le navigateur du client, alors identifié par un nom et une valeur.
- L'objet `Cookie` permet la gestion des cookies en Java.
- Pour récupérer les cookies d'un client, il suffit d'appeler la méthode `request.getCookies()`, qui renvoie un tableau de cookies.
- Pour analyser un cookie en particulier, il suffit de vérifier son nom via `cookie.getName()` et sa valeur via `cookie.getValue()`.
- Pour initialiser un cookie sur le serveur, il suffit d'appeler `new Cookie(nom, valeur)`.
- Pour l'envoyer au client, il suffit d'appeler la méthode `response.addCookie(cookie)`.
- Un cookie ne peut pas être supprimé manuellement depuis le serveur, mais il contient une date d'expiration afin que le navigateur puisse déterminer quand le détruire.

TP Fil rouge - Étape 4

Les sessions constituant déjà un gros morceau à elles seules, dans cette étape du fil rouge vous n'allez pas manipuler de cookies. Et croyez-moi, même sans ça vous avez du pain sur la planche !

Objectifs

Fonctionnalités

Vous allez devoir effectuer une modification importante, qui va impacter presque tout votre code existant : je vous demande dans cette étape d'enregistrer en session les clients et commandes créés par un utilisateur. Cela pourrait très bien être un jeu d'enfants, si je ne vous demandais rien d'autre derrière... Mais ne rêvez pas, vous n'êtes pas ici pour vous tourner les pouces ! 😊

Enregistrement en session

Pour commencer, comme je viens de vous l'annoncer, vous allez devoir enregistrer en session les clients et commandes créés par un utilisateur. Ainsi, les informations saisies par l'utilisateur ne seront plus perdues après validation d'un formulaire de création !

Liste récapitulative des clients et commandes créés

Deuxièmement, vous allez devoir créer deux nouvelles pages :

- **listerClients.jsp**, qui listera les clients créés par l'utilisateur ;
- **listerCommandes.jsp**, qui listera les commandes créées par l'utilisateur.

Vous les placerez bien entendu sous **/WEB-INF** tout comme leurs consœurs. Vous en profiterez pour mettre à jour la page **menu.jsp** en y ajoutant deux liens vers les nouvelles servlets gérant ces deux pages fraîchement créées.

Un formulaire de création de commande intelligent

Troisièmement, et cette fois il va falloir réfléchir davantage, je vous demande de modifier le formulaire de création de commandes. Maintenant que votre application est capable d'enregistrer les clients créés par l'utilisateur, vous allez lui donner un choix lorsqu'il crée une commande :

- si la commande qu'il veut créer concerne un nouveau client, alors affichez-lui les champs permettant la saisie des informations du nouveau client, comme vous le faisiez déjà auparavant ;
- si la commande qu'il veut créer concerne un client déjà existant, alors affichez-lui une liste déroulante des clients existants, lui permettant de choisir son client et lui évitant ainsi de saisir à nouveau ces informations.

Système de suppression des clients et commandes

Quatrièmement, vous allez devoir mettre en place un système permettant la suppression d'un client ou d'une commande enregistrée dans la session. Vous allez ensuite devoir compléter vos pages **listerClients.jsp** et **listerCommandes.jsp** pour qu'elles affichent à côté de chaque client et commande un lien vers ce système, permettant la suppression de l'entité correspondante.

Gestion de l'encodage UTF-8 des données

Cinquièmement, vous allez appliquer le filtre d'encodage de Tomcat à votre projet, afin de rendre votre application capable de gérer n'importe quelle donnée UTF-8.

Enfin, vous devrez vous préparer une tasse un thermos de café ou de thé bien fort, parce qu'avec tout ce travail, vous n'êtes pas couchés ! 😊

Exemples de rendus

Voici aux figures suivantes quelques exemples de rendu.

Liste des clients créés au cours de la session, ici avec quatre clients.

The screenshot shows a web browser window with the URL `localhost:8080/tp4/listeClients`. The page contains a sidebar with links: [Créer un nouveau client](#), [Créer une nouvelle commande](#), [Voir les clients existants](#), and [Voir les commandes existantes](#). Below the sidebar is a table listing four clients:

Nom	Prénom	Adresse	Téléphone	Email	Action
Martin	Pierre	6 avenue du Parc, Lyon	0488776655	pmartin@mail.com	X
Dupuis	Germain	3 rue du Tholon, Bourges	0311223344	gdupuis@mail.fr	X
Durand	Pascal	12 rue de la marmotte, Avoriaz	0412345678		X
Coyote		Pékin, Chine	123456789		X

Liste des commandes lorsqu'aucune commande n'a été créée au cours de la session :

The screenshot shows a web browser window with the URL `localhost:8080/tp4/listeCommandes`. The page contains a sidebar with links: [Créer un nouveau client](#), [Créer une nouvelle commande](#), [Voir les clients existants](#), and [Voir les commandes existantes](#). Below the sidebar, a message in red text reads: **Aucune commande enregistrée.**

Formulaire de création d'une commande lorsque la case "Nouveau client : Oui" est cochée :

localhost:8080/tp4/creationCommande

[Créer un nouveau client](#)

[Créer une nouvelle commande](#)

[Voir les clients existants](#)

[Voir les commandes existantes](#)

Informations client

Nouveau client ? * Oui Non

Nom * Durand

Prénom Pascal

Adresse de livraison * 1 rue du pois, Chiche

Numéro de téléphone * 0877665544

Adresse email pascal.d@yahee.com

Informations commande

Date *

Montant * 1234.56

Mode de paiement * CB

Statut du paiement

Mode de livraison * UPS

Statut de la livraison

Valider **Remettre à zéro**

Formulaire de création d'une commande lorsque la case "Nouveau client : Non" est cochée :

[Créer un nouveau client](#)

[Créer une nouvelle commande](#)

[Voir les clients existants](#)

[Voir les commandes existantes](#)

Informations client

Nouveau client ? *

Oui Non

Choisissez un client...

Choisissez un client...

- Pierre Martin
- Germain Dupuis
- Pascal Durand
- Coyote

Mode de paiement *

Statut du paiement

Mode de livraison *

Statut de la livraison

Valider **Remettre à zéro**

Liste des commandes lorsque deux commandes ont été créées au cours de la session :

[Créer un nouveau client](#)

[Créer une nouvelle commande](#)

[Voir les clients existants](#)

[Voir les commandes existantes](#)

Client	Date	Montant	Mode de paiement	Statut de paiement	Mode de livraison	Statut de livraison	Action
Pascal Durand	26/06/2012 10:08:50	1234.56	CB		UPS		
Coyote	26/06/2012 10:08:11	789456.0	Chèque		ACME Delivery		

Conseils

Enregistrement en session

Concernant l'aspect technique de cette problématique, vous avez toutes les informations dans le chapitre sur ce sujet : il vous suffit de récupérer la session en cours depuis l'objet requête, et d'y mettre en place des attributs. Concernant la mise en place dans ce cas précis par contre, il y a une question que vous allez devoir vous poser : quel type d'attributs enregistrer en session ? Autrement dit, comment stocker les clients et commandes ? C'est une bonne question. Essayez d'y réfléchir par vous-mêmes avant de lire le conseil qui suit...

Ici, ce qui vous intéresse, c'est d'enregistrer les clients et les commandes créés. Une liste de clients et une liste de commandes

pourraient donc a priori faire l'affaire, mais retrouver quelque chose dans une liste, ce n'est pas simple... Pourtant, vous aimerez bien pouvoir identifier facilement un client ou une commande dans ces listes. Pour cette raison, une Map semble la solution adaptée. Oui, mais comment organiser cette Map ? Les objets **Client** et **Commande** seront bien entendu les valeurs, mais qu'est-ce qui va bien pouvoir servir de clé ? Il y a tout un tas de solutions possibles, mais je vous conseille pour le moment de mettre en place la moins contraignante : considérez que le **nom** permet d'identifier de manière unique un client, et que la **date** permet d'identifier de manière unique une commande. Cela implique que votre application ne permettra pas de créer deux commandes au même instant, ni de créer deux clients portant le même nom. Ce n'est pas génial comme comportement, mais pour le moment votre application ne gère toujours pas les données, donc ça ne vous pose absolument aucun problème ! Vous aurez tout le loisir de régler ce petit souci dans l'étape 6 du fil rouge ! 😊



Bref, je vous conseille donc de placer en session une `Map<String, Client>` contenant les clients identifiés par leur **nom**, et une `Map<String, Commande>` contenant les commandes identifiées par leur **date**.

Liste récapitulative des clients et commandes créés

Deux pages JSP accédant directement aux attributs stockés en session suffisent ici. Puisque vous allez les placer sous `/WEB-INF`, elles ne seront pas accessibles directement par leur URL et vous devrez mettre en place des servlets en amont, qui se chargeront de leur retransmettre les requêtes reçues. Vous pouvez par exemple les appeler **ListeClients** et **ListeCommandes**, et vous n'oublierez pas de les déclarer dans votre fichier `web.xml`.

Dans chacune de ces JSP, le plus simple est de générer un tableau HTML qui affichera ligne par ligne les clients ou commandes créés (voir les exemples de rendus). Pour cela, il vous suffit de parcourir les Map enregistrées en tant qu'attributs de session à l'aide de la balise de boucle JSTL `<c:forEach>`. Relisez le passage du cours sur la JSTL si vous avez oublié comment itérer sur une collection, et relisez la correction de l'exercice sur la JSTL Core si vous avez oublié comment atteindre les clés et valeurs d'une Map depuis une EL ! Dans le corps de cette boucle, vous placerez les balises HTML nécessaires à la création des lignes du tableau HTML, contenant les entrées de la Map des clients ou des commandes.

En bonus, pour aérer la lecture du tableau final dans le navigateur, vous pouvez utiliser un compteur au sein de votre boucle pour alterner la couleur de fond d'une ligne à l'autre. Cela se fait par exemple en testant simplement si l'indice de parcours de la boucle est pair ou impair.

Second bonus, vous pouvez mettre en place un test vérifiant si les objets présents en sessions existent ou non, via par exemple la balise `<c:choose>` : si non, alors vous pouvez par exemple afficher un message signalant qu'aucun client ou aucune commande n'existe (voir les exemples de rendus).

Un formulaire de création de commande intelligent

En ce qui concerne cette modification, vous allez devoir réfléchir un peu à la solution à mettre en place : comment donnez un choix à l'utilisateur ? Si vous regardez les exemples de rendus que je vous ai donnés ci-dessus, vous verrez que j'ai mis en place deux simples boutons de type `<input type="radio" />` :

- au clic sur "Oui", la première partie du formulaire classique s'affiche ;
- au clic sur "Non", un nouveau champ `<select>` listant les clients existants remplace la première partie du formulaire !

Bien entendu, vous n'êtes pas forcés d'utiliser ce moyen ! Si vous souhaitez que votre apprentissage soit efficace, alors vous devez réfléchir par vous-mêmes à cette petite problématique et lui trouver une solution adaptée sans mon aide. Ce petit exercice de conception donnera du fil à retordre à vos neurones et vous fera prendre encore un peu plus d'aisance avec le Java EE ! 😊

En outre, la solution que je propose ici est un peu évoluée, car elle implique un petit morceau de code Javascript pour permettre la modification du formulaire au clic sur un bouton radio. Si vous ne la sentez pas, prenez le temps et pensez à un autre système plus simple et peut-être moins évolué graphiquement qui permettrait à l'utilisateur de choisir entre une liste des clients existants et un formulaire classique comme vous l'affichiez par défaut auparavant. Ne vous découragez pas, testez et réussissez !



Note : une telle modification du formulaire va bien évidemment impliquer une modification de l'objet métier qui y est associé, c'est-à-dire **CreationCommandeForm**. En effet, puisque vous proposez la réutilisation d'un client existant à l'utilisateur dans le formulaire, vous devrez faire en sorte dans votre objet métier de ne valider les informations clients que si un nouveau client a été créé, et simplement récupérer l'objet **Client** existant si l'utilisateur choisit un ancien client dans la liste !

Système de suppression des clients et commandes

Pour terminer, il vous suffit ici de créer deux servlets, dédiées chacune à la suppression d'un client et d'une commande de la Map présente en session. Vous contacterez ces servlets via de simples liens HTML depuis vos pages **listerClients.jsp** ou **listerCommandes.jsp** ; il faudra donc y implémenter la méthode `doGet()`. Ces liens contiendront alors, soit un nom de client, soit une date de commande en tant que paramètre d'URL, et les servlets se chargeront alors de retirer l'entrée correspondante de la Map des clients ou des commandes, grâce à la méthode `remove()`.

Dans les exemples de rendus ci-dessus, les croix rouges affichées en fin de chaque ligne des tableaux sont des liens vers les servlets de suppression respectives, que vous pouvez par exemple nommer **SuppressionClient** et **SuppressionCommande**.

Gestion de l'encodage UTF-8 des données

Rien de particulier à vous conseiller ici, il suffit simplement de recopier la déclaration du filtre de Tomcat dans le **web.xml** du projet, et le tour est joué ! 😊

Correction

Cette fois, la longueur du sujet n'est pas trompeuse : le travail que vous devez fournir est bien important ! 😊 Posez-vous calmement les bonnes questions, faites l'analogie avec ce que vous avez appris dans les chapitres de cours et n'oubliez pas les bases que vous avez découvertes auparavant (création d'une servlet, modification du fichier web.xml, utilisation de la JSTL, etc.). 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 !



Pour que cette correction soit entièrement fonctionnelle, vous devrez inclure la bibliothèque jQuery dans le répertoire **/inc** de votre projet, ainsi que le fichier image utilisé pour illustrer le bouton de suppression.

Voici les deux fichiers à télécharger :

- [jquery.js](#)
- [supprimer.png](#)

Le code des vues

Ajout des deux nouveaux liens dans la page **menu.jsp** :

Code : JSP - /inc/menu.jsp

```
<%@ page pageEncoding="UTF-8" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<div id="menu">
    <p><a href="

```

Modification du formulaire de création d'une commande :

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

```
<%@ page pageEncoding="UTF-8" %>
<%@ 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="" />
</head>
<body>
    <c:import url="/inc/menu.jsp" />
    <div>
        <form method="post" action="">
            <fieldset>
                <legend>Informations client</legend>
                <%-- Si et seulement si la Map des clients en session n'est pas vide, alors on propose un choix à l'utilisateur --%>
                <c:if test="${!empty sessionScope.clients}">
                    <label for="choixNouveauClient">Nouveau client ? <span class="requis">*</span></label>
                    <input type="radio" id="choixNouveauClient" name="choixNouveauClient" value="nouveauClient" checked /> Oui
                    <input type="radio" id="choixNouveauClient" name="choixNouveauClient" value="ancienClient" /> Non
                    <br/><br />
                </c:if>

                <c:set var="client" value="${ commande.client }" scope="request" />
                <div id="nouveauClient">
                    <c:import url="/inc/inc_client_form.jsp" />
                </div>

                <%-- Si et seulement si la Map des clients en session n'est pas vide, alors on crée la liste déroulante --%>
                <c:if test="${!empty sessionScope.clients}">
                    <div id="ancienClient">
                        <select name="listeClients" id="listeClients">
                            <option value="">Choisissez un client...</option>
                            <%-- Boucle sur la map des clients --%>
                            <c:forEach items="${ sessionScope.clients }" var="mapClients">
                                <%-- L'expression EL ${mapClients.value} permet de cibler l'objet Client stocké en tant que valeur dans la Map, et on cible ensuite simplement ses propriétés nom et prenom comme on le ferait avec n'importe quel bean. --%>
                                <option value="${ mapClients.value.nom }" ${ mapClients.value.prenom } ${ mapClients.value.nom }></option>
                            </c:forEach>
                        </select>
                    </div>
                </c:if>
            </fieldset>
            <fieldset>
                <legend>Informations commande</legend>

                <label for="dateCommande">Date <span class="requis">*</span></label>
                <input type="text" id="v" name="dateCommande" value="" 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="" 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>

<%-- Inclusion de la bibliothèque jQuery. Vous trouverez des cours sur
JavaScript et jQuery aux adresses suivantes :
    - http://www.siteduzero.com/tutoriel-3-309961-dynamisez-vos-sites-web-avec-javascript.html
    - http://www.siteduzero.com/tutoriel-3-659477-un-site-web-dynamique-avec-jquery.html

```

Si vous ne souhaitez pas télécharger et ajouter jQuery à votre projet, vous pouvez utiliser la version fournie directement en ligne par Google :

```

        <script
src="https://ajax.googleapis.com/ajax/libs/jquery/1.8.0/jquery.min.js"></script>
        --%
        <script src=<c:url value="/inc/jquery.js"/>></script>

<%-- Petite fonction jQuery permettant le remplacement de la première
partie du formulaire par la liste déroulante, au clic sur le bouton radio. --%
        <script>
            jQuery(document).ready(function(){
                /* 1 - Au lancement de la page, on cache le bloc d'éléments du
formulaire correspondant aux clients existants */
                $("div#ancienClient").hide();
                /* 2 - Au clic sur un des deux boutons radio "choixNouveauClient", on
affiche le bloc d'éléments correspondant (nouveau ou ancien client) */
                jQuery('input[name=choixNouveauClient]:radio').click(function()
                    $("div#nouveauClient").hide();
                    $("div#ancienClient").hide();
                    var divId = jQuery(this).val();

```

```

        $("div#" + divId).show();
    });
});
</script>
</body>
</html>
```



Ajout des styles CSS des éléments des tableaux :

Code : CSS - /inc/style.css

```

...
/*
 * Tableaux -----
 */
table{
    border-collapse: collapse;
}
tr.pair{
    background-color: #efefef;
}
tr.impair{
    background-color: #fff;
}
th{
    color: #0568CD;
    border: 1px solid #0568CD;
    padding: 5px;
}
th.action{
    border: 1px solid #900;
    color: #900;
}
td{
    border: 1px solid #ddd;
    padding: 5px;
}
td.action{
    text-align: center;
}
```

Création des pages listant les clients et commandes créés :

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

```

<%@ page pageEncoding="UTF-8" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Liste des clients existants</title>
        <link type="text/css" rel="stylesheet" href="" />
    </head>
    <body>
        <c:import url="/inc/menu.jsp" />
        <div id="corps">
            <c:choose>
                <%-- Si aucun client n'existe en session, affichage d'un message par défaut. --%>
                <c:when test="${empty sessionScope.clients}">
```

```

        <p class="erreur">Aucun client enregistré.</p>
    </c:when>
    <%-- Sinon, affichage du tableau. --%>
    <c:otherwise>
        <table>
            <tr>
                <th>Nom</th>
                <th>Prénom</th>
                <th>Adresse</th>
                <th>Téléphone</th>
                <th>Email</th>
                <th class="action">Action</th>
            </tr>
            <%-- Parcours de la Map des clients en session, et
utilisation de l'objet varStatus. --%>
            <c:forEach items="${ sessionScope.clients }"
var="mapClients" varStatus="boucle">
                <%-- Simple test de parité sur l'index de parcours,
pour alterner la couleur de fond de chaque ligne du tableau. --%>
                <tr class="\${boucle.index % 2 == 0 ? 'pair' :
'impair'}">
                    <%-- Affichage des propriétés du bean Client,
qui est stocké en tant que valeur de l'entrée courante de la map -
-%>
                    <td><c:out value="\${ mapClients.value.nom
}" /></td>
                    <td><c:out value="\${ mapClients.value.prenom
}" /></td>
                    <td><c:out value="\${ mapClients.value.adresse
}" /></td>
                    <td><c:out value="\${ mapClients.value.telephone
}" /></td>
                    <td><c:out value="\${ mapClients.value.email
}" /></td>
                    <%-- Lien vers la servlet de suppression, avec
passage du nom du client - c'est-à-dire la clé de la Map - en
paramètre grâce à la balise <c:param/>. --%>
                    <td class="action">
                        <a href="

```

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

```

<%@ page pageEncoding="UTF-8" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Liste des commandes existantes</title>
        <link type="text/css" rel="stylesheet" href="

```

```
<body>
    <c:import url="/inc/menu.jsp" />
    <div id="corps">
        <c:choose>
            <%-- Si aucune commande n'existe en session, affichage d'un message par défaut. --%>
            <c:when test="${ empty sessionScope.commandes }">
                <p class="erreur">Aucune commande enregistrée.</p>
            </c:when>
            <%-- Sinon, affichage du tableau. --%>
            <c:otherwise>
                <table>
                    <tr>
                        <th>Client</th>
                        <th>Date</th>
                        <th>Montant</th>
                        <th>Mode de paiement</th>
                        <th>Statut de paiement</th>
                        <th>Mode de livraison</th>
                        <th>Statut de livraison</th>
                        <th class="action">Action</th>
                    </tr>
                    <%-- Parcours de la Map des commandes en session, et utilisation de l'objet varStatus. --%>
                    <c:forEach items="${ sessionScope.commandes }"
var="mapCommandes" varStatus="boucle">
                        <%-- Simple test de parité sur l'index de parcours, pour alterner la couleur de fond de chaque ligne du tableau. --%>
                        <tr class="\${boucle.index % 2 == 0 ? 'pair' :
'impaire' }">
                            <%-- Affichage des propriétés du bean Commande, qui est stocké en tant que valeur de l'entrée courante de la map --%>
                            <td><c:out value="\${ mapCommandes.value.client.prenom } ${ mapCommandes.value.client.nom }"/></td>
                            <td><c:out value="\${ mapCommandes.value.date }"/></td>
                            <td><c:out value="\${ mapCommandes.value.montant }"/></td>
                            <td><c:out value="\${ mapCommandes.value.modePaiement }"/></td>
                            <td><c:out value="\${ mapCommandes.value.statutPaiement }"/></td>
                            <td><c:out value="\${ mapCommandes.value.modeLivraison }"/></td>
                            <td><c:out value="\${ mapCommandes.value.statutLivraison }"/></td>
                            <%-- Lien vers la servlet de suppression, avec passage de la date de la commande - c'est-à-dire la clé de la Map - en paramètre grâce à la balise <c:param/>. --%>
                            <td class="action">
                                <a href="" alt="Supprimer" />
                                </a>
                            </td>
                        </tr>
                    </c:forEach>
                </table>
            </c:otherwise>
        </c:choose>
    </div>
</body>
</html>
```

Le code des servlets

Ajout des quatre servlets de suppression et de listage dans le fichier **web.xml** :

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

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
    <filter>
        <filter-name>Set Character Encoding</filter-name>
        <filter-
    class>org.apache.catalina.filters.SetCharacterEncodingFilter</filter-
    class>
        <init-param>
            <param-name>encoding</param-name>
            <param-value>UTF-8</param-value>
        </init-param>
        <init-param>
            <param-name>ignore</param-name>
            <param-value>false</param-value>
        </init-param>
    </filter>
    <filter-mapping>
        <filter-name>Set Character Encoding</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

    <servlet>
        <servlet-name>CreationClient</servlet-name>
        <servlet-class>com.sdzee.tp.servlets.CreationClient</servlet-
    class>
    </servlet>
    <servlet>
        <servlet-name>ListeClients</servlet-name>
        <servlet-class>com.sdzee.tp.servlets.ListeClients</servlet-class>
    </servlet>
    <servlet>
        <servlet-name>SuppressionClient</servlet-name>
        <servlet-
    class>com.sdzee.tp.servlets.SuppressionClient</servlet-class>
    </servlet>
    <servlet>
        <servlet-name>CreationCommande</servlet-name>
        <servlet-
    class>com.sdzee.tp.servlets.CreationCommande</servlet-class>
    </servlet>
    <servlet>
        <servlet-name>ListeCommandes</servlet-name>
        <servlet-class>com.sdzee.tp.servlets.ListeCommandes</servlet-
    class>
    </servlet>
    <servlet>
        <servlet-name>SuppressionCommande</servlet-name>
        <servlet-
    class>com.sdzee.tp.servlets.SuppressionCommande</servlet-class>
    </servlet>

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

```

<servlet-name>SuppressionClient</servlet-name>
<url-pattern>/suppressionClient</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>CreationCommande</servlet-name>
    <url-pattern>/creationCommande</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>ListeCommandes</servlet-name>
    <url-pattern>/listeCommandes</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>SuppressionCommande</servlet-name>
    <url-pattern>/suppressionCommande</url-pattern>
</servlet-mapping>
</web-app>

```

Création des servlets de listage :

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

```

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;

public class ListeClients extends HttpServlet {
    public static final String ATT_CLIENT = "client";
    public static final String ATT_FORM = "form";

    public static final String VUE = "/WEB-INF/listerClients.jsp";

    public void doGet( HttpServletRequest request,
HttpServletResponse response ) throws ServletException, IOException
{
    /* À la réception d'une requête GET, affichage de la liste
des clients */
    this.getServletContext().getRequestDispatcher( VUE
).forward( request, response );
}
}

```

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

```

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;

public class ListeCommandes extends HttpServlet {
    public static final String ATT_COMMANDE = "commande";
    public static final String ATT_FORM = "form";

    public static final String VUE = "/WEB-

```

```

INF/listerCommandes.jsp";

public void doGet( HttpServletRequest request,
HttpServletResponse response ) throws ServletException, IOException
{
    /* À la réception d'une requête GET, affichage de la liste
des commandes */
    this.getServletContext().getRequestDispatcher( VUE
).forward( request, response );
}
}

```

Création des servlets de suppression :

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

```

package com.sdzee.tp.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;
import javax.servlet.http.HttpSession;

import com.sdzee.tp.beans.Client;

public class SuppressionClient extends HttpServlet {

    public static final String PARAM_NOM_CLIENT = "nomClient";
    public static final String SESSION_CLIENTS = "clients";

    public static final String VUE = "/listeClients";

    public void doGet( HttpServletRequest request,
HttpServletResponse response ) throws ServletException, IOException
{
    /* Récupération du paramètre */
    String nomClient = getValeurParametre( request,
PARAM_NOM_CLIENT );

    /* Récupération de la Map des clients enregistrés en
session */
    HttpSession session = request.getSession();
    Map<String, Client> clients = (HashMap<String, Client>)
session.getAttribute( SESSION_CLIENTS );

    /* Si le nom du client et la Map des clients ne sont pas
vides */
    if ( nomClient != null && clients != null ) {
        /* Alors suppression du client de la Map */
        clients.remove( nomClient );
        /* Et remplacement de l'ancienne Map en session par la
nouvelle */
        session.setAttribute( SESSION_CLIENTS, clients );
    }

    /* Redirection vers la fiche récapitulative */
    response.sendRedirect( request.getContextPath() + VUE );
}

/*
* Méthode utilitaire qui retourne null si un paramètre est vide, et

```

```

son
* contenu sinon.
*/
private static String getValeurParametre( HttpServletRequest
request, String nomChamp ) {
    String valeur = request.getParameter( nomChamp );
    if ( valeur == null || valeur.trim().length() == 0 ) {
        return null;
    } else {
        return valeur;
    }
}
}

```

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

```

package com.sdzee.tp.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;
import javax.servlet.http.HttpSession;

import com.sdzee.tp.beans.Commande;

public class SuppressionCommande extends HttpServlet {

    public static final String PARAM_DATE_COMMANDE = "dateCommande";
    public static final String SESSION_COMMANDES = "commandes";

    public static final String VUE =
"/listeCommandes";

    public void doGet( HttpServletRequest request,
HttpServletResponse response ) throws ServletException, IOException
{
    /* Récupération du paramètre */
    String dateCommande = getValeurParametre( request,
PARAM_DATE_COMMANDE );

    /* Récupération de la Map des commandes enregistrées en
session */
    HttpSession session = request.getSession();
    Map<String, Commande> commandes = (HashMap<String,
Commande>) session.getAttribute( SESSION_COMMANDES );

    /* Si la date de la commande et la Map des commandes ne
sont pas vides */
    if ( dateCommande != null && commandes != null ) {
        /* Alors suppression de la commande de la Map */
        commandes.remove( dateCommande );
        /* Et remplacement de l'ancienne Map en session par la
nouvelle */
        session.setAttribute( SESSION_COMMANDES, commandes );
    }

    /* Redirection vers la fiche récapitulative */
    response.sendRedirect( request.getContextPath() + VUE );
}

/*
 * Méthode utilitaire qui retourne null si un paramètre est vide, et

```

```
son
* contenu sinon.
*/
private static String getValeurParametre( HttpServletRequest
request, String nomChamp ) {
    String valeur = request.getParameter( nomChamp );
    if ( valeur == null || valeur.trim().length() == 0 ) {
        return null;
    } else {
        return valeur;
    }
}
```

Modification des servlets existantes pour l'enregistrement en session :

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

```

package com.sdzee.tp.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;
import javax.servlet.http.HttpSession;

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 SESSION_CLIENTS = "clients";

    public static final String VUE_SUCCES      = "/WEB-INF/afficherClient.jsp";
    public static final String VUE_FORM        = "/WEB-INF/creerClient.jsp";

    public void doGet( HttpServletRequest request,
HttpServletResponse response ) throws ServletException, IOException
{
        /* À la réception d'une requête GET, simple affichage du formulaire */
        this.getServletContext().getRequestDispatcher( VUE_FORM )
).forward( request, response );
    }

    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 );
}
}

```

```

    /* Si aucune erreur */
    if ( form.getErreurs().isEmpty() ) {
        /* Alors récupération de la map des clients dans la
        session */
        HttpSession session = request.getSession();
        Map<String, Client> clients = (HashMap<String, Client>)
        session.getAttribute( SESSION_CLIENTS );
        /* Si aucune map n'existe, alors initialisation d'une
        nouvelle map */
        if ( clients == null ) {
            clients = new HashMap<String, Client>();
        }
        /* Puis ajout du client courant dans la map */
        clients.put( client.getNom(), client );
        /* Et enfin (ré)enregistrement de la map en session */
        session.setAttribute( SESSION_CLIENTS, clients );

        /* 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 : Java - com.sdzee.tp.servlets.CreationCommande

```

package com.sdzee.tp.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;
import javax.servlet.http.HttpSession;

import com.sdzee.tp.beans.Client;
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 SESSION_CLIENTS   = "clients";
    public static final String SESSION_COMMANDES = "commandes";

    public static final String VUE_SUCCES       = "/WEB-
INF/afficherCommande.jsp";
    public static final String VUE_FORM         = "/WEB-
INF/creerCommande.jsp";

    public void doGet( HttpServletRequest request,
HttpServletResponse response ) throws ServletException, IOException
{
    /* À la réception d'une requête GET, simple affichage du
formulaire */
    this.getServletContext().getRequestDispatcher( VUE_FORM
).forward( request, response );
}
}

```

```

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 );

    /* Si aucune erreur */
    if ( form.getErreurs().isEmpty() ) {
        /* Alors récupération de la map des clients dans la
session */
        HttpSession session = request.getSession();
        Map<String, Client> clients = (HashMap<String, Client>)
session.getAttribute( SESSION_CLIENTS );
        /* Si aucune map n'existe, alors initialisation d'une
nouvelle map */
        if ( clients == null ) {
            clients = new HashMap<String, Client>();
        }
        /* Puis ajout du client de la commande courante dans la
map */
        clients.put( commande.getClient().getNom(),
commande.getClient() );
        /* Et enfin (ré)enregistrement de la map en session */
        session.setAttribute( SESSION_CLIENTS, clients );

        /* Ensuite récupération de la map des commandes dans la
session */
        Map<String, Commande> commandes = (HashMap<String,
Commande>) session.getAttribute( SESSION_COMMANDES );
        /* Si aucune map n'existe, alors initialisation d'une
nouvelle map */
        if ( commandes == null ) {
            commandes = new HashMap<String, Commande>();
        }
        /* Puis ajout de la commande courante dans la map */
        commandes.put( commande.getDate(), commande );
        /* Et enfin (ré)enregistrement de la map en session */
        session.setAttribute( SESSION_COMMANDES, commandes );

        /* 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 );
    }
}
}

```

Le code des objets métiers

Modification de l'objet gérant la 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 javax.servlet.http.HttpSession;

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_CHOIX_CLIENT = "choixNouveauClient";
    private static final String CHAMP_LISTE_CLIENTS = "listeClients";
    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 ANCIEN_CLIENT = "ancienClient";
    private static final String SESSION_CLIENTS = "clients";
    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 ) {
        Client client;
        /*
         * Si l'utilisateur choisit un client déjà existant, pas de validation à effectuer
         */
        String choixNouveauClient = getValeurChamp( request, CHAMP_CHOIX_CLIENT );
        if ( ANCIEN_CLIENT.equals( choixNouveauClient ) ) {
            /* Récupération du nom du client choisi */
            String nomAncienClient = getValeurChamp( request, CHAMP_LISTE_CLIENTS );
            /* Récupération de l'objet client correspondant dans la session */
            HttpSession session = request.getSession();
        }
    }
}

```

```
        client = ( Map<String, Client>) session.getAttribute(SESSION_CLIENTS ) ).get( nomAncienClient );
    } else {
        /*
 * Sinon on garde l'ancien mode, pour la validation des champs.
 *
 * L'objet métier pour valider la création d'un client existe déjà,
 * il est donc déconseillé de dupliquer ici son contenu ! À 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 = 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_PAITEMENT );
    String statutPaiement = getValeurChamp( request,
CHAMP_STATUT_PAITEMENT );
    String modeLivraison = getValeurChamp( request,
CHAMP_MODE_LIVRAISON );
    String statutLivraison = getValeurChamp( request,
CHAMP_STATUT_LIVRAISON );

    Commande commande = new Commande();

    commande.setClient( client );

    double valeurMontant = -1;
    try {
        valeurMontant = validationMontant( montant );
    } catch ( Exception e ) {
        setErreur( CHAMP_MONTANT, e.getMessage() );
    }
    commande.setMontant( valeurMontant );

    commande.setDate( date );

    try {
        validationModePaiement( modePaiement );
    } catch ( Exception e ) {
        setErreur( CHAMP_MODE_PAITEMENT, e.getMessage() );
    }
    commande.setModePaiement( modePaiement );

    try {
```

```
        validationStatutPaiement( statutPaiement );
    } catch ( Exception e ) {
        setErreur( CHAMP_STATUT_PAITEMENT, e.getMessage() );
    }
    commande.setStatutPaiement( statutPaiement );

    try {
        validationModeLivraison( modeLivraison );
    } catch ( Exception e ) {
        setErreur( CHAMP_MODE_LIVRAISON, e.getMessage() );
    }
    commande.setModelLivraison( 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
paiement." );
    }
}

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;
    }
}
```

Formulaires : l'envoi de fichiers

Nous savons gérer toutes sortes de saisies simples - champs de type texte, case à cocher, liste déroulante, bouton radio, etc. - mais il nous reste encore à traiter le cas du champ de formulaire permettant l'envoi d'un fichier. C'est un gros chapitre qui vous attend : il y a beaucoup de choses à découvrir, prenez le temps de bien assimiler toutes les notions présentées !

Création du formulaire

Pour permettre au visiteur de naviguer et sélectionner un fichier pour envoi via un champ de formulaire, il faut utiliser la balise HTML `<input type="file">`. Pour rappel, et c'est d'ailleurs explicité dans la [spécification HTML](#), pour envoyer un fichier il faut utiliser la méthode POST lors de l'envoi des données du formulaire. En outre, nous y apprenons que l'attribut optionnel `enctype` doit être défini à "`multipart/form-data`".

Sans plus tarder, créons sous le répertoire **/WEB-INF** une page **upload.jsp** qui affichera un tel formulaire à l'utilisateur :

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

```

<%@ page pageEncoding="UTF-8" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Envoi de fichier</title>
        <link type="text/css" rel="stylesheet" href="" />
    </head>
    <body>
        <form action="" method="post" enctype="multipart/form-data">
            <fieldset>
                <legend>Envoi de fichier</legend>

                <label for="description">Description du fichier</label>
                <input type="text" id="description" name="description" value="" />
                <br />

                <label for="fichier">Emplacement du fichier <span class="requis">*</span></label>
                <input type="file" id="fichier" name="fichier" />
                <br />

                <input type="submit" value="Envoyer" class="sansLabel" />
                <br />
            </fieldset>
        </form>
    </body>
</html>

```

Remarquez bien aux lignes 11 et 20 :

- l'utilisation de l'attribut optionnel `enctype`, dont nous n'avions pas besoin dans nos formulaires d'inscription et de connexion puisqu'ils contenaient uniquement des champs classiques ;
- la mise en place d'un champ `<input type="file"/>` dédié à l'envoi de fichiers.

C'est la seule page nécessaire : j'ai réutilisé la même feuille de style CSS que pour nos précédents formulaires.

Récupération des données

Mise en place

Vous commencez à être habitués maintenant : l'étape suivante est la mise en place de la servlet associée à cette page. Il nous faut donc créer une servlet, que nous allons nommer **Upload** et qui est presque vide pour l'instant :

Code : Java - com.sdzee.servlets.Upload

```

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 Upload extends HttpServlet {
    public static final String VUE = "/WEB-INF/upload.jsp";

    public void doGet( HttpServletRequest request, HttpServletResponse response ) throws ServletException, IOException{
        /* Affichage de la page d'envoi de fichiers */
        this.getServletContext().getRequestDispatcher( VUE ).forward(
request, response );
    }

    public void doPost( HttpServletRequest request, HttpServletResponse response ) throws ServletException, IOException{
        /* Méthode vide, pour l'instant... */
    }
}

```

Puis l'associer à la requête HTTP émise par le formulaire, en la déclarant dans le **web.xml** de notre application :

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

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app>
    ...
    <servlet>
        <servlet-name>Upload</servlet-name>
        <servlet-class>com.sdzee.servlets.Upload</servlet-class>
    </servlet>
    ...
    <servlet-mapping>
        <servlet-name>Upload</servlet-name>
        <url-pattern>/upload</url-pattern>
    </servlet-mapping>
</web-app>

```

Avec une telle configuration, nous pouvons accéder au formulaire en nous rendant depuis notre navigateur sur <http://localhost:8080/pro/upload> (voir la figure suivante).

The screenshot shows a simple HTML form titled "Envoi de fichier". It contains two text input fields: one for "Description du fichier" and another for "Emplacement du fichier *". The second field includes a "Parcourir..." button to browse for files. Below the fields is a large "Envoyer" button.

L'étape suivante consiste bien évidemment à compléter notre servlet pour traiter les données reçues !

Traitement des données

Après avoir soumis un tel formulaire, les données envoyées sont dans un format binaire "**multipart**", et sont disponibles dans le corps de la requête POST.

À ce sujet, une subtilité importante mérite d'être portée à votre attention : **ce format de requête n'est pas supporté par les versions de Tomcat antérieures à 7.0.6**. L'explication de ce comportement réside principalement dans :

- la version de l'API servlet utilisée : ce n'est qu'à partir de Java EE 6 que la version 3.0 du conteneur de servlets a été mise en place. L'API en version 2.x ne supporte pas de telles requêtes, elle ne sait gérer que le **enctype** par défaut ;
- un bug dans les premières éditions de Tomcat 7 : aucun problème en ce qui nous concerne, car les versions récentes ont corrigé ce problème.

Avec l'API servlet 2.x

Lorsque des données sont envoyées avec le type **multipart**, les méthodes telles que `request.getParameter()` retournent toutes **null**. Il est en théorie possible d'analyser le corps de la requête vous-mêmes en vous basant sur la méthode `getInputStream()` de l'objet `HttpServletRequest`, mais c'est un vrai travail d'orfèvre qui requiert une parfaite connaissance de la norme [RFC2388](#) !

 Nous n'allons pas étudier en détail la méthode à mettre en place avec cette ancienne version de l'API, mais je vais tout de même vous donner les éléments principaux pour que vous sachiez par où commencer, si jamais vous devez travailler un jour sur une application qui tourne sur une version de l'API antérieure à la 3.0.

La coutume est plutôt d'utiliser [Apache Commons FileUpload](#) pour parser les données **multipart** du formulaire. Cette bibliothèque est une implémentation très robuste de la RFC2388 qui dispose d'excellents [guide utilisateur](#) et [FAQ](#) (ressources en anglais, mais je vous recommande de parcourir les deux attentivement si vous travaillez avec cette version de l'API servlet). Pour l'utiliser, il est nécessaire de placer les fichiers **commons-fileupload.jar** et **commons-io.jar** dans le répertoire **/WEB-INF/lib** de votre application.

Je vous présente ici succinctement le principe général, mais je ne détaille volontairement pas la démarche et ne vous fais pas mettre en place d'exemple pratique : vous allez le découvrir un peu plus bas, nous allons utiliser dans notre projet la démarche spécifique à l'API servlet 3.0. Voici cependant un exemple montrant ce à quoi devrait ressembler la méthode `doPost()` de votre servlet d'upload si vous utilisez [Apache Commons FileUpload](#) :

Code : Java - Exemple d'utilisation de la bibliothèque Apache Commons FileUpload

```
...
import org.apache.commons.fileupload.FileItem;
import org.apache.commons.fileupload.FileUploadException;
import org.apache.commons.fileupload.disk.DiskFileItemFactory;
import org.apache.commons.fileupload.servlet.ServletFileUpload;

...

public void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    try {
        List<FileItem> items = new ServletFileUpload(new
DiskFileItemFactory()).parseRequest(request);
        for (FileItem item : items) {
            if (item.isFormField()) {
                /* Traiter les champs classiques ici (input
type="text|radio|checkbox|etc", select, etc). */
                String nomChamp = item.getFieldName();
                String valeurChamp = item.getString();
                /* ... (traitement à faire) */
            } else {
                /* Traiter les champs de type fichier (input
type="file"). */
                String nomChamp = item.getFieldName();
                String nomFichier =
```

```

        FilenameUtils.getName(item.getName());
        InputStream contenuFichier = item.getInputStream();
        /* ... (traitement à faire) */
    }
}
} catch (FileUploadException e) {
    throw new ServletException("Échec de l'analyse de la requête
multipart.", e);
}

// ...
}

```

Je vous renvoie à la documentation de la bibliothèque si vous souhaitez en apprendre davantage sur son fonctionnement.

En guise d'ouverture pour cette solution, une alternative intéressante à ce système serait d'intégrer tout cela dans un Filter qui analyserait le contenu automatiquement et réinsérerait le tout dans la Map des paramètres de la requête, comme s'il s'agissait d'un champ de formulaire classique, rendant ainsi possible de manière transparente :

- l'utilisation d'un simple `request.getParameter()` comme lors de la récupération d'un paramètre quelconque ;
- l'obtention du fichier uploadé via `request.getAttribute()`.

Vous pouvez trouver un tel exemple sur [cet excellent article](#).

Avec l'API servlet 3.0

En ce qui nous concerne, notre application se base sur l'API servlet 3.0, la solution précédente ne nous est donc pas nécessaire ! Dans cette dernière mouture, une nouvelle méthode `getParts()` est mise à disposition dans l'objet `HttpServletRequest`, et permet de collecter très simplement les éléments de données de type **multipart** ! Auparavant, il était impossible de parvenir à cela simplement sans bibliothèque externe.

 Pour la petite histoire, afin de rendre cette fonctionnalité disponible, la plupart des conteneurs implémentant l'API servlet 3.0 utilisent en réalité le code de la bibliothèque **Apache Commons FileUpload** dans les coulisses ! C'est notamment le cas de Tomcat 7 (Apache) et de GlassFish 3 (Oracle).

Pour commencer, nous devons compléter la déclaration de notre servlet dans le fichier `web.xml` avec une section `<multipart-config>` afin de faire en sorte que la méthode `getParts()` fonctionne :

Code : XML - Extrait du fichier /WEB-INF/web.xml

```

<servlet>
    <servlet-name>Upload</servlet-name>
    <servlet-class>com.sdzee.servlets.Upload</servlet-class>
    <multipart-config>
        <location>c:/fichiers</location>
        <max-file-size>10485760</max-file-size> <!-- 10 Mo -->
        <max-request-size>52428800</max-request-size> <!-- 5 x 10 Mo -->
        <file-size-threshold>1048576</file-size-threshold> <!-- 1 Mo -->
    </multipart-config>
</servlet>

```

Vous remarquez que cette section s'ajoute au sein de la balise de déclaration `<servlet>` de notre servlet d'upload. Voici une rapide description des paramètres optionnels existants :

- `<location>` contient une **URL absolue** vers un répertoire du système. Un chemin relatif au contexte de l'application n'est pas supporté dans cette balise, il s'agit bien là d'un chemin absolu vers le système. Cette URL sera utilisée pour stocker temporairement un fichier lors du traitement des fragments d'une requête, lorsque la taille du fichier est plus grande que la taille spécifiée dans `<file-size-threshold>`. Si vous précisez ici un répertoire qui n'existe pas sur le disque, alors Tomcat enverra une `java.io.IOException` lorsque vous tenterez d'envoyer un fichier plus gros que cette limite ;
- `<file-size-threshold>` précise la taille en octets à partir de laquelle un fichier reçu sera temporairement stocké

sur le disque ;

- **<max-file-size>** précise la taille maximum en octets autorisée pour **un fichier** envoyé. Si la taille d'un fichier envoyé dépasse cette limite, le conteneur enverra une exception. En l'occurrence, Tomcat lancera une `IllegalStateException` ;
- **<max-request-size>** précise la taille maximum en octets autorisée pour **une requête multipart/form-data**. Si la taille totale des données envoyées dans une seule requête dépasse cette limite, le conteneur enverra une exception.

En paramétrant ainsi notre servlet, toutes les données **multipart/form-data** seront disponibles à travers la méthode `request.getParts()`. Celle-ci retourne une collection d'éléments de type `Part`, et doit être utilisée en lieu et place de l'habituelle méthode `request.getParameter()` pour récupérer les contenus des champs de formulaire.

À l'utilisation, il s'avère que c'est bien plus pratique que d'utiliser directement du pur **Apache Commons FileUpload**, comme c'était nécessaire avec les versions antérieures de l'API Servlet ! Par contre, je me répète, mais je viens de vous annoncer que les contenus des champs du formulaire allaient maintenant être disponibles en tant que collection d'éléments de type `Part` et ça, ça va nous poser un petit problème... Car si vous étudiez attentivement l'interface `Part`, vous vous rendez compte qu'elle est plutôt limitée en termes d'abstraction : c'est simple, elle ne propose tout bonnement aucune méthode permettant de déterminer si une donnée reçue renferme un champ classique ou un champ de type fichier !

 Dans ce cas, comment savoir si une requête contient des fichiers ?

Heureusement, il va être facile de nous en sortir par nous-mêmes. Afin de déterminer si les données transmises dans une requête HTTP contiennent d'éventuels fichiers ou non, il suffit d'analyser ses en-têtes. Regardez plutôt ces deux extraits d'en-tête HTTP (commentés) :

Code : HTTP - Exemples de content-disposition dans l'en-tête d'une requête HTTP

```
// Pour un champ <input type="text"> nommé 'description'
Content-Disposition: form-data; name="description"

// Pour un champ <input type="file"> nommé 'fichier'
Content-Disposition: form-
data; name="fichier"; filename="nom_fichier.ext"
```

Comme vous pouvez le constater, la seule différence est la présence d'un attribut nommé **filename**. Il suffit donc de s'assurer qu'un en-tête contient le mot-clé **filename** pour être certain que le fragment traité est un fichier.

 Tout cela est magnifique, mais comment allons-nous récupérer le contenu des en-têtes relatifs à un fragment donné ?

Comme d'habitude, il n'y a pas de miracle : tout est dans la documentation ! 😊 Encore une fois, si vous étudiez attentivement l'interface `Part`, vous constatez qu'elle contient une méthode `part.getHeader()` qui renvoie l'en-tête correspondant à un élément. Exactement ce qu'il nous faut !

Ainsi, nous allons pouvoir examiner le contenu des en-têtes relatifs à un fragment et y vérifier la présence du mot-clé **filename**, afin de savoir si le champ traité est de type fichier ou non.

Lançons-nous, et implémentons un début de méthode `doPost()` dans notre servlet d'upload :

Code : Java - com.sdzee.servlets.Upload

```
...
public static final String CHAMP_DESCRIPTION = "description";
public static final String CHAMP_FICHIER      = "fichier";

...
public void doPost( HttpServletRequest request, HttpServletResponse
response ) throws ServletException, IOException{
/* Récupération du contenu du champ de description */
```

```

String description = request.getParameter( CHAMP_DESCRIPTION );
request.setAttribute( CHAMP_DESCRIPTION, description );

/*
 * Les données reçues sont multipart, on doit donc utiliser la
méthode
* getPart() pour traiter le champ d'envoi de fichiers.
*/
Part part = request.getPart( CHAMP_FICHIER );

/*
* Il faut déterminer s'il s'agit d'un champ classique
* ou d'un champ de type fichier : on délègue cette opération
* à la méthode utilitaire getNomFichier().
*/
String nomFichier = getNomFichier( part );

/*
* Si la méthode a renvoyé quelque chose, il s'agit donc d'un champ
* de type fichier (input type="file").
*/
if ( nomFichier != null && !nomFichier.isEmpty() ) {
    String nomChamp = part.getName();
    request.setAttribute( nomChamp, nomFichier );
}

this.getServletContext().getRequestDispatcher( VUE ).forward(
request, response );
}

/*
* Méthode utilitaire qui a pour unique but d'analyser l'en-tête
"content-disposition",
* et de vérifier si le paramètre "filename" y est présent. Si oui,
alors le champ traité
* est de type File et la méthode retourne son nom, sinon il s'agit
d'un champ de formulaire
* classique et la méthode retourne null.
*/
private static String getNomFichier( Part part ) {
    /* Boucle sur chacun des paramètres de l'en-tête "content-
disposition". */
    for ( String contentDisposition : part.getHeader( "content-
disposition" ).split( ";" ) ) {
        /* Recherche de l'éventuelle présence du paramètre "filename".
*/
        if ( contentDisposition.trim().startsWith("filename") ) {
            /* Si "filename" est présent, alors renvoi de sa valeur,
c'est-à-dire du nom de fichier. */
            return contentDisposition.substring(
contentDisposition.indexOf( '=' ) + 1 );
        }
    }
    /* Et pour terminer, si rien n'a été trouvé... */
    return null;
}

```

Cette ébauche est assez commentée pour que vous puissiez comprendre son fonctionnement sans problème. À la ligne 16 nous accédons au fragment correspondant au champ **fichier** du formulaire, puis nous analysons son en-tête pour déterminer s'il s'agit d'un champ de type fichier ou non.

Je vous donne pour finir des précisions au sujet de la méthode utilitaire `getNomFichier()`. Je vous l'ai annoncé un peu plus tôt, l'en-tête HTTP lu est de la forme :

Code : HTTP - Exemple de content-disposition dans l'en-tête d'une requête HTTP

```
Content-Disposition: form-data; filename="nomdufichier.ext"
```

Afin de sélectionner uniquement la valeur du paramètre **filename**, je réalise dans la méthode utilitaire :

- un `part.getHeader("content-disposition")`, afin de ne traiter que la ligne de l'en-tête concernant le **content-disposition**. Par ailleurs, vous pouvez remarquer que la méthode ne prête aucune attention à la casse dans l'argument que vous lui passez, c'est-à-dire aux éventuelles majuscules qu'il contient : que vous écriviez "content-disposition", "Content-disposition" ou encore "Content-Disposition", c'est toujours le même en-tête HTTP qui sera ciblé ;
- un `split(";")`, afin de distinguer les différents éléments constituant la ligne, séparés comme vous pouvez le constater dans l'exemple d'en-tête ci-dessus par un ";" ;
- un `startsWith("filename")`, afin de ne sélectionner que la chaîne commençant par **filename** ;
- un `substring()`, afin de ne finalement sélectionner que la valeur associée au paramètre **filename**, contenue après le caractère "=".

Avec tous ces détails, vous devriez comprendre parfaitement comment tout cela s'organise.

Modifions alors notre JSP afin d'afficher les valeurs lues et stockées dans les attributs de requêtes **description** et **fichier**, que j'ai mis en place dans notre servlet :

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

```
<%@ page pageEncoding="UTF-8" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Envoi de fichier</title>
        <link type="text/css" rel="stylesheet" href="form.css">
    </head>
    <body>
        <form action="upload" method="post" enctype="multipart/form-data">
            <fieldset>
                <legend>Envoi de fichier</legend>

                <label for="description">Description du fichier</label>
                <input type="text" id="description" name="description" value="" />
                <span class="succes"><c:out value="${description}" /></span>
                <br />

                <label for="fichier">Emplacement du fichier <span class="requis">*</span></label>
                <input type="file" id="fichier" name="fichier" />
                <span class="succes"><c:out value="${fichier}" /></span>
                <br />

                <input type="submit" value="Envoyer" class="sansLabel" />
            <br />
        </fieldset>
    </form>
</body>
</html>
```

Je ne fais ici que réafficher le contenu des champs, à savoir le texte de description et le titre du fichier sélectionné par l'utilisateur, aux lignes 17 et 23. Je n'ai volontairement pas ajouté d'étapes de validation sur ces deux champs, afin de ne pas surcharger le code de ce chapitre inutilement. Après tout, vous savez déjà comment procéder : il suffit de suivre exactement le même principe que lorsque nous avions mis en place des validations sur les champs de nos formulaires d'inscription et de connexion dans les

chapitres précédents.

Nous pouvons dorénavant tester notre ébauche ! J'ai, pour ma part, précisé ces données dans mon formulaire, comme vous pouvez le constater à la figure suivante (à adapter à votre cas selon le fichier que vous allez choisir).

Exemple de données dans le formulaire

Et j'ai obtenu ce résultat (voir la figure suivante).

Rendu après envoi

Pour ceux d'entre vous qui utilisent le navigateur Internet Explorer, vous allez obtenir un rendu sensiblement différent, comme vous pouvez le voir sur la figure suivante.

Rendu après

envoi sous IE

Vous constatez que :

- le nom du fichier a correctement été extrait, mais est entouré de guillemets ;
- avec Internet Explorer, le nom du fichier contient en réalité le chemin complet du fichier sur la machine du client...



Nous obtenons ici un bel exemple nous prouvant que la tambouille interne réalisée dans certains navigateurs peut imposer certaines spécificités ! La morale de l'histoire, c'est qu'il est primordial de tester le fonctionnement d'une application web sous différents navigateurs afin d'éviter ce genre de problèmes d'une plate-forme à une autre...

Afin de pallier les différents petits soucis que nous venons de rencontrer, nous devons apporter quelques modifications à notre servlet.

Dans le bloc **if** traitant le champ **fichier**, pour corriger le bug IE :

Code : Java - com.sdzee.servlets.Upload

```
/*
 * Si la méthode a renvoyé quelque chose, il s'agit donc d'un champ
 * de type fichier (input type="file").
 */
if ( nomFichier != null && !nomFichier.isEmpty() ) {
    String nomChamp = part.getName();

    /*
     * Antibug pour Internet Explorer, qui transmet pour une raison
     * mystique le chemin du fichier local à la machine du client...
     *
     * Ex : C:/dossier/sous-dossier/fichier.ext
    */
}
```

```

/*
* On doit donc faire en sorte de ne sélectionner que le nom et
* l'extension du fichier, et de se débarrasser du superflu.
*/
    nomFichier = nomFichier.substring( nomFichier.lastIndexOf( '/' )
+ 1 ) .substring( nomFichier.lastIndexOf( '\\\\' ) + 1 );
    request.setAttribute( nomChamp, nomFichier );
}

```

Et dans la méthode utilitaire, à la ligne 17 pour retirer les guillemets superflus :

Code : Java - com.sdzee.servlets.Upload

```

/*
* Méthode utilitaire qui a pour unique but d'analyser l'en-tête
* "content-disposition", et de vérifier si le paramètre "filename"
* y est
* présent. Si oui, alors le champ traité est de type File et la
* méthode
* retourne son nom, sinon il s'agit d'un champ de formulaire
* classique et
* la méthode retourne null.
*/
private static String getNomFichier( Part part ) {
    /* Boucle sur chacun des paramètres de l'en-tête "content-
disposition". */
    for ( String contentDisposition : part.getHeader( "content-
disposition" ).split( ";" ) ) {
        /* Recherche de l'éventuelle présence du paramètre
"filename". */
        if ( contentDisposition.trim().startsWith( "filename" ) ) {
            /*
* Si "filename" est présent, alors renvoi de sa valeur,
* c'est-à-dire du nom de fichier sans guillemets.
*/
            return contentDisposition.substring(
contentDisposition.indexOf( '=' ) + 1 ).trim().replace( "\\"", "" );
        }
    }
    /* Et pour terminer, si rien n'a été trouvé... */
    return null;
}

```

Je vous laisse analyser ces deux petites corrections par vous-mêmes, il s'agit uniquement de bricolages sur les `String` qui posaient problème !

Vous pouvez maintenant tester, et vérifier que vous obtenez le résultat indiqué à la figure suivante, peu importe le navigateur utilisé.

The screenshot shows a web form titled "Envoyer un fichier". It contains two input fields. The first field is labeled "Description du fichier" and contains the text "texte". The second field is labeled "Emplacement du fichier *" and contains the text "Parcourir..." followed by the file name "eclipse.ini". Below these fields is a large "Envoyer" button.

La différence entre la théorie et la pratique

Dans le code de notre servlet, j'ai utilisé la méthode `request.getParameter()` pour accéder au contenu du champ

description, qui est un simple champ de type texte. « Logique ! » vous dites-vous, nous avons toujours fait comme ça auparavant et il n'y a pas de raison pour que cela change ! Eh bien détrompez-vous...

Comme je vous l'annonçais dans le court passage sur les moyens à mettre en place avec l'API servlet en version 2.x, avant la version 3.0 un appel à la méthode `request.getParameter()` renvoyait `null` dès lors que le type des données envoyées par un formulaire était **multipart**. Depuis la version 3.0, les spécifications ont changé et nous pouvons maintenant y trouver ce passage au sujet de l'envoi de fichiers :

Citation : Spécifications de l'API servlet 3.0

For parts with form-data as the Content-Disposition, but without a filename, the string value of the part will also be available via the `getParameter` / `getParameterValues` methods on `HttpServletRequest`, using the name of the part.

Pour les non-anglophones, ce passage explicite noir sur blanc que lors de la réception de données issues d'un formulaire, envoyées avec le type **multipart**, la méthode `request.getParameter()` doit pouvoir être utilisée pour récupérer le contenu des champs qui ne sont pas des fichiers.



Et alors, où est le problème ? C'est bien ce que nous avons fait ici, n'est-ce pas ?

Le problème est ici relativement sournois. En effet, ce que nous avons fait fonctionne, mais uniquement parce que nous utilisons le serveur d'applications Tomcat 7 ! Alors qu'habituellement, Tomcat souffre de carences en comparaison à ses confrères comme GlassFish ou JBoss, il se distingue cette fois en respectant à la lettre ce passage de la norme.

Par contre, le serveur considéré comme LA référence des serveurs d'applications Java EE - le serveur GlassFish 3 développé par Oracle - ne remplissait toujours pas cette fonctionnalité il y a quelques mois de ça ! En réalité, il se comportait toujours comme avec les anciennes versions de l'API, ainsi un appel à la méthode `request.getParameter()` retournaient `null`... C'était plutôt étrange, dans la mesure où cette spécification de l'API servlet date tout de même de 2009 ! Heureusement, plus de deux ans après la première sortie de GlassFish en version 3, ce problème qui avait depuis été [rapporté comme un bug](#), a finalement été corrigé dans la dernière version 3.1.2 du serveur.

Bref, vous comprenez maintenant mieux le titre de ce paragraphe : en théorie, tout est censé fonctionner correctement, mais dans la pratique ce n'est pas encore le cas partout, et selon le serveur que vous utilisez il vous faudra parfois ruser pour parvenir à vos fins.



Du coup, comment faire sur un serveur qui ne respecte pas ce passage de la norme ?

Eh bien dans un tel cas, il va falloir que nous récupérons nous-mêmes le contenu binaire de l'élément `Part` correspondant au champ de type texte, et le reconstruire sous forme d'un `String`. Appétissant n'est-ce pas ? 😊



Les explications qui suivent sont uniquement destinées à vous donner un moyen de récupérer les données contenues dans les champs classiques d'un formulaire de type **multipart** dans une application qui tournerait sur un serveur ne respectant pas le point des spécifications que nous venons d'étudier. Si vous n'êtes pas concernés, autrement dit si le serveur que vous utilisez ne présente pas ce bug, vous n'avez pas besoin de mettre en place ces modifications dans votre code, vous pouvez utiliser simplement `request.getParameter()` comme nous l'avons fait dans l'exemple jusqu'à présent.

Ne vous inquiétez pas, nous avons déjà presque tout mis en place dans notre servlet, les modifications vont être minimes ! Dans notre code, nous disposons déjà d'une méthode utilitaire `getNomfichier()` qui nous permet de savoir si un élément `Part` concerne un champ de type fichier ou un champ classique. Le seul vrai travail va être de créer la méthode responsable de la reconstruction dont je viens de vous parler.

Code : Java - com.sdzee.servlets.Upload

```
...
public void doPost( HttpServletRequest request, HttpServletResponse
response ) throws ServletException, IOException {
...
    Part part = request.getPart( CHAMP FICHIER );
```

```

/*
 * Il faut déterminer s'il s'agit d'un champ classique
 * ou d'un champ de type fichier : on délègue cette opération
 * à une méthode utilitaire getNomFichier().
 */
String nomFichier = getNomFichier( part );
if ( nomFichier == null ) {
    /* La méthode a renvoyé null, il s'agit donc d'un champ
    classique ici (input type="text|radio|checkbox|etc", select, etc).
*/
    String nomChamp = part.getName();
    /* Récupération du contenu du champ à l'aide de notre
    nouvelle méthode */
    String valeurChamp = getValeur( part );
    request.setAttribute( nomChamp, valeurChamp );
} else if ( !nomFichier.isEmpty() ) {
    /* La méthode a renvoyé quelque chose, il s'agit donc d'un
    champ de type fichier (input type="file"). */
    ...
}
...
}

/*
 * Méthode utilitaire qui a pour unique but de lire l'InputStream
 * contenu
 * dans l'objet part, et de le convertir en une banale chaîne de
 * caractères.
*/
private String getValeur( Part part ) throws IOException {
    BufferedReader reader = new BufferedReader( new
    InputStreamReader( part.getInputStream(), "UTF-8" ) );
    StringBuilder valeur = new StringBuilder();
    char[] buffer = new char[1024];
    int longueur = 0;
    while ( ( longueur = reader.read( buffer ) ) > 0 ) {
        valeur.append( buffer, 0, longueur );
    }
    return valeur.toString();
}

```

La modification importante dans le code de la méthode `doPost()` est la décomposition de la vérification du retour de la méthode `getNomFichier()` en un `if / else if`. Si la méthode retourne `null`, alors nous savons que nous avons affaire à un champ classique, et nous devons alors récupérer son contenu avec la nouvelle méthode utilitaire `getValeur()` que nous venons de mettre en place.

En ce qui concerne la méthode utilitaire en elle-même, encore une fois c'est du bricolage de flux et de `String`, rien de bien passionnant... du pur Java comme on l'aime chez nous !

Cette nouvelle solution n'apporte aucune nouvelle fonctionnalité, mais offre l'avantage d'être multiplateforme, alors que le code de notre exemple précédent ne fonctionnait que sous certains serveurs. Au final, vous voyez que ce n'est pas si tordu, mais c'est quand même bien moins intuitif et agréable qu'en utilisant directement la méthode `request.getParameter()` : il est ici nécessaire de faire appel à la méthode `getValeur()` sur chaque champ de type texte dont les données sont envoyées à travers un formulaire de type **multipart**. Espérons que les quelques serveurs d'applications qui sont encore à la traîne combleront ce manque rapidement, afin que les développeurs comme vous et moi puissent se passer de cette tambouille peu ragoûtante. 😊

Enregistrement du fichier

Maintenant que nous sommes capables de récupérer les données envoyées depuis notre formulaire, nous pouvons nous attaquer à la gestion du fichier en elle-même : la lecture du fichier envoyé et son écriture sur le disque !

Définition du chemin physique

Commençons par définir un nom de répertoire physique dans lequel nous allons écrire nos fichiers sur le disque. Rappelez-vous : le chemin que nous avons précisé dans le champ `<location>` de la section `<multipart-config>` est uniquement utilisé

par l'API servlet pour stocker les fichiers de manière temporaire. Ce chemin ne sera pas récupérable ailleurs, et donc pas réutilisable.

Il y a plusieurs solutions possibles ici : nous pouvons nous contenter d'écrire en dur le chemin dans une constante directement au sein de notre servlet, ou encore mettre en place un fichier `Properties` dans lequel nous préciserons le chemin. Dans notre exemple, nous allons utiliser un autre moyen : nous allons passer le chemin à notre servlet via un paramètre d'initialisation. Si vous vous souvenez d'un de nos tout premiers chapitres, celui où nous avons découvert la servlet, vous devez également vous souvenir des options de déclaration d'une servlet dans le fichier `web.xml`, notamment d'un bloc nommé `<init-param>`. C'est celui-ci que nous allons mettre en place dans la déclaration de notre servlet d'upload :

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

```

<servlet>
  <servlet-name>Upload</servlet-name>
  <servlet-class>com.sdzee.servlets.Upload</servlet-class>
  <init-param>
    <param-name>chemin</param-name>
    <param-value>/fichiers/</param-value>
  </init-param>
  <multipart-config>
    <location>c:/fichiers</location>
    <max-file-size>10485760</max-file-size> <!-- 10 Mo -->
    <max-request-size>52428800</max-request-size> <!-- 5 x 10 Mo -->
    <file-size-threshold>1048576</file-size-threshold> <!-- 1 Mo -->
  </multipart-config>
</servlet>

```

En procédant ainsi, notre servlet va pouvoir accéder à un paramètre nommé **chemin**, disponible à travers la méthode `getInitParameter()` de l'objet `ServletConfig` !

Écriture du fichier sur le disque

Reprendons maintenant notre servlet pour y récupérer ce fameux **chemin**, et mettre en place proprement l'ouverture des flux dans une méthode dédiée à l'écriture du fichier :

Code : Java - com.sdzee.servlets.Upload

```

...
public static final String CHEMIN          = "chemin";
public static final int TAILLE_TAMPON = 10240; // 10 ko

...
public void doPost( HttpServletRequest request, HttpServletResponse
response ) throws ServletException, IOException {
/*
 * Lecture du paramètre 'chemin' passé à la servlet via la
déclaration
 * dans le web.xml
*/
String chemin = this.getServletConfig().getInitParameter( CHEMIN
);

/* Récupération du contenu du champ de description */
String description = request.getParameter( CHAMP_DESCRIPTION );
request.setAttribute( CHAMP_DESCRIPTION, description );

/*
 * Les données reçues sont multipart, on doit donc utiliser la
méthode
 * getPart() pour traiter le champ d'envoi de fichiers.
*/
Part part = request.getPart( CHAMP_FICHIER );

```

```
/*
 * Il faut déterminer s'il s'agit d'un champ classique
 * ou d'un champ de type fichier : on délègue cette opération
 * à la méthode utilitaire getNomFichier().
 */
String nomFichier = getNomFichier( part );

/*
 * Si la méthode a renvoyé quelque chose, il s'agit donc d'un champ
 * de type fichier (input type="file").
 */
if ( nomFichier != null && !nomFichier.isEmpty() ) {
    String nomChamp = part.getName();
    /*
     * Antibug pour Internet Explorer, qui transmet pour une raison
     * mystique le chemin du fichier local à la machine du client...
     *
     * Ex : C:/dossier/sous-dossier/fichier.ext
     *
     * On doit donc faire en sorte de ne sélectionner que le nom et
     * l'extension du fichier, et de se débarrasser du superflu.
     */
    nomFichier = nomFichier.substring( nomFichier.lastIndexOf(
        '/' ) + 1 )
        .substring( nomFichier.lastIndexOf( '\\' ) + 1 );

    /* Écriture du fichier sur le disque */
    ecrireFichier( part, nomFichier, chemin );
    request.setAttribute( nomChamp, nomFichier );
}

this.getServletContext().getRequestDispatcher( VUE ).forward(
request, response );
}

...
/*
 * Méthode utilitaire qui a pour but d'écrire le fichier passé en
paramètre
 * sur le disque, dans le répertoire donné et avec le nom donné.
*/
private void ecrireFichier( Part part, String nomFichier, String
chemin ) throws IOException {
    /* Prépare les flux. */
    BufferedInputStream entree = null;
    BufferedOutputStream sortie = null;
    try {
        /* Ouvre les flux. */
        entree = new BufferedInputStream( part.getInputStream(),
TAILLE_TAMPON );
        sortie = new BufferedOutputStream( new FileOutputStream( new
File( chemin + nomFichier ) ),
TAILLE_TAMPON );
        /* ... */
    } finally {
        try {
            sortie.close();
        } catch ( IOException ignore ) {
        }
        try {
            entree.close();
        } catch ( IOException ignore ) {
        }
    }
}
```

Ici, nous pourrions très bien utiliser directement les flux de type `InputStream` et `FileOutputStream`, mais les objets `BufferedInputStream` et `BufferedOutputStream` permettent, via l'utilisation d'une mémoire tampon, une gestion plus souple de la mémoire disponible sur le serveur :

- dans le flux **entrée**, il nous suffit de récupérer le flux directement depuis la méthode `getInputStream()` de l'objet `Part`. Nous décorons ensuite ce flux avec un `BufferedInputStream`, avec ici dans l'exemple un tampon de 10 ko ;
- dans le flux **sortie**, nous devons mettre en place un fichier sur le disque, en vue d'y écrire ensuite le contenu de l'entrée. Nous décorons ensuite ce flux avec un `BufferedOutputStream`, avec ici dans l'exemple un tampon de 10 ko.

 Si j'ai pris la peine de vous détailler l'ouverture des flux, c'est pour que vous remarquiez ici la bonne pratique mise en place, que je vous recommande de suivre dès lors que vous manipulez des flux : **toujours ouvrir les flux dans un bloc `try`, et les fermer dans le bloc `finally` associé**. Ainsi, nous nous assurons, quoi qu'il arrive, que la fermeture de nos flux sera bien effectuée !

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

Ceci fait, il ne nous reste plus qu'à mettre en place le tampon et à écrire notre fichier sur le disque. Voici la méthode complétée :

Code : Java - com.sdzee.servlets.Upload

```
/*
 * Méthode utilitaire qui a pour but d'écrire le fichier passé en
 * paramètre
 * sur le disque, dans le répertoire donné et avec le nom donné.
 */
private void ecrireFichier( Part part, String nomFichier, String
chemin ) throws IOException {
    /* Prépare les flux. */
    BufferedInputStream entree = null;
    BufferedOutputStream sortie = null;
    try {
        /* Ouvre les flux. */
        entree = new BufferedInputStream( part.getInputStream() ,
TAILLE_TAMPON );
        sortie = new BufferedOutputStream( new FileOutputStream( new
File( chemin + nomFichier ) ),
TAILLE_TAMPON );

        /*
         * Lit le fichier reçu et écrit son contenu dans un fichier sur le
         * disque.
         */
        byte[] tampon = new byte[TAILLE_TAMPON];
        int longueur;
        while ( ( longueur = entree.read( tampon ) ) > 0 ) {
            sortie.write( tampon, 0, longueur );
        }
    } finally {
        try {
            sortie.close();
        } catch ( IOException ignore ) {
        }
        try {
            entree.close();
        } catch ( IOException ignore ) {
        }
    }
}
```

À l'aide d'un tableau d'octets jouant le rôle de tampon, la boucle ici mise en place parcourt le contenu du fichier reçu et l'écrit morceau par morceau dans le fichier créé sur le disque. Si vous n'êtes pas familiers avec la manipulation de fichiers en Java, ou si vous avez oublié comment cela fonctionne, vous pouvez jeter un œil à [ce chapitre du cours de Java](#).

Test du formulaire d'upload

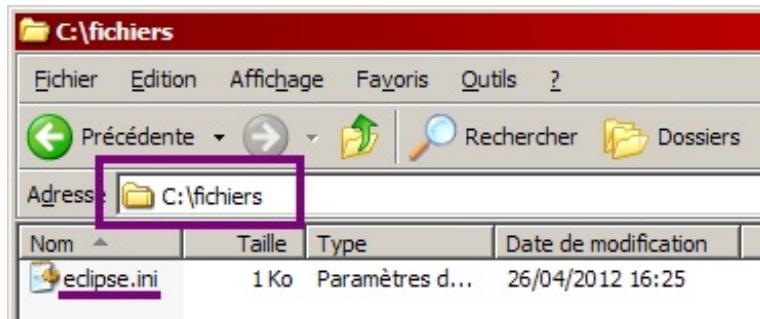
Il ne nous reste maintenant plus qu'à vérifier que tout se déroule correctement.

Avec la configuration mise en place sur mon poste, fonctionnant sous Windows et Tomcat tournant sur le disque C:\, si je reprends les mêmes données que dans les exemples précédents, j'obtiens ce résultat (voir la figure suivante).

Envoi de fichier

Description du fichier	<input type="text" value="texte"/>
Emplacement du fichier *	<input type="text" value="C:\eclipse\eclipse.ini"/> <input type="button" value="Parcourir..."/>
<input type="button" value="Envoyer"/>	

Alors après validation, le répertoire **C:\fichiers** de mon disque contient bien une copie du fichier **eclipse.ini**, comme l'indique la figure suivante.



Ça y est, vous êtes maintenant capables de récupérer des fichiers envoyés par vos utilisateurs ! 😊

Problèmes et limites

Tout cela semble bien joli, mais nous allons un peu vite en besogne. En effet, plusieurs problématiques importantes se posent avec la solution mise en place :

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

Comment gérer les fichiers de mêmes noms ?

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

- ajouter un suffixe à votre fichier si un fichier du même nom existe déjà sur le disque ;
- placer vos fichiers dans des répertoires différents selon leur type ou leur extension ;
- renommer, préfixer ou suffixer vos fichiers par un [timestamp](#) ;
- vous baser sur un hashCode du contenu binaire du fichier pour générer une arborescence et un nom unique ;
- etc.

À vous de définir quelle(s) solution(s) convient(nen)t le mieux aux contraintes de votre projet.

Comment éviter les doublons ?

Là, il s'agit plus d'une optimisation que d'un réel problème. Mais effectivement, si votre application est vouée à être massivement utilisée, vous pourriez éventuellement juger intéressant d'économiser un peu d'espace disque sur votre serveur en ne réenregistrant pas un fichier qui existe déjà sur le disque. Autrement dit, faire en sorte que si un utilisateur envoie un fichier qu'un autre utilisateur a déjà envoyé par le passé, votre application sache le reconnaître rapidement et agir en conséquence.

Comme vous devez vous en douter, c'est un travail un peu plus ardu que la simple gestion des fichiers de mêmes noms que nous venons d'aborder. Toutefois, une des solutions précédentes peut convenir : en vous basant sur un hashCode du contenu binaire du fichier pour générer une arborescence et un nom unique, vous pouvez ainsi à la fois vous affranchir du nom que l'utilisateur a donné à son fichier, et vous assurer qu'un contenu identique ne sera pas dupliqué à deux endroits différents sur votre disque.

Exemple :

1. un utilisateur envoie un fichier nommé **pastèque.jpg** ;
2. votre application le reçoit, et génère un hashCode basé sur son contenu, par exemple **a8cb45e3d6f1dd5e** ;
3. elle stocke alors le fichier dans l'arborescence **/a8cb/45e3/d6f1/dd5e.jpg**, construite à partir du hashCode ;
4. un autre utilisateur envoie plus tard un fichier nommé **watermelon.jpg**, dont le contenu est exactement identique au précédent fichier ;
5. votre application le reçoit, et génère alors le même hashCode que précédemment, puisque le contenu est identique ;
6. elle se rend alors compte que l'arborescence **/a8cb/45e3/d6f1/dd5e.jpg** existe déjà, et saute l'étape d'écriture sur le disque.

Bien évidemment cela demande un peu de réflexion et d'ajustements. Il faudrait en effet générer un hashCode qui soit absolument unique : si deux contenus différents peuvent conduire au même hashCode, alors ce système ne fonctionne plus ! Mais c'est une piste sérieuse qui peut, si elle est bien développée, remplir la mission sans accrocs. 😊

Où stocker les fichiers reçus ?

C'est en effet une bonne question, qui mérite qu'on s'y attarde un instant. A priori, deux possibilités s'offrent à nous :

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

Vous avez ici aveuglément suivi mes consignes, et ainsi implémenté la seconde option. En effet, dans l'exemple j'ai bien enregistré le fichier dans un répertoire placé à la racine de mon disque local. Mais vous devez vous rendre compte que cette solution peut poser un problème important : tous les fichiers placés en dehors de l'application, un peu à la manière des fichiers placés sous son répertoire **/WEB-INF**, sont invisibles au contexte web, c'est-à-dire qu'ils ne sont pas accessibles directement via une URL. Autrement dit, vous ne pourrez pas proposer aux utilisateurs de télécharger ces fichiers !



Dans ce cas, pourquoi ne pas avoir opté pour la première solution ? 😊

Eh bien tout simplement parce que si elle a l'avantage de pouvoir rendre disponibles les fichiers directement aux utilisateurs, puisque tout fichier placé sous la racine d'une application est accessible directement via une URL, elle présente un autre inconvénient : en stockant les fichiers directement dans le conteneur de votre application, vous les rendez vulnérables à un écrasement lors d'un prochain redémarrage serveur ou d'un prochain redéploiement de l'application.

Vous en apprendrez plus à ce sujet dans les annexes du cours, contentez-vous pour le moment de retenir la pratique qui découle de ces contraintes : il est déconseillé de stocker les fichiers uploadés par les utilisateurs dans le conteneur.



Mais alors, comment faire pour rendre nos fichiers externes disponibles au téléchargement ?

La réponse à cette question nécessite un peu de code et d'explications, et vous attend dans le chapitre suivant !

Rendre le tout entièrement automatique

Dans le cas d'un petit formulaire comme celui de notre exemple, tout va bien. Nous sommes là pour apprendre, nous avons le temps de perdre notre temps 😊 à développer des méthodes utilitaires et à réfléchir à des solutions adaptées. Mais dans une vraie application, des formulaires qui contiennent des champs de type fichier, vous risquez d'en rencontrer plus d'un ! Et vous allez vite déchanter quand vous aurez à créer toutes les servlets responsables des traitements...

L'idéal, ça serait de pouvoir continuer à utiliser les méthodes `request.getParameter()` comme si de rien n'était ! Eh bien pour cela, pas de miracle, il faut mettre en place un filtre qui va se charger d'effectuer les vérifications et conversions nécessaires, et qui va rendre disponible le contenu des fichiers simplement. C'est un travail conséquent et plutôt difficile. Plutôt que de vous faire coder le tout à partir de zéro, je vais vous laisser admirer la superbe classe présentée dans [cet excellent article](#). C'est en anglais, mais le code est extrêmement clair et professionnel, essayez d'y jeter un œil et de le comprendre.

Intégration dans MVC

Nous avons déjà fourni beaucoup d'efforts, mais il nous reste encore une étape importante à franchir : nous devons encore intégrer ce que nous venons de mettre en place dans notre architecture MVC. Nous l'avons déjà fait avec le formulaire d'inscription et de connexion, vous devez commencer à être habitués ! Il va nous falloir :

- créer un bean représentant les données manipulées, en l'occurrence il s'agit d'un fichier et de sa description ;
- créer un objet métier qui regroupe les traitements jusqu'à présent réalisés dans la servlet ;
- en profiter pour y ajouter des méthodes de validation de la description et du fichier, et ainsi permettre une gestion fine des erreurs et exceptions ;
- reprendre la servlet pour l'adapter aux objets du modèle fraîchement créés ;
- modifier la page JSP pour qu'elle récupère les nouvelles informations qui lui seront transmises.

Le plus gros du travail va se situer dans l'objet métier responsable des traitements et validations. C'est là que nous allons devoir réfléchir un petit peu, afin de trouver un moyen efficace pour gérer toutes les exceptions possibles lors de la réception de données issues de notre formulaire. Sans plus attendre, voici les codes commentés et expliqués...

Création du bean représentant un fichier

Commençons par le plus simple. Un fichier étant représenté par son nom et sa description, nous avons uniquement besoin d'un bean que nous allons nommer **Fichier**, qui contiendra deux propriétés et que nous allons placer comme ses confrères dans le package `com.sdzee.beans` :

Code : Java - `com.sdzee.beans.Fichier`

```
package com.sdzee.beans;

public class Fichier {

    private String description;
    private String nom;

    public String getDescription() {
        return description;
    }

    public void setDescription( String description ) {
        this.description = description;
    }

    public String getNom() {
        return nom;
    }

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

Jusque-là, ça va...

Création de l'objet métier en charge du traitement du formulaire

Arrivés là, les choses se compliquent un peu pour nous :

- pour assurer la validation des champs, nous allons simplement vérifier que le champ `description` est renseigné et qu'il

- contient au moins 15 caractères, et nous allons vérifier que le champ **fichier** existe bien et contient bien des données ;
- pour assurer la gestion des erreurs, nous allons, comme pour nos systèmes d'inscription et de connexion, initialiser une Map d'erreurs et une chaîne contenant le résultat final du processus.

La difficulté va ici trouver sa source dans la gestion des éventuelles erreurs qui peuvent survenir. À la différence de simples champs texte, la manipulation d'un fichier fait intervenir beaucoup de composants différents, chacun pouvant causer des exceptions bien particulières. Je vous donne le code pour commencer, celui de l'objet métier **UploadForm** placé dans le package `com.sdzee.forms`, et nous en reparlons ensuite. Ne paniquez pas s'il vous semble massif, plus de la moitié des lignes sont des commentaires destinés à vous en faciliter la compréhension ! 😊

Code : Java - com.sdzee.forms.UploadForm

```

package com.sdzee.forms;

import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.util.HashMap;
import java.util.Map;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.Part;

import com.sdzee.beans.Fichier;

public final class UploadForm {
    private static final String CHAMP_DESCRIPTION = "description";
    private static final String CHAMP_FICHIER = "fichier";
    private static final int TAILLE_TAMPON = 10240;
    // 10 ko

    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 Fichier enregistrerFichier( HttpServletRequest request,
    String chemin ) {
        /* Initialisation du bean représentant un fichier */
        Fichier fichier = new Fichier();

        /* Récupération du champ de description du formulaire */
        String description = getValeurChamp( request,
        CHAMP_DESCRIPTION );

        /*
        * Récupération du contenu du champ fichier du formulaire. Il faut
        * ici
        * utiliser la méthode getPart(), comme nous l'avions fait dans
        * notre
        * servlet auparavant.
        */
        String nomFichier = null;
        InputStream contenuFichier = null;
        try {
            Part part = request.getPart( CHAMP_FICHIER );

```

```
        /*
 * Il faut déterminer s'il s'agit bien d'un champ de type fichier :
 * on délègue cette opération à la méthode utilitaire
 * getNomFichier().
 */
        nomFichier = getNomFichier( part );

        /*
 * Si la méthode a renvoyé quelque chose, il s'agit donc d'un
 * champ de type fichier (input type="file").
 */
        if ( nomFichier != null && !nomFichier.isEmpty() ) {
            /*
 * Antibug pour Internet Explorer, qui transmet pour une
 * raison mystique le chemin du fichier local à la machine
 * du client...
 *
 * Ex : C:/dossier/sous-dossier/fichier.ext
 *
 * On doit donc faire en sorte de ne sélectionner que le nom
 * et l'extension du fichier, et de se débarrasser du
 * superflu.
 */
            nomFichier = nomFichier.substring(
nomFichier.lastIndexOf( '/' ) + 1 )
                    .substring( nomFichier.lastIndexOf( '\\' ) +
1 );
        }

        /* Récupération du contenu du fichier */
        contenuFichier = part.getInputStream();

    }
} catch ( IllegalStateException e ) {
/*
 * Exception retournée si la taille des données dépasse les limites
 * définies dans la section <multipart-config> de la déclaration de
 * notre servlet d'upload dans le fichier web.xml
 */
    e.printStackTrace();
    setErreur( CHAMP_FICHIER, "Les données envoyées sont
trop volumineuses." );
} catch ( IOException e ) {
/*
 * Exception retournée si une erreur au niveau des répertoires de
 * stockage survient (répertoire inexistant, droits d'accès
 * insuffisants, etc.)
*/
    e.printStackTrace();
    setErreur( CHAMP_FICHIER, "Erreur de configuration du
serveur." );
} catch ( ServletException e ) {
/*
 * Exception retournée si la requête n'est pas de type
 * multipart/form-data. Cela ne peut arriver que si l'utilisateur
 * essaie de contacter la servlet d'upload par un formulaire
 * différent de celui qu'on lui propose... pirate ! :|
*/
    e.printStackTrace();
    setErreur( CHAMP_FICHIER,
                "Ce type de requête n'est pas supporté, merci
d'utiliser le formulaire prévu pour envoyer votre fichier." );
}

/* Si aucune erreur n'est survenue jusqu'à présent */
if ( erreurs.isEmpty() ) {
    /* Validation du champ de description. */
    try {
        validationDescription( description );
    } catch ( Exception e ) {
        setErreur( CHAMP DESCRIPTION, e.getMessage() );
    }
}
```

```
        }
        fichier.setDescription( description );

        /* Validation du champ fichier. */
        try {
            validationFichier( nomFichier, contenuFichier );
        } catch ( Exception e ) {
            setErreur( CHAMP_FICHIER, e.getMessage() );
        }
        fichier.setNom( nomFichier );
    }

    /* Si aucune erreur n'est survenue jusqu'à présent */
    if ( erreurs.isEmpty() ) {
        /* Écriture du fichier sur le disque */
        try {
            ecrireFichier( contenuFichier, nomFichier, chemin );
        } catch ( Exception e ) {
            setErreur( CHAMP_FICHIER, "Erreur lors de l'écriture
du fichier sur le disque." );
        }
    }

    /* Initialisation du résultat global de la validation. */
    if ( erreurs.isEmpty() ) {
        resultat = "Succès de l'envoi du fichier.";
    } else {
        resultat = "Échec de l'envoi du fichier.";
    }

    return fichier;
}

/*
 * Valide la description saisie.
*/
private void validationDescription( String description ) throws
Exception {
    if ( description != null ) {
        if ( description.length() < 15 ) {
            throw new Exception( "La phrase de description du
fichier doit contenir au moins 15 caractères." );
        }
    } else {
        throw new Exception( "Merci d'entrer une phrase de
description du fichier." );
    }
}

/*
 * Valide le fichier envoyé.
*/
private void validationFichier( String nomFichier, InputStream
contenuFichier ) throws Exception {
    if ( nomFichier == null || contenuFichier == null ) {
        throw new Exception( "Merci de sélectionner un fichier à
envoyer." );
    }
}

/*
 * 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;
    }
}

/*
* Méthode utilitaire qui a pour unique but d'analyser l'en-tête
* "content-disposition", et de vérifier si le paramètre "filename"
y est
* présent. Si oui, alors le champ traité est de type File et la
méthode
* retourne son nom, sinon il s'agit d'un champ de formulaire
classique et
* la méthode retourne null.
*/
    private static String getNomFichier( Part part ) {
        /* Boucle sur chacun des paramètres de l'en-tête "content-
disposition". */
        for ( String contentDisposition : part.getHeader( "content-
disposition" ).split( ";" ) ) {
            /* Recherche de l'éventuelle présence du paramètre
"filename". */
            if ( contentDisposition.trim().startsWith( "filename" ) )
) {
            /*
* Si "filename" est présent, alors renvoi de sa valeur,
* c'est-à-dire du nom de fichier sans guillemets.
*/
                return contentDisposition.substring(
contentDisposition.indexOf( '=' ) + 1 ).trim().replace( "\\"", "" );
}
        }
        /* Et pour terminer, si rien n'a été trouvé... */
        return null;
    }

/*
* Méthode utilitaire qui a pour but d'écrire le fichier passé en
paramètre
* sur le disque, dans le répertoire donné et avec le nom donné.
*/
    private void ecrireFichier( InputStream contenu, String
nomFichier, String chemin ) throws Exception {
        /* Prépare les flux. */
        BufferedInputStream entree = null;
        BufferedOutputStream sortie = null;
        try {
            /* Ouvre les flux. */
            entree = new BufferedInputStream( contenu, TAILLE_TAMPON
);
            sortie = new BufferedOutputStream( new FileOutputStream(
new File( chemin + nomFichier ) ),
TAILLE_TAMPON );
            /*
* Lit le fichier reçu et écrit son contenu dans un fichier sur le
* disque.
*/
            byte[] tampon = new byte[TAILLE_TAMPON];
            int longueur = 0;
            while ( ( longueur = entree.read( tampon ) ) > 0 ) {
                sortie.write( tampon, 0, longueur );
            }
        }
    }
}

```

```
        }
    } finally {
        try {
            sortie.close();
        } catch ( IOException ignore ) {
        }
        try {
            entree.close();
        } catch ( IOException ignore ) {
        }
    }
}
```

Sans surprise, dans ce code vous retrouvez bien :

- les méthodes utilitaires `écrireFichier()` et `getNomFichier()`, que nous avions développées dans notre servlet ;
 - l'analyse des éléments de type `Part` que nous avions mis en place là encore dans notre servlet, pour retrouver le champ contenant le fichier ;
 - l'architecture que nous avions mise en place dans nos anciens objets métiers dans les systèmes d'inscription et de connexion. Notamment la `Map erreurs`, la chaîne `resultat`, les méthodes `getValeurChamp()` et `setErreurs()`, ainsi que la grosse méthode centrale ici nommée `enregistrerFichier()`, qui correspond à la méthode appelée depuis la servlet pour effectuer les traitements.

Comme vous pouvez le constater, la seule différence notable est le nombre de blocs **try** / **catch** qui interviennent, et le nombre de vérifications de la présence d'erreurs. À propos, je ne me suis pas amusé à vérifier trois fois de suite - aux lignes 109, 128 et 138 - si des erreurs avaient eu lieu au cours du processus ou non. C'est simplement afin d'éviter de continuer inutilement le processus de validation si des problèmes surviennent lors des traitements précédents, et également afin de pouvoir renvoyer un message d'erreur précis à l'utilisateur.

Enfin, vous remarquerez pour finir que j'ai modifié légèrement la méthode utilitaire `écrireFichier()`, et que je lui passe désormais le contenu de type `InputStream` renvoyé par la méthode `part.getInputStream()`, et non plus directement l'élément `Part`. En procédant ainsi je peux m'assurer, en amont de la demande d'écriture sur le disque, si le contenu existe et est bien manipulable par notre méthode utilitaire.

Reprise de la servlet

C'est ici un petit travail de simplification, nous devons nettoyer le code de notre servlet pour qu'elle remplisse uniquement un rôle d'aiguilleur :

Code : Java - com.sdzee.servlets.Upload

```
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.Fichier;
import com.sdzee.forms.UploadForm;

public class Upload extends HttpServlet {
    public static final String CHEMIN      = "chemin";
    public static final String ATT_FICHIER = "fichier";
    public static final String ATT_FORM    = "form";
    public static final String VUE         = "/WEB-INF/upload.jsp";
```

```

public void doGet( HttpServletRequest request,
HttpServletResponse response ) throws ServletException, IOException
{
    /* Affichage de la page d'upload */
    this.getServletContext().getRequestDispatcher( VUE
).forward( request, response );
}

public void doPost( HttpServletRequest request,
HttpServletResponse response ) throws ServletException, IOException
{
    /*
     * Lecture du paramètre 'chemin' passé à la servlet via la
     déclaration
     * dans le web.xml
    */
    String chemin = this.getServletConfig().getInitParameter(
CHEMIN );

    /* Préparation de l'objet formulaire */
    UploadForm form = new UploadForm();

    /* Traitement de la requête et récupération du bean en
résultant */
    Fichier fichier = form.enregistrerFichier( request, chemin
);

    /* Stockage du formulaire et du bean dans l'objet request
*/
    request.setAttribute( ATT_FORM, form );
    request.setAttribute( ATT_FICHIER, fichier );

    this.getServletContext().getRequestDispatcher( VUE
).forward( request, response );
}
}

```

L'unique différence avec les anciennes servlets d'inscription et de connexion se situe dans l'appel à la méthode centrale de l'objet métier. Cette fois, nous devons lui passer un argument supplémentaire en plus de l'objet **request** : le **chemin** physique de stockage des fichiers sur le disque local, que nous récupérons - souvenez-vous - via le paramètre d'initialisation défini dans le bloc **<init-param>** de la déclaration de la servlet.

Adaptation de la page JSP aux nouvelles informations transmises

Pour terminer, il faut modifier légèrement le formulaire pour qu'il affiche les erreurs que nous gérons dorénavant grâce à notre objet métier. Vous connaissez le principe, cela reprend là encore les mêmes concepts que ceux que nous avions mis en place dans nos systèmes d'inscription et de connexion :

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

```

<%@ page pageEncoding="UTF-8" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Envoi de fichier</title>
        <link type="text/css" rel="stylesheet" href="" />
    </head>
    <body>
        <form action="

```

```

<legend>Envoi de fichier</legend>

<label for="description">Description du
fichier</label>
<input type="text" id="description"
name="description" value=">"

<span
class="erreur">${form.erreurs['description']}</span>
<br />

<label for="fichier">Emplacement du fichier <span
class="requis">*</span></label>
<input type="file" id="fichier" name="fichier"
value=">" />
<span
class="erreur">${form.erreurs['fichier']}</span>
<br />

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

<p class="${empty form.erreurs ? 'succes' :
'erreur'}">${form.resultat}</p>
</fieldset>
</form>
</body>
</html>

```

Comportement de la solution finale

Vous pouvez maintenant tester votre formulaire sous toutes ses coutures : envoyez des champs vides ou mal renseignés, n'envoyez qu'un champ sur deux, envoyez un fichier trop gros, configurez votre servlet pour qu'elle enregistre les fichiers dans un dossier qui n'existe pas sur votre disque, etc.

Dans le désordre, voici aux figures suivantes un aperçu de quelques rendus que vous devriez obtenir dans différents cas d'échecs.

Envoi de fichier

Description du fichier	<input type="text"/>	Merci d'entrer une phrase de description du fichier.
Emplacement du fichier *	<input type="file"/>	<input type="button" value="Parcourir..."/> Merci de sélectionner un fichier à envoyer.
<input type="button" value="Envoyer"/>		

Échec de l'envoi du fichier.

Envoi sans description ni fichier

Envoi de fichier

Description du fichier	<input type="text"/>
Emplacement du fichier *	<input type="file"/>
<input type="button" value="Parcourir..."/> Les données envoyées sont trop volumineuses.	
<input type="button" value="Envoyer"/>	

Échec de l'envoi du fichier.

Envoi d'un fichier trop lourd

Envoi de fichier

Description du fichier	<input type="text" value="texte"/>	La phrase de description du fichier doit contenir au moins 15 caractères.
Emplacement du fichier *	<input type="file"/>	Parcourir... Merci de sélectionner un fichier à envoyer.
<input type="button" value="Envoyer"/>		

Échec de l'envoi du fichier.

Envoi avec une description trop courte et pas de fichier



Ne prenez pas cette dernière étape à la légère : prenez le temps de bien analyser tout ce qui est géré par notre objet métier, et de bien tester le bon fonctionnement de la gestion du formulaire. Cela vous aidera à bien assimiler tout ce qui intervient dans l'application, ainsi que les relations entre ses différents composants.

- Pour envoyer un fichier depuis un formulaire, il faut utiliser une requête de type **multipart** via `<form ... enctype="multipart/form-data">`.
- Pour récupérer les données d'une telle requête depuis le serveur, l'API Servlet 3.0 fournit la méthode `request.getParts()`.
- Pour rendre disponible cette méthode dans une servlet, il faut compléter sa déclaration dans le fichier web.xml avec une section `<multipart-config>`.
- Pour vérifier si une **Part** contient un fichier, il faut vérifier son type en analysant son en-tête **Content-Disposition**.
- Pour écrire le contenu d'un tel fichier sur le serveur, il suffit ensuite de récupérer le flux via la méthode `part.getInputStream()` et de le manipuler comme on manipulerait un fichier local.
- Lors de la sauvegarde de fichiers sur un serveur, il faut penser aux contraintes imposées par le web : noms de fichiers identiques, contenus de fichiers identiques, faux fichiers, etc.

Le téléchargement de fichiers

Mettons-nous maintenant du côté du client : comment permettre aux utilisateurs de récupérer un fichier présent sur le serveur ? Nous pourrions nous contenter de placer nos documents dans un répertoire du serveur accessible au public, et de leur donner des liens directs vers les fichiers, mais :

- c'est une mauvaise pratique, pour les raisons évoquées dans le chapitre précédent ;
- nous sommes fidèles à MVC, et nous aimons bien tout contrôler : un seul point d'entrée pour les téléchargements, pas cinquante !

C'est dans cette optique que nous allons réaliser une servlet qui aura pour unique objectif de permettre aux clients de télécharger des fichiers.

Une servlet dédiée

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

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

Création de la servlet

Pour commencer nous allons créer une ébauche de servlet, que nous allons nommer **Download** et placer dans `com.sdzee.servlets` :

Code : Java - `com.sdzee.servlets.Download`

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

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

Paramétrage de la servlet

Configurons ensuite l'URL d'accès à notre servlet :

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

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

```
</servlet>
...
<servlet-mapping>
    <servlet-name>Download</servlet-name>
    <url-pattern>/fichiers/*</url-pattern>
</servlet-mapping>
...
```

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

Nous devons maintenant préciser à notre servlet où elle va devoir aller chercher les fichiers sur le disque.



Comment lui faire connaître ce répertoire ?

Il y a plusieurs manières de faire, mais puisque nous avions précisé ce chemin dans le fichier `web.xml` pour la servlet d'upload, nous allons faire de même avec notre servlet de download ! Si votre mémoire est bonne, vous devez vous souvenir d'une balise optionnelle permettant de préciser à une servlet des paramètres d'initialisation... La balise `<init-param>`, ça vous dit quelque chose ? 😊

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

```
...
<servlet>
    <servlet-name>Download</servlet-name>
    <servlet-class>com.sdzee.servlets.Download</servlet-class>
    <init-param>
        <param-name>chemin</param-name>
        <param-value>/fichiers/</param-value>
    </init-param>
</servlet>
...
<servlet-mapping>
    <servlet-name>Download</servlet-name>
    <url-pattern>/fichiers/*</url-pattern>
</servlet-mapping>
...
```

En procédant ainsi, nous mettons à disposition de notre servlet un objet qui contient la valeur spécifiée ! Ainsi, côté servlet il nous suffit de lire la valeur associée depuis notre méthode `doGet()`, vide jusqu'à présent :

Code : Java - com.sdzee.servlets.Download

```
public void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    /* Lecture du paramètre 'chemin' passé à la servlet via la déclaration dans le web.xml */
    String chemin = this.getServletConfig().getInitParameter(
        "chemin" );
}
```

Vous retrouvez ici la méthode permettant l'accès aux paramètres d'initialisation `getInitParameter()`, qui prend en argument le nom du paramètre ciblé.

Analyse du fichier

Maintenant que tout est précisé côté serveur, il nous faut donner au client un moyen de préciser quel fichier il souhaite télécharger. La première idée qui nous vient à l'esprit est évidemment un paramètre de requête, comme nous avons toujours fait jusqu'à présent notamment avec nos formulaires. Oui, mais nous n'allons pas procéder ainsi...



Quel est le problème avec les paramètres de requêtes ?

Le problème, c'est... Internet Explorer, entre autres. Alors que la plupart des navigateurs sont capables de détecter proprement un nom de fichier initialisé dans les en-têtes HTTP, IE ignore tout simplement ce champ et considère lors de l'affichage de la fenêtre "Enregistrer sous..." que le nom du fichier à enregistrer correspond à la fin de l'URL demandée, c'est-à-dire dans notre cas à l'URL de notre servlet de téléchargement ! Autrement dit, il va faire télécharger une page blanche à l'utilisateur. Certains navigateurs sont incapables de détecter correctement le contenu de l'en-tête **Content-Type**.

Bref, afin d'éviter tous ces ennuis dus aux différentes moutures des navigateurs existant, il nous reste un moyen simple et propre de faire passer notre nom de fichier : l'inclure directement dans l'URL. Autrement dit, faire en sorte qu'il nous suffise d'appeler une URL du type <http://localhost:8080/pro/fichiers/test.txt> pour télécharger le fichier nommé **test.txt** !

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

Code : Java - com.sdzee.servlets.Download

```
/* Récupération du chemin du fichier demandé au sein de l'URL de la
requête */
String fichierRequis = request.getPathInfo();
```

Dans la documentation de la méthode `getPathInfo()`, nous remarquons qu'elle retourne **null** si aucun chemin n'existe dans l'URL, et qu'un chemin existant commence toujours par /. Nous devons donc vérifier si un chemin vide est transmis en ajoutant cette condition :

Code : Java - com.sdzee.servlets.Download

```
/* Vérifie qu'un fichier a bien été fourni */
if ( fichierRequis == null || "/".equals( fichierRequis ) ) {
    /* Si non, alors on envoie une erreur 404, qui signifie que la
    ressource demandée n'existe pas */
    response.sendError( HttpServletResponse.SC_NOT_FOUND );
    return;
}
```

Vous remarquez ici l'emploi d'une méthode de l'objet `HttpServletResponse` qui vous était encore inconnue jusque-là : `sendError()`. Elle permet de retourner au client les messages et codes d'erreur HTTP souhaités. Je vous laisse parcourir la documentation et découvrir par vous-mêmes les noms des différentes constantes représentant les codes d'erreur accessibles ; en l'occurrence celui que j'ai utilisé ici correspond à la fameuse erreur 404. Bien entendu, vous pouvez opter pour quelque chose de moins abrupt, en initialisant par exemple un message d'erreur quelconque que vous transmettez ensuite à une page JSP dédiée, pour affichage à l'utilisateur.

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

Code : Java - com.sdzee.servlets.Download

```
/* Décode le nom de fichier récupéré, susceptible de contenir des
espaces et autres caractères spéciaux, et prépare l'objet File */
fichierRequis = URLDecoder.decode( fichierRequis, "UTF-8" );
File fichier = new File( chemin, fichierRequis );
```

```

/* Vérifie que le fichier existe bien */
if ( !fichier.exists() ) {
    /* Si non, alors on envoie une erreur 404, qui signifie que la
       ressource demandée n'existe pas */
    response.sendError(HttpServletRequest.SC_NOT_FOUND);
    return;
}

```

Avant de créer un objet `File` basé sur le chemin du fichier récupéré, il est nécessaire de convertir les éventuels caractères spéciaux qu'il contient à l'aide de la méthode `URLDecoder.decode()`. Une fois l'objet créé, là encore si le fichier n'existe pas sur le disque, j'utilise la méthode `sendError()` pour envoyer une erreur 404 au client et ainsi lui signaler que la ressource demandée n'a pas été trouvée.

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

Code : Java - com.sdzee.servlets.Download

```

/* Récupère le type du fichier */
String type = getServletContext().getMimeType( fichier.getName() );

/* Si le type de fichier est inconnu, alors on initialise un type
   par défaut */
if ( type == null ) {
    type = "application/octet-stream";
}

```

Pour information, les types de fichiers sont déterminés par le conteneur lui-même. Lorsque le conteneur reçoit une requête demandant un fichier et qu'il le trouve, il le renvoie au client. Dans la réponse HTTP renvoyée, il renseigne alors l'en-tête **Content-Type**. Pour ce faire, il se base sur les types MIME dont il a connaissance, en fonction de l'extension du fichier à retourner. Ces types sont spécifiés dans le fichier `web.xml` global du conteneur, qui est situé dans le répertoire `/conf/` du *Tomcat Home*. Si vous l'éditez, vous verrez qu'il en contient déjà une bonne quantité ! En voici un court extrait :

Code : XML - /apache-tomcat-7.0.20/conf/web.xml

```

...
<mime-mapping>
    <extension>jpeg</extension>
    <mimeType>image/jpeg</mimeType>
</mime-mapping>
<mime-mapping>
    <extension>jpg</extension>
    <mimeType>image/jpeg</mimeType>
</mime-mapping>
...

```

Ainsi, il est possible d'ajouter un type inconnu au serveur, il suffit pour cela d'ajouter une section `<mime-mapping>` au fichier. De même, il est possible d'apporter de telles modifications sur le `web.xml` de votre projet web, afin de limiter l'impact des changements effectués à votre application uniquement, et non pas à toute instance de Tomcat lancée sur votre poste.

Bref, dans notre cas nous n'allons pas nous embêter : nous nous contentons de spécifier un type par défaut si l'extension du fichier demandée est inconnue.

Génération de la réponse HTTP

Après tous ces petits traitements, nous avons maintenant tout en main pour initialiser une réponse HTTP et y renseigner les en-têtes nécessaires, à savoir :

- Content-Type ;
- Content-Length ;
- Content-Disposition.

Voici donc le code en charge de l'initialisation de la réponse :

Code : Java - com.sdzee.servlets.Download

```
private static final int DEFAULT_BUFFER_SIZE = 10240; // 10 ko

...
/* Initialise la réponse HTTP */
response.reset();
response.setBufferSize( DEFAULT_BUFFER_SIZE );
response.setContentType( type );
response.setHeader( "Content-Length", String.valueOf(
fichier.length() ) );
response.setHeader( "Content-Disposition", "attachment; filename=\"" +
fichier.getName() + "\"" );
```

Voici quelques explications sur l'enchaînement ici réalisé :

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

Lecture et envoi du fichier

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

Code : Java - com.sdzee.servlets.Download

```
/* Prépare les flux */
BufferedInputStream entree = null;
BufferedOutputStream sortie = null;
try {
    /* Ouvre les flux */
    entree = new BufferedInputStream( new FileInputStream( fichier ),
TAILLE_TAMPON );
    sortie = new BufferedOutputStream( response.getOutputStream(),
TAILLE_TAMPON );

    /* ... */
} finally {
    try {
        sortie.close();
    } catch ( IOException ignore ) {
    }
    try {
        entree.close();
    } catch ( IOException ignore ) {
    }
}
```

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

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



Encore une fois, je prends la peine de vous détailler l'ouverture des flux. N'oubliez jamais de **toujours ouvrir les flux dans un bloc `try`, et de les fermer dans le bloc `finally` associé**.

Ceci fait, il ne nous reste plus qu'à mettre en place un tampon et à envoyer notre fichier au client, le tout depuis notre bloc `try` :

Code : Java - com.sdzee.servlets.Download

```
/* Lit le fichier et écrit son contenu dans la réponse HTTP */
byte[] tampon = new byte[TAILLE_TAMPO];
int longueur;
while ( ( longueur= entree.read( tampon ) ) > 0 ) {
    sortie.write( tampon, 0, longueur );
}
```

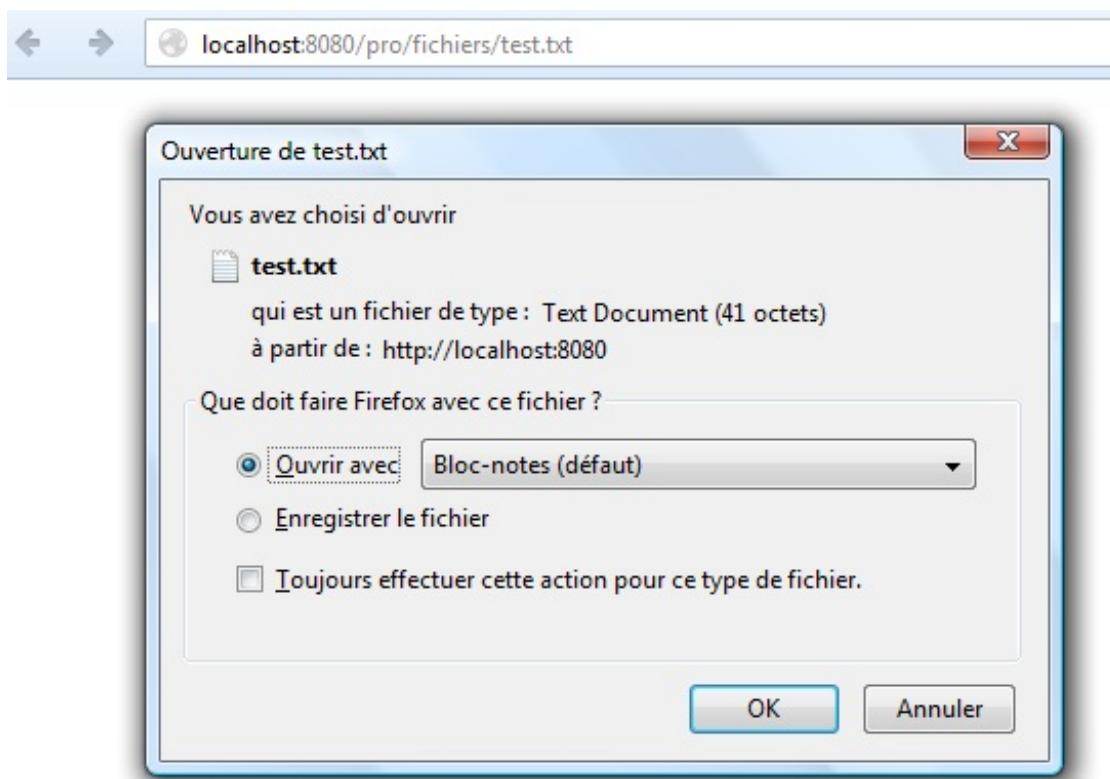
À l'aide d'un tableau d'octets jouant le rôle de tampon, la boucle mise en place parcourt le fichier et l'écrit, morceau par morceau, dans la réponse.

Nous y voilà finalement : notre servlet de téléchargement est opérationnelle ! Vous pouvez télécharger son code intégral en cliquant sur [ce lien](#).

Vérification de la solution

Avec une telle servlet mise en place, les clients peuvent dorénavant télécharger sur leur poste les fichiers qui sont présents sous le répertoire **/fichiers** du disque du serveur sur lequel tourne votre application. Je vous l'ai déjà précisé dans le chapitre précédent, en ce qui me concerne, avec la configuration en place sur mon poste, ce répertoire pointe vers **c:\fichiers**.

Ainsi, si je veux qu'un client puisse télécharger un fichier **test.txt** depuis mon application, il me suffit de le placer dans **c:\fichiers** ! Le client devra alors saisir l'URL <http://localhost:8080/pro/fichiers/test.txt> dans son navigateur et obtiendra le résultat suivant si le fichier existe bien sur mon serveur (voir la figure suivante).



Et il obtiendra logiquement une erreur 404 si le fichier n'existe pas sur le serveur.

De même, vous pouvez vérifier qu'il obtiendra la même erreur s'il essaie d'accéder à la servlet de téléchargement sans préciser un nom de fichier, via l'URL <http://localhost:8080/pro/fichiers/>.

Une solution plus simple

Je vais maintenant vous faire découvrir une manière de faire, bien moins chronophage, mais qui n'existe que sur certains serveurs d'applications.

En effet, si nous utilisions le serveur GlassFish en lieu et place de Tomcat, nous n'aurions tout bonnement pas besoin d'écrire une servlet dédiée au téléchargement de fichiers placés à l'extérieur du conteneur !



Comment est-ce possible ?

Eh bien il se trouve que le serveur d'Oracle est livré avec une fonctionnalité très intéressante qui permet littéralement de "monter" un répertoire externe au conteneur dans une application web : les [Alternate Document Roots](#).

Le principe est relativement simple : il s'agit d'un système qui permet de mettre en place une correspondance automatique entre des URL qui suivent un format particulier, et un répertoire du disque local sur lequel le serveur tourne. Par exemple, pour notre exemple il nous suffirait de définir que toute URL respectant le *pattern* `/pro/fichiers/*` pointe vers son équivalent `c:\fichiers*`. La mise en place de ce mécanisme se fait via l'ajout d'une section dans l'équivalent du fichier web.xml sous GlassFish. Si vous utilisez ce serveur et souhaitez faire le test, je vous laisse parcourir la documentation du système pour plus d'informations, elle est en anglais mais reste très accessible, même pour un débutant. 😊

L'état d'un téléchargement

Notre précédente servlet vous paraît peut-être un peu compliquée, mais elle ne fait en réalité qu'obtenir un `InputStream` de la ressource désirée, et l'écrire dans l'`OutputStream` de la réponse HTTP, accompagné d'en-têtes modifiés. C'est une approche plutôt simpliste, car elle ne permet pas de connaître l'état du téléchargement en cours : autrement dit, en cas de coupure côté client il est impossible de lui proposer la reprise d'un téléchargement en cours.

Pourtant, lorsqu'un utilisateur télécharge un fichier massif et subit un problème de réseau quelconque en cours de route, par exemple à 99% du fichier... il aimerait bien que nous ayons sauvégardé l'état de son précédent téléchargement, et que nous lui proposions à son retour de poursuivre là où il s'était arrêté, et de télécharger uniquement le 1% restant !



Comment proposer la reprise d'un téléchargement ?

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

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



À propos de ce dernier en-tête, note importante : un *timestamp* Java est précis à la milliseconde près, alors que le *timestamp* attendu dans l'en-tête n'est précis qu'à la seconde près. Afin de combler cet écart d'incertitude, il est donc nécessaire d'ajouter une seconde à la date de modification retournée par le client dans sa requête, avant de la traiter.

Bref, vous l'aurez compris la manœuvre devient vite bien plus compliquée dès lors que l'on souhaite réaliser quelque chose de plus évolué et *user-friendly* ! Il n'est rien d'insurmontable pour vous, mais ce travail requiert de la patience, une bonne lecture du protocole HTTP, une bonne logique de traitement dans votre servlet et enfin une campagne de tests très poussée, afin de ne laisser passer aucun cas particulier ni aucune erreur. Quoi qu'il en soit, vous avez ici toutes les informations clés pour mettre en place un tel système !

En ce qui nous concerne, nous allons, dans le cadre de ce cours, nous contenter de notre servlet simpliste, le principe y est posé et l'intérêt pédagogique d'un système plus complexe serait d'autant plus faible. 😊

Réaliser des statistiques

Avec notre servlet en place, nous avons centralisé la gestion des fichiers demandés par les clients, qui passent dorénavant tous par cette unique passerelle de sortie. Si vous envisagez dans une application de réaliser des statistiques sur les téléchargements effectués par les utilisateurs, il est intuitif d'envisager la modification de cette servlet pour qu'elle relève les données souhaitées : combien de téléchargements par fichier, combien de fichiers par utilisateur, combien de téléchargements simultanés, la proportion d'images téléchargées par rapport aux autres types de fichiers, etc. Bref, n'importe quelle information dont vous auriez besoin.

Seulement comme vous le savez, nous essayons de garder nos contrôleurs les plus légers possibles, et de suivre au mieux MVC. Voilà pourquoi cette servlet, unique dans l'application, ne va en réalité pas se charger de cette tâche.



Dans ce cas, où réaliser ce type de traitements ?

Si vous réfléchissez bien à cette problématique, la solution est évidente : le composant idéal pour s'occuper de ce type de traitements, c'est le filtre ! Il suffit en effet d'en appliquer un sur le *pattern /fichiers/** pour que celui-ci ait accès à toutes les demandes de téléchargement effectuées par les clients. Le nombre et la complexité des traitements qu'il devra réaliser dépendront bien évidemment des informations que vous souhaitez collecter. En ce qui concerne la manière, vous savez déjà que tout va passer par la méthode `doFilter()` du filtre en question... 😊

- Le téléchargement de fichiers peut être géré simplement via une servlet dédiée, chargée de faire la correspondance entre le pattern d'URL choisi côté public et l'emplacement du fichier physique côté serveur.
- Elle se charge de transmettre les données lues sur le serveur au navigateur du client via la réponse HTTP.
- Pour permettre un tel transfert, il faut réinitialiser une réponse HTTP, puis définir ses en-têtes **Content-Type**, **Content-Length** et **Content-Disposition**.
- Pour envoyer les données au client, il suffit de lire le fichier comme n'importe quel fichier local et de recopier son contenu dans le flux de sortie accessible via la méthode `response.getOutputStream()`.

TP Fil rouge - Étape 5

Dans cette cinquième étape du fil rouge qui clôture cette partie du cours, vous allez ajouter un champ au formulaire de création d'un client, permettant à l'utilisateur d'envoyer une image. Servlets d'upload et de download sont au programme !

Objectifs

Fonctionnalités

Votre mission cette fois est de permettre à l'utilisateur d'envoyer une image lors de la création d'un client via le formulaire existant. Votre application devra ensuite vérifier que le fichier envoyé est bien une image, et vous en profiterez pour vérifier que le poids de l'image ne dépasse pas 1 Mo avant de l'enregistrer dans un dossier externe au conteneur web.

Bien entendu, l'objectif suivant va être d'afficher un lien vers cette image sur la liste des clients existants. Au clic sur ce lien, l'utilisateur pourra alors visualiser l'image associée au client.

Enfin, vous devrez veiller à ce que l'utilisateur puisse toujours procéder à la création d'un client depuis le formulaire de création d'une commande, comme c'était déjà le cas jusqu'à présent.

Voici aux figures suivantes quelques exemples de rendu.

[Créer un nouveau client](#)

[Créer une nouvelle commande](#)

[Voir les clients existants](#)

[Voir les commandes existantes](#)

Succès de la création du client.

Nom :

Dupond

Prénom :

André

Adresse :

3 rue de l'hirondelle

Numéro de téléphone :

0123456789

Email :

tuto.png

[Créer un nouveau client](#)

[Créer une nouvelle commande](#)

[Voir les clients existants](#)

[Voir les commandes existantes](#)

Nom	Prénom	Adresse	Téléphone	Email	Image	Action
Durand	Marcel	26, boulevard de la mouette	0987654321	marcel.durand@mail.fr		X
Dupond	André	3 rue de l'hirondelle	0123456789		Voir	X

Créer un nouveau client
Créer une nouvelle commande
Voir les clients existants
Voir les commandes existantes

Informations client	
Nom *	<input type="text"/> Merci d'entrer un nom d'utilisateur.
Prénom	<input type="text"/>
Adresse de livraison *	<input type="text"/> Merci d'entrer une adresse de livraison.
Numéro de téléphone *	<input type="text"/> Merci d'entrer un numéro de téléphone.
Adresse email	<input type="text"/>
Image	<input type="file"/> Parcourir... Le fichier envoyé ne doit pas dépasser 1Mo.

Échec de la création du client.

[Valider](#) [Remettre à zéro](#)

À propos de cette dernière capture d'écran, vous remarquerez que lorsque la taille maximale définie pour un fichier est dépassée, toutes les informations saisies dans le reste du formulaire disparaissent. Ne vous inquiétez pas, il s'agit du comportement normal de Tomcat dans ce cas d'utilisation : une `IllegalStateException` est levée, et les appels aux méthodes `getParameter()` renvoient tous `null`. Ce n'est pas très ergonomique, mais nous nous en contenterons dans le cadre de ce TP. 😊



Conseils

Envoi du fichier

Première étape, la modification du formulaire existant. Vous allez devoir reprendre le code du fragment de JSP contenant les champs décrivant un client, et y ajouter un champ de type `<input type="file">` pour permettre à l'utilisateur d'envoyer une image.

En conséquence, vous allez devoir ajouter un attribut `enctype` aux balises `<form>` dans les formulaires des deux JSP responsables de la création d'un client et d'une commande, afin qu'elles gèrent correctement les requêtes contenant des données sous forme de fichiers.

Validation et enregistrement du fichier

Côté serveur, vous allez devoir analyser les données reçues dans le nouveau paramètre de requête correspondant au champ de type fichier. Pour commencer, faites en sorte que ce champ soit optionnel, autrement dit que l'utilisateur puisse le laisser vide, qu'il ne soit pas obligé d'envoyer un fichier lors de la création d'un client.

Si un fichier est envoyé, alors vous allez devoir :

- vérifier que le poids du fichier envoyé ne dépasse pas 1 Mo ;
- vérifier que le fichier envoyé est bien une image ;
- enregistrer le fichier dans un répertoire du disque local, en dehors du conteneur ;
- ajouter le chemin vers l'image dans le bean `Client`, afin de pouvoir retrouver l'image par la suite.

Le poids du fichier

Comme je vous l'ai appris, avec l'API servlet 3.0 c'est très simple : les contraintes de taille sont imposées par la déclaration de la servlet dans le fichier `web.xml`, par l'intermédiaire de la section `<multipart-config>`. C'est donc ici que vous allez devoir limiter à 1 Mo la taille maximale d'un fichier envoyé.

De même, vous savez que Tomcat enverra une **IllegalStateException** en cas de dépassement des limites définies. Vous savez donc ce qu'il vous reste à faire pour renvoyer un message d'erreur précis à l'utilisateur, en cas d'envoi d'un fichier trop volumineux ! 😊

Le type du fichier

Il s'agit de l'étape la plus délicate à réaliser. La solution la plus légère consiste à se baser uniquement sur l'extension du fichier envoyé par l'utilisateur, mais comme vous vous en doutez, ce n'est pas la solution que je vous demande d'adopter. Souvenez-vous : ne faites jamais confiance à l'utilisateur ! Qui vous dit qu'un d'entre eux ne va pas envoyer un prétendu fichier image contenant en réalité un exécutable, une archive, un script ou que sais-je encore ?...

Ainsi, vous allez devoir mettre en place un moyen plus efficace pour déterminer le type réel du fichier transmis. Pas de panique, il existe des bibliothèques qui se chargent de tout cela pour vous ! Je vous conseille ici l'utilisation de **MimeUtil**, car c'est probablement celle qui présente le moins de dépendances externes.

Voici les liens de téléchargement des deux jar nécessaires à son bon fonctionnement :

- [MimeUtil](#)
- [SLF4J](#)

Il vous suffit de les déposer tous deux dans le répertoire **/WEB-INF/lib** de votre projet.

Je vous donne ci-dessous un exemple d'utilisation de la bibliothèque, vérifiant si un fichier est une image :

Code : Java

```
/* Extraction du type MIME du fichier depuis l'InputStream nommé
"contenu" */
MimeUtil.registerMimeTypeDetector(
"eu.medsea.mimeutil.detector.MagicMimeTypeDetector" );
Collection<?> mimeTypes = MimeUtil.getMimeTypes( contenu );

/*
 * Si le fichier est bien une image, alors son en-tête MIME
 * commence par la chaîne "image"
 */
if ( mimeTypes.toString().startsWith( "image" ) ) {
    /* Appeler ici la méthode d'écriture du fichier sur le
    disque... */
} else {
    /* Envoyer ici une exception précisant que le fichier doit être
    une image... */
}
```

L'enregistrement du fichier

Rien de nouveau ici, cela se passe exactement comme nous l'avons fait dans le cours. Vous allez devoir mettre en place une méthode dédiée à l'écriture du fichier sur le disque, en manipulant proprement les flux (n'oubliez pas la traditionnelle structure **try/catch/finally**) et en gérant les différentes erreurs possibles.

Le chemin du fichier

Vous devez, pour terminer, sauvegarder le chemin de l'image dans le bean **Client**. Il vous faudra donc le modifier pour y ajouter une propriété de type **String** que vous pouvez par exemple nommer **image**.

Affichage d'un lien vers l'image

Depuis votre JSP, vous allez devoir récupérer le chemin vers l'image que vous avez placé dans le bean **Client**, et en faire un lien vers la servlet de téléchargement que vous allez par la suite mettre en place.

Pour obtenir le rendu affiché dans le paragraphe précédent, il vous suffit d'ajouter au tableau généré par la page **listerClients.jsp**

une colonne, qui contiendra un lien HTML vers l'image si une image existe, et rien sinon.

Vous pouvez utiliser pour cela une simple condition **<c:if>**, testant si la propriété **image** du bean **Client** est vide ou non. Si elle n'est pas vide, alors vous afficherez un lien dont l'URL pourra par exemple prendre la forme **/pro/images/nomDuFichier.ext**, que vous générerez bien entendu via la balise **<c:url>**.

Ré-affichage de l'image

La dernière étape du TP consiste à créer la servlet de téléchargement des images, qui va se charger de faire la correspondance entre l'URL que vous avez créée dans votre JSP - celle de la forme **/pro/images/nomDuFichier.ext** - et le répertoire du disque local dans lequel sont stockées les images.

Elle va ressembler très fortement à la servlet de download que vous avez mise en place dans le chapitre précédent, à ceci près qu'elle va cette fois uniquement traiter des images ; vous allez donc pouvoir modifier l'en-tête **"Content-Disposition"** de **"attachment"** vers **"inline"**. Ainsi, le navigateur du client va afficher directement l'image après un clic sur le lien "Voir" que vous avez mis en place, et ne va plus ouvrir une fenêtre "Enregistrer sous..." comme c'était le cas avec la servlet de téléchargement de fichiers.

Correction

Faites attention à bien modifier tous les fichiers nécessaires au bon fonctionnement du système, vous pouvez relire les deux précédents chapitres pour vous assurer de ne rien oublier. 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 !

Par ailleurs, vous allez probablement devoir adapter cette correction à la configuration de votre poste, car comme vous le savez, les déclarations des chemins dans le fichier **web.xml** dépendent en partie des répertoires externes que vous utilisez.



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 précédents chapitres. La pratique est très importante, ne vous ruez pas sur la solution !

Le code des objets métiers

Code : Java - com.sdzee.tp.forms.CreationClientForm

```
package com.sdzee.tp.forms;

import java.io.BufferedReader;
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.util.Collection;
import java.util.HashMap;
import java.util.Map;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.Part;

import com.sdzee.tp.beans.Client;
import eu.medsea.mimeutil.MimeUtil;

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 static final String CHAMP_IMAGE    = "imageClient";

    private static final int     TAILLE_TAMPON = 10240;
    // 10ko
```

```
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
chemin ) {
    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 );
    String image = null;

    Client client = new Client();

    try {
        validationNom( nom );
    } catch ( FormValidationException e ) {
        setErreur( CHAMP_NOM, e.getMessage() );
    }
    client.setNom( nom );

    try {
        validationPrenom( prenom );
    } catch ( FormValidationException e ) {
        setErreur( CHAMP_PRENOM, e.getMessage() );
    }
    client.setPrenom( prenom );

    try {
        validationAdresse( adresse );
    } catch ( FormValidationException e ) {
        setErreur( CHAMP_ADRESSE, e.getMessage() );
    }
    client.setAdresse( adresse );

    try {
        validationTelephone( telephone );
    } catch ( FormValidationException e ) {
        setErreur( CHAMP_TELEPHONE, e.getMessage() );
    }
    client.setTelephone( telephone );

    try {
        validationEmail( email );
    } catch ( FormValidationException e ) {
        setErreur( CHAMP_EMAIL, e.getMessage() );
    }
    client.setEmail( email );

    try {
        image = validationImage( request, chemin );
    } catch ( FormValidationException e ) {
        setErreur( CHAMP_IMAGE, e.getMessage() );
    }
    client.setImage( image );

    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
FormValidationException {
    if ( nom != null ) {
        if ( nom.length() < 2 ) {
            throw new FormValidationException( "Le nom
d'utilisateur doit contenir au moins 2 caractères." );
        }
    } else {
        throw new FormValidationException( "Merci d'entrer un
nom d'utilisateur." );
    }
}

private void validationPrenom( String prenom ) throws
FormValidationException {
    if ( prenom != null && prenom.length() < 2 ) {
        throw new FormValidationException( "Le prénom
d'utilisateur doit contenir au moins 2 caractères." );
    }
}

private void validationAdresse( String adresse ) throws
FormValidationException {
    if ( adresse != null ) {
        if ( adresse.length() < 10 ) {
            throw new FormValidationException( "L'adresse de
livraison doit contenir au moins 10 caractères." );
        }
    } else {
        throw new FormValidationException( "Merci d'entrer une
adresse de livraison." );
    }
}

private void validationTelephone( String telephone ) throws
FormValidationException {
    if ( telephone != null ) {
        if ( !telephone.matches( "^\d+$" ) ) {
            throw new FormValidationException( "Le numéro de
téléphone doit uniquement contenir des chiffres." );
        } else if ( telephone.length() < 4 ) {
            throw new FormValidationException( "Le numéro de
téléphone doit contenir au moins 4 chiffres." );
        }
    } else {
        throw new FormValidationException( "Merci d'entrer un
numéro de téléphone." );
    }
}

private void validationEmail( String email ) throws
FormValidationException {
    if ( email != null && !email.matches(
"([^.@]+)(\\.[^.@]+)*@[^.@]+\\.)+([^.@]+)" ) ) {
        throw new FormValidationException( "Merci de saisir une
adresse mail valide." );
    }
}

private String validationImage( HttpServletRequest request,
String chemin ) throws FormValidationException {
/*
 * Récupération du contenu du champ image du formulaire. Il faut ici
 * utiliser la méthode getPart().
 */
}
```

```

String nomFichier = null;
InputStream contenuFichier = null;
try {
    Part part = request.getPart( CHAMP_IMAGE );
    nomFichier = getNomFichier( part );

    /*
    * Si la méthode getNomFichier() a renvoyé quelque chose, il s'agit
    * donc d'un champ de type fichier (input type="file").
    */
    if ( nomFichier != null && !nomFichier.isEmpty() ) {
        /*
        * Antibus pour Internet Explorer, qui transmet pour une raison
        * mystique le chemin du fichier local à la machine du client...
        *
        * Ex : C:/dossier/sous-dossier/fichier.ext
        *
        * On doit donc faire en sorte de ne sélectionner que le nom et
        * l'extension du fichier, et de se débarrasser du superflu.
        */
        nomFichier = nomFichier.substring(
nomFichier.lastIndexOf( '/' ) + 1 )
                            .substring( nomFichier.lastIndexOf( '\\' ) +
1 );

        /* Récupération du contenu du fichier */
        contenuFichier = part.getInputStream();

        /* Extraction du type MIME du fichier depuis
l'InputStream */
        MimeUtil.registerMimeType(
"eu.medsea.mimeutil.detector.MagicMimeTypeDetector" );
        Collection<?> mimeTypes = MimeUtil.getMimeTypes(
contenuFichier );

        /*
        * Si le fichier est bien une image, alors son en-tête MIME
        * commence par la chaîne "image"
        */
        if ( mimeTypes.toString().startsWith( "image" ) ) {
            /* Ecriture du fichier sur le disque */
            ecrireFichier( contenuFichier, nomFichier,
chemin );
        } else {
            throw new FormValidationException( "Le fichier
envoyé doit être une image." );
        }
    }
    } catch ( IllegalStateException e ) {
        /*
        * Exception retournée si la taille des données dépasse les limites
        * définies dans la section <multipart-config> de la déclaration de
        * notre servlet d'upload dans le fichier web.xml
        */
        e.printStackTrace();
        throw new FormValidationException( "Le fichier envoyé ne
doit pas dépasser 1Mo." );
    } catch ( IOException e ) {
        /*
        * Exception retournée si une erreur au niveau des répertoires de
        * stockage survient (répertoire inexistant, droits d'accès
        * insuffisants, etc.)
        */
        e.printStackTrace();
        throw new FormValidationException( "Erreur de
configuration du serveur." );
    } catch ( ServletException e ) {
        /*
        * Exception retournée si la requête n'est pas de type
        * multipart/form-data.
        */
    }
}

```

```

        */
        e.printStackTrace();
        throw new FormValidationException(
            "Ce type de requête n'est pas supporté, merci
d'utiliser le formulaire prévu pour envoyer votre fichier." );
    }

    return nomFichier;
}

/*
* 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;
    }
}

/*
* Méthode utilitaire qui a pour unique but d'analyser l'en-tête
* "content-disposition", et de vérifier si le paramètre "filename"
y est
* présent. Si oui, alors le champ traité est de type File et la
méthode
* retourne son nom, sinon il s'agit d'un champ de formulaire
classique et
* la méthode retourne null.
*/
private static String getNomFichier( Part part ) {
    /* Boucle sur chacun des paramètres de l'en-tête "content-
disposition". */
    for ( String contentDisposition : part.getHeader( "content-
disposition" ).split( ";" ) ) {
        /* Recherche de l'éventuelle présence du paramètre
"filename". */
        if ( contentDisposition.trim().startsWith( "filename" ) )
    }
    /*
    * Si "filename" est présent, alors renvoi de sa valeur,
    * c'est-à-dire du nom de fichier sans guillemets.
    */
    return contentDisposition.substring(
        contentDisposition.indexOf( '=' ) + 1 ).trim().replace( "\\"", "" );
}
    /* Et pour terminer, si rien n'a été trouvé... */
    return null;
}

/*
* Méthode utilitaire qui a pour but d'écrire le fichier passé en
paramètre
* sur le disque, dans le répertoire donné et avec le nom donné.
*/
private void ecrireFichier( InputStream contenuFichier, String

```

```
nomFichier, String chemin )
    throws FormValidationException {
/* Prépare les flux. */
BufferedInputStream entree = null;
BufferedOutputStream sortie = null;
try {
    /* Ouvre les flux. */
    entree = new BufferedInputStream( contenuFichier,
TAILLE_TAMPON );
    sortie = new BufferedOutputStream( new FileOutputStream(
new File( chemin + nomFichier ) ),
TAILLE_TAMPON );

    /*
* Lit le fichier reçu et écrit son contenu dans un fichier sur le
* disque.
*/
        byte[] tampon = new byte[TAILLE_TAMPON];
        int longueur = 0;
        while ( ( longueur = entree.read( tampon ) ) > 0 ) {
            sortie.write( tampon, 0, longueur );
        }
    } catch ( Exception e ) {
        throw new FormValidationException( "Erreur lors de
l'écriture du fichier sur le disque." );
    } finally {
        try {
            sortie.close();
        } catch ( IOException ignore ) {
        }
        try {
            entree.close();
        } catch ( IOException ignore ) {
        }
    }
}
}
```

Code : Java - com.sdzee.tp.forms.CreationClientCommande

```
package com.sdzee.tp.forms;

import java.util.HashMap;
import java.util.Map;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;

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_CHOIX_CLIENT
"choixNouveauClient";
    private static final String CHAMP_LISTE_CLIENTS
"listeClients";
    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_PAIMENT =
"statutPaiementCommande";
    private static final String CHAMP_MODE_LIVRAISON =
"modeLivraisonCommande";
    private static final String CHAMP_STATUT_LIVRAISON =
"statutLivraisonCommande";

    private static final String ANCIEN_CLIENT =
"ancienClient";
    private static final String SESSION_CLIENTS =
    private static final String FORMAT_DATE
HH:mm:ss";

    private String resultat;
    private Map<String, String> erreurs
HashMap<String, String>();

public Map<String, String> getErreurs() {
    return erreurs;
}

public String getResultat() {
    return resultat;
}

public Commande creerCommande( HttpServletRequest request,
String chemin ) {
    Client client;
    /*
    * Si l'utilisateur choisit un client déjà existant, pas de
validation à
    * effectuer
    */
    String choixNouveauClient = getValeurChamp( request,
CHAMP_CHOIX_CLIENT );
    if ( ANCIEN_CLIENT.equals( choixNouveauClient ) ) {
        /* Récupération du nom du client choisi */
        String nomAncienClient = getValeurChamp( request,
CHAMP_LISTE_CLIENTS );
        /* Récupération de l'objet client correspondant dans la
session */
        HttpSession session = request.getSession();
        client = ( (Map<String, Client>) session.getAttribute(
SESSION_CLIENTS ) ).get( nomAncienClient );
    } else {
        /*
        *
        * Sinon on garde l'ancien mode, pour la validation des champs.
        *
        * 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 = clientForm.creerClient( request, chemin );

        /*
        * 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();
DateTimeFormat formatter = DateTimeFormat.forPattern(
FORMAT_DATE );
String date = dt.toString( formatter );

String montant = getValeurChamp( request, CHAMP_MONTANT );
String modePaiement = getValeurChamp( request,
CHAMP_MODE_PAITEMENT );
String statutPaiement = getValeurChamp( request,
CHAMP_STATUT_PAITEMENT );
String modeLivraison = getValeurChamp( request,
CHAMP_MODE_LIVRAISON );
String statutLivraison = getValeurChamp( request,
CHAMP_STATUT_LIVRAISON );

Commande commande = new Commande();

commande.setClient( client );

double valeurMontant = -1;
try {
    valeurMontant = validationMontant( montant );
} catch ( FormValidationException e ) {
    setErreur( CHAMP_MONTANT, e.getMessage() );
}
commande.setMontant( valeurMontant );

commande.setDate( date );

try {
    validationModePaiement( modePaiement );
} catch ( FormValidationException e ) {
    setErreur( CHAMP_MODE_PAITEMENT, e.getMessage() );
}
commande.setModePaiement( modePaiement );

try {
    validationStatutPaiement( statutPaiement );
} catch ( FormValidationException e ) {
    setErreur( CHAMP_STATUT_PAITEMENT, e.getMessage() );
}
commande.setStatutPaiement( statutPaiement );

try {
    validationModeLivraison( modeLivraison );
} catch ( FormValidationException e ) {
    setErreur( CHAMP_MODE_LIVRAISON, e.getMessage() );
}
commande.setModeLivraison( modeLivraison );

try {
    validationStatutLivraison( statutLivraison );
} catch ( FormValidationException 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
FormValidationException {
    double temp;
    if ( montant != null ) {
        try {
            temp = Double.parseDouble( montant );
            if ( temp < 0 ) {
                throw new FormValidationException( "Le montant
doit être un nombre positif." );
            }
        } catch ( NumberFormatException e ) {
            temp = -1;
            throw new FormValidationException( "Le montant doit
être un nombre." );
        }
    } else {
        temp = -1;
        throw new FormValidationException( "Merci d'entrer un
montant." );
    }
    return temp;
}

private void validationModePaiement( String modePaiement )
throws FormValidationException {
    if ( modePaiement != null ) {
        if ( modePaiement.length() < 2 ) {
            throw new FormValidationException( "Le mode de
paiement doit contenir au moins 2 caractères." );
        }
    } else {
        throw new FormValidationException( "Merci d'entrer un
mode de paiement." );
    }
}

private void validationStatutPaiement( String statutPaiement )
throws FormValidationException {
    if ( statutPaiement != null && statutPaiement.length() < 2 )
{
    throw new FormValidationException( "Le statut de
paiement doit contenir au moins 2 caractères." );
}
}

private void validationModeLivraison( String modeLivraison )
throws FormValidationException {
    if ( modeLivraison != null ) {
        if ( modeLivraison.length() < 2 ) {
            throw new FormValidationException( "Le mode de
livraison doit contenir au moins 2 caractères." );
        }
    } else {
        throw new FormValidationException( "Merci d'entrer un
mode de livraison." );
    }
}

private void validationStatutLivraison( String statutLivraison )
throws FormValidationException {
    if ( statutLivraison != null && statutLivraison.length() < 2
) {
    throw new FormValidationException( "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;
        }
    }
}

```

Le code de l'exception personnalisée

Optionnelle, cette exception permet de mieux s'y retrouver dans le code des objets métiers, et d'y reconnaître rapidement les exceptions gérées. L'intérêt principal est on ne peut plus simple : un `throw new FormValidationException(...)` est bien plus explicite qu'un banal `throw new Exception(...)` ! C'est d'autant plus utile que nous allons bientôt faire intervenir une base de données, et ainsi être amenés à gérer d'autres types d'exceptions. En prenant l'habitude de spécialiser vos exceptions, vous rendrez votre code bien plus lisible ! 😊

Code : Java - com.sdzee.tp.forms.FormValidationException

```

package com.sdzee.tp.forms;

public class FormValidationException extends Exception {
    /*
    * Constructeur
    */
    public FormValidationException( String message ) {
        super( message );
    }
}

```

Le code des servlets

Code : XML

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app>
    <filter>
        <filter-name>Set Character Encoding</filter-name>
        <filter-
class>org.apache.catalina.filters.SetCharacterEncodingFilter</filter-
class>
        <init-param>
            <param-name>encoding</param-name>
            <param-value>UTF-8</param-value>
        </init-param>
        <init-param>

```

```
        <param-name>ignore</param-name>
        <param-value>false</param-value>
    </init-param>
</filter>
<filter-mapping>
    <filter-name>Set Character Encoding</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>

<servlet>
    <servlet-name>CreationClient</servlet-name>
    <servlet-class>com.sdzee.tp.servlets.CreationClient</servlet-class>
    <init-param>
        <param-name>chemin</param-name>
        <param-value>/fichiers/images/</param-value>
    </init-param>
    <multipart-config>
        <location>c:/fichiers/images</location>
        <max-file-size>2097152</max-file-size> <!-- 2 Mo -->
        <max-request-size>10485760</max-request-size> <!-- 5 x 2Mo -->
        <file-size-threshold>1048576</file-size-threshold> <!-- 1 Mo -->
    </multipart-config>
</servlet>
<servlet>
    <servlet-name>ListeClients</servlet-name>
    <servlet-class>com.sdzee.tp.servlets.ListeClients</servlet-class>
</servlet>
<servlet>
    <servlet-name>SuppressionClient</servlet-name>
    <servlet-class>com.sdzee.tp.servlets.SuppressionClient</servlet-
class>
</servlet>
<servlet>
    <servlet-name>CreationCommande</servlet-name>
    <servlet-class>com.sdzee.tp.servlets.CreationCommande</servlet-
class>
    <init-param>
        <param-name>chemin</param-name>
        <param-value>/fichiers/images/</param-value>
    </init-param>
    <multipart-config>
        <location>c:/fichiers/images</location>
        <max-file-size>2097152</max-file-size> <!-- 2 Mo -->
        <max-request-size>10485760</max-request-size> <!-- 5 x 2Mo -->
        <file-size-threshold>1048576</file-size-threshold> <!-- 1 Mo -->
    </multipart-config>
</servlet>
<servlet>
    <servlet-name>ListeCommandes</servlet-name>
    <servlet-class>com.sdzee.tp.servlets.ListeCommandes</servlet-class>
</servlet>
<servlet>
    <servlet-name>SuppressionCommande</servlet-name>
    <servlet-class>com.sdzee.tp.servlets.SuppressionCommande</servlet-
class>
</servlet>
<servlet>
    <servlet-name>Image</servlet-name>
    <servlet-class>com.sdzee.tp.servlets.Image</servlet-class>
    <init-param>
        <param-name>chemin</param-name>
        <param-value>/fichiers/images/</param-value>
    </init-param>
</servlet>

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

```

<servlet-mapping>
    <servlet-name>ListeClients</servlet-name>
    <url-pattern>/listeClients</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>SuppressionClient</servlet-name>
    <url-pattern>/suppressionClient</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>CreationCommande</servlet-name>
    <url-pattern>/creationCommande</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>ListeCommandes</servlet-name>
    <url-pattern>/listeCommandes</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>SuppressionCommande</servlet-name>
    <url-pattern>/suppressionCommande</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>Image</servlet-name>
    <url-pattern>/images/*</url-pattern>
</servlet-mapping>
</web-app>

```

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

```

package com.sdzee.tp.servlets;

import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.net.URLDecoder;

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

public class Image extends HttpServlet {
    public static final int TAILLE_TAMPON = 10240; // 10ko

    public void doGet( HttpServletRequest request,
HttpServletResponse response ) throws ServletException, IOException
{
    /*
     * Lecture du paramètre 'chemin' passé à la servlet via la
     * déclaration
     * dans le web.xml
     */
    String chemin = this.getServletConfig().getInitParameter(
"chemin" );

    /*
     * Récupération du chemin du fichier demandé au sein de l'URL de la
     * requête
     */
    String fichierRequis = request.getPathInfo();

    /* Vérifie qu'un fichier a bien été fourni */
    if ( fichierRequis == null || "/".equals( fichierRequis ) )
{
    /*
     * Si non, alors on envoie une erreur 404, qui signifie que la
     */
}
}

```

```
* ressource demandée n'existe pas
*/
    response.sendError( HttpServletResponse.SC_NOT_FOUND );
    return;
}

/*
* Décode le nom de fichier récupéré, susceptible de contenir des
* espaces et autres caractères spéciaux, et prépare l'objet File
*/
fichierRequis = URLDecoder.decode( fichierRequis, "UTF-8" );
File fichier = new File( chemin, fichierRequis );

/* Vérifie que le fichier existe bien */
if ( !fichier.exists() ) {
    /*
* Si non, alors on envoie une erreur 404, qui signifie que la
* ressource demandée n'existe pas
*/
    response.sendError( HttpServletResponse.SC_NOT_FOUND );
    return;
}

/* Récupère le type du fichier */
String type = getServletContext().getMimeType(
fichier.getName() );

/*
* Si le type de fichier est inconnu, alors on initialise un type
par
* défaut
*/
if ( type == null ) {
    type = "application/octet-stream";
}

/* Initialise la réponse HTTP */
response.reset();
response.setBufferSize( TAILLE_TAMPON );
response.setContentType( type );
response.setHeader( "Content-Length", String.valueOf(
fichier.length() ) );
response.setHeader( "Content-Disposition", "inline;
filename=\"" + fichier.getName() + "\"" );

/* Prépare les flux */
BufferedInputStream entree = null;
BufferedOutputStream sortie = null;
try {
    /* Ouvre les flux */
    entree = new BufferedInputStream( new FileInputStream(
fichier ), TAILLE_TAMPON );
    sortie = new BufferedOutputStream(
response.getOutputStream(), TAILLE_TAMPON );

    /* Lit le fichier et écrit son contenu dans la réponse
HTTP */
    byte[] tampon = new byte[TAILLE_TAMPON];
    int longueur;
    while ( ( longueur = entree.read( tampon ) ) > 0 ) {
        sortie.write( tampon, 0, longueur );
    }
} finally {
    try {
        sortie.close();
    } catch ( IOException ignore ) {
    }
    try {
        entree.close();
    } catch ( IOException ignore ) {
```

```

        }
    }
}
}
```

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

```

package com.sdzee.tp.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;
import javax.servlet.http.HttpSession;

import com.sdzee.tp.beans.Client;
import com.sdzee.tp.forms.CreationClientForm;

public class CreationClient extends HttpServlet {
    public static final String CHEMIN           = "chemin";
    public static final String ATT_CLIENT       = "client";
    public static final String ATT_FORM         = "form";
    public static final String SESSION_CLIENTS  = "clients";

    public static final String VUE_SUCCES      = "/WEB-
INF/afficherClient.jsp";
    public static final String VUE_FORM        = "/WEB-
INF/creerClient.jsp";

    public void doGet( HttpServletRequest request,
HttpServletResponse response ) throws ServletException, IOException
{
    /* À la réception d'une requête GET, simple affichage du
formulaire */
    this.getServletContext().getRequestDispatcher( VUE_FORM
).forward( request, response );
}

    public void doPost( HttpServletRequest request,
HttpServletResponse response ) throws ServletException, IOException
{
    /*
    * Lecture du paramètre 'chemin' passé à la servlet via la
déclaration
    * dans le web.xml
    */
    String chemin = this.getServletConfig().getInitParameter(
CHEMIN );

    /* Préparation de l'objet formulaire */
    CreationClientForm form = new CreationClientForm();

    /* Traitement de la requête et récupération du bean en
résultant */
    Client client = form.creerClient( request, chemin );

    /* Ajout du bean et de l'objet métier à l'objet requête */
    request.setAttribute( ATT_CLIENT, client );
    request.setAttribute( ATT_FORM, form );

    /* Si aucune erreur */
}
```

```

        if ( form.getErreurs().isEmpty() ) {
            /* Alors récupération de la map des clients dans la
            session */
            HttpSession session = request.getSession();
            Map<String, Client> clients = (HashMap<String, Client>)
            session.getAttribute( SESSION_CLIENTS );
            /* Si aucune map n'existe, alors initialisation d'une
            nouvelle map */
            if ( clients == null ) {
                clients = new HashMap<String, Client>();
            }
            /* Puis ajout du client courant dans la map */
            clients.put( client.getNom(), client );
            /* Et enfin (ré)enregistrement de la map en session */
            session.setAttribute( SESSION_CLIENTS, clients );

            /* 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 : Java - com.sdzee.tp.servlets.CreationCommande

```

package com.sdzee.tp.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;
import javax.servlet.http.HttpSession;

import com.sdzee.tp.beans.Client;
import com.sdzee.tp.beans.Commande;
import com.sdzee.tp.forms.CreationCommandeForm;

public class CreationCommande extends HttpServlet {
    public static final String CHEMIN = "chemin";
    public static final String ATT_COMMANDE = "commande";
    public static final String ATT_FORM = "form";
    public static final String SESSION_CLIENTS = "clients";
    public static final String SESSION_COMMANDES = "commandes";

    public static final String VUE_SUCCES = "/WEB-INF/afficherCommande.jsp";
    public static final String VUE_FORM = "/WEB-INF/creerCommande.jsp";

    public void doGet( HttpServletRequest request,
HttpServletResponse response ) throws ServletException, IOException
{
    /* À la réception d'une requête GET, simple affichage du
formulaire */
    this.getServletContext().getRequestDispatcher( VUE_FORM
).forward( request, response );
}

```

```
}

public void doPost( HttpServletRequest request,
HttpServletResponse response ) throws ServletException, IOException
{
    /*
     * Lecture du paramètre 'chemin' passé à la servlet via la
     déclaration
     * dans le web.xml
    */
    String chemin = this.getServletConfig().getInitParameter(
CHEMIN );

    /* Préparation de l'objet formulaire */
    CreationCommandeForm form = new CreationCommandeForm();

    /* Traitement de la requête et récupération du bean en
résultant */
    Commande commande = form.creerCommande( request, chemin );

    /* Ajout du bean et de l'objet métier à l'objet requête */
    request.setAttribute( ATT_COMMANDE, commande );
    request.setAttribute( ATT_FORM, form );

    /* Si aucune erreur */
    if ( form.getErreurs().isEmpty() ) {
        /* Alors récupération de la map des clients dans la
session */
        HttpSession session = request.getSession();
        Map<String, Client> clients = (HashMap<String, Client>)
session.getAttribute( SESSION_CLIENTS );
        /* Si aucune map n'existe, alors initialisation d'une
nouvelle map */
        if ( clients == null ) {
            clients = new HashMap<String, Client>();
        }
        /* Puis ajout du client de la commande courante dans la
map */
        clients.put( commande.getClient().getNom(),
commande.getClient() );
        /* Et enfin (ré)enregistrement de la map en session */
        session.setAttribute( SESSION_CLIENTS, clients );

        /* Ensuite récupération de la map des commandes dans la
session */
        Map<String, Commande> commandes = (HashMap<String,
Commande>) session.getAttribute( SESSION_COMMANDES );
        /* Si aucune map n'existe, alors initialisation d'une
nouvelle map */
        if ( commandes == null ) {
            commandes = new HashMap<String, Commande>();
        }
        /* Puis ajout de la commande courante dans la map */
        commandes.put( commande.getDate(), commande );
        /* Et enfin (ré)enregistrement de la map en session */
        session.setAttribute( SESSION_COMMANDES, commandes );

        /* 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 );
    }
}
```

Le code des JSP

Code : JSP - listerClients.jsp

```

<%@ page pageEncoding="UTF-8" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Liste des clients existants</title>
        <link type="text/css" rel="stylesheet" href="" />
    </head>
    <body>
        <c:import url="/inc/menu.jsp" />
        <div id="corps">
            <c:choose>
                <%-- Si aucun client n'existe en session, affichage d'un message par défaut. --%>
                <c:when test="${ empty sessionScope.clients }">
                    <p class="erreur">Aucun client enregistré.</p>
                </c:when>
                <%-- Sinon, affichage du tableau. --%>
                <c:otherwise>
                    <table>
                        <tr>
                            <th>Nom</th>
                            <th>Prénom</th>
                            <th>Adresse</th>
                            <th>Téléphone</th>
                            <th>Email</th>
                            <th>Image</th>
                            <th class="action">Action</th>
                        </tr>
                        <%-- Parcours de la Map des clients en session, et utilisation de l'objet varStatus. --%>
                        <c:forEach items="${ sessionScope.clients }" var="mapClients" varStatus="boucle">
                            <%-- Simple test de parité sur l'index de parcours, pour alterner la couleur de fond de chaque ligne du tableau. --%>
                            <tr class="\${boucle.index \% 2 == 0 ? 'pair' : 'impair'}">
                                <%-- Affichage des propriétés du bean Client, qui est stocké en tant que valeur de l'entrée courante de la map --%>
                                <td><c:out value="\${ mapClients.value.nom }" /></td>
                                <td><c:out value="\${ mapClients.value.prenom }" /></td>
                                <td><c:out value="\${ mapClients.value.adresse }" /></td>
                                <td><c:out value="\${ mapClients.value.telephone }" /></td>
                                <td><c:out value="\${ mapClients.value.email }" /></td>
                                <td>
                                    <%-- On ne construit et affiche un lien vers l'image que si elle existe. --%>
                                    <c:if test="\${ !empty mapClients.value.image }">
                                        <c:set var="image"><c:out value="\${ mapClients.value.image }" /></c:set>
                                        <a href="

```

```

        </c:if>
    </td>
    <%-- Lien vers la servlet de suppression, avec
    passage du nom du client - c'est-à-dire la clé de la Map - en
    paramètre grâce à la balise <c:param/>. --%>
    <td class="action">
        <a href=<c:url
value="/suppressionClient"><c:param name="nomClient" value="${mapClients.key }" /></c:url>">
            <img src=<c:url
value="/inc/supprimer.png"/> alt="Supprimer" />
        </a>
    </td>
</tr>
</c:forEach>
</table>
</c:otherwise>
</c:choose>
</div>
</body>
</html>

```

Code : JSP - /inc/inc_client_form.jsp

```

<%@ page pageEncoding="UTF-8" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<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']}

```

Code : JSP - creerClient.jsp

```

<%@ page pageEncoding="UTF-8" %>
<%@ 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="" />
    </head>
    <body>
        <c:import url="/inc/menu.jsp" />
        <div>
            <form method="post" action="" enctype="multipart/form-data">
                <fieldset>
                    <legend>Informations client</legend>
                    <c:import url="/inc/inc_client_form.jsp" />
                </fieldset>
                <p class="info">${ form.resultat }</p>
                <input type="submit" value="Valider" />
                <input type="reset" value="Remettre à zéro" /> <br>
            </form>
        </div>
    </body>
</html>

```

Code : JSP - creerCommande.jsp

```

<%@ page pageEncoding="UTF-8" %>
<%@ 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="" />
    </head>
    <body>
        <c:import url="/inc/menu.jsp" />
        <div>
            <form method="post" action="" enctype="multipart/form-data">
                <fieldset>
                    <legend>Informations client</legend>
                    <%-- Si et seulement si la Map des clients en session n'est pas vide, alors on propose un choix à l'utilisateur --%>
                    <c:if test="${ !empty sessionScope.clients }">
                        <label for="choixNouveauClient">Nouveau client ? <span class="requis">*</span></label>
                        <input type="radio" id="choixNouveauClient" name="choixNouveauClient" value="nouveauClient" checked /> Oui
                        <input type="radio" id="choixNouveauClient" name="choixNouveauClient" value="ancienClient" /> Non
                        <br/><br />
                    </c:if>
                    <c:set var="client" value="${ commande.client }" scope="request" />
                    <div id="nouveauClient">

```

```

        <c:import url="/inc/inc_client_form.jsp" />
</div>

        <%-- Si et seulement si la Map des clients en session n'est pas vide, alors on crée la liste déroulante --%>
        <c:if test="${!empty sessionScope.clients}">
<div id="ancienClient">
    <select name="listeClients" id="listeClients">
        <option value="">Choisissez un client...</option>
        <%-- Boucle sur la map des clients --%>
        <c:forEach items="${sessionScope.clients}" var="mapClients">
            <%-- L'expression EL ${mapClients.value} permet de cibler l'objet Client stocké en tant que valeur dans la Map, et on cible ensuite simplement ses propriétés nom et prenom comme on le ferait avec n'importe quel bean. --%>
            <option value="${mapClients.value.nom}">${mapClients.value.prenom} ${mapClients.value.nom}</option>
        </c:forEach>
    </select>
</div>
</c:if>
</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>

<%-- Inclusion de la bibliothèque jQuery. Vous trouverez des cours sur
JavaScript et jQuery aux adresses suivantes :
    - http://www.siteduzero.com/tutoriel-3-309961-dynamisez-vos-sites-web-avec-javascript.html
    - http://www.siteduzero.com/tutoriel-3-659477-un-site-web-dynamique-avec-jquery.html

Si vous ne souhaitez pas télécharger et ajouter jQuery à votre
projet, vous pouvez utiliser la version fournie directement en ligne par Google
:
<script
src="https://ajax.googleapis.com/ajax/libs/jquery/1.8.0/jquery.min.js"></script>
--%>
<script src=""></script>

<%-- Petite fonction jQuery permettant le remplacement de la première
partie du formulaire par la liste déroulante, au clic sur le bouton radio. --%>
<script>
jQuery(document).ready(function(){
    /* 1 - Au lancement de la page, on cache le bloc d'éléments du
formulaire correspondant aux clients existants */
    $("div#ancienClient").hide();
    /* 2 - Au clic sur un des deux boutons radio "choixNouveauClient", on
affiche le bloc d'éléments correspondant (nouveau ou ancien client) */
    $("input[name=choixNouveauClient]:radio").click(function()
        $("div#nouveauClient").hide();
        $("div#ancienClient").hide();
        var divId = jQuery(this).val();
        $("div#" + divId).show();
    });
});
</script>
</body>
</html>
```

Code : JSP - afficherClient.jsp

```

<%@ page pageEncoding="UTF-8" %>
<%@ 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="" />
    </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>
<p>Image : <c:out value="${ client.image }"/></p>
</div>
</body>
</html>

```

Code : JSP - afficherCommande.jsp

```

<%@ page pageEncoding="UTF-8" %>
<%@ 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="" />
    </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>Image : <c:out value="${ commande.client.image }"/></p>
            <p>Commande</p>
            <p>Date : <c:out value="${ 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>

```

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...

Partie 5 : Les bases de données avec Java EE

Cette partie est consacrée entièrement à la gestion des données dans une application Java EE. Nous commencerons par rappeler de quoi est constituée une base de données, comment la faire interagir avec notre application, puis nous étudierons les requêtes principales, avant de découvrir que notre modèle devra être divisé en deux couches distinctes : la couche métier et la couche données.

Introduction à MySQL et JDBC

Ce chapitre d'introduction est une entrée en matière qui a pour objectif de vous présenter rapidement les raisons d'être et les composants d'une base de données, vous guider dans son installation et vous expliquer comment une application Java communique avec elle.

Là encore, nous allons survoler un sujet extrêmement vaste et il faudrait un cours complet pour traiter proprement le sujet. Et puisque le hasard fait très bien les choses sur le Site du Zéro, [Taguan](#) a justement rédigé un excellent tutoriel de prise en main de la technologie que nous allons utiliser, que je vous encourage bien évidemment à lire avant de suivre ce chapitre : [Administrez vos bases de données avec MySQL](#).

Présentation des bases de données



Une base de données... pourquoi ?

Vous connaissez déjà plusieurs moyens de stocker des informations depuis votre application :

- **dans des objets** et leurs attributs, mais ceux-ci ne restent en mémoire que de manière temporaire, cette durée étant déterminée par leur portée. Au final, lorsque le serveur d'applications est arrêté, toutes les données sont perdues ;
- **dans des fichiers**, dans lesquels vous savez écrire en manipulant les flux d'entrée et sortie. Les données ainsi écrites sur le disque ont le mérite d'être sauvegardées de manière permanente, et sont accessibles peu importe que l'application soit en ligne ou non. Le souci et vous le savez, c'est que cela devient vite très compliqué dès que vous avez beaucoup de données à enregistrer et à gérer.

Dans une application web, vous ne pouvez pas y couper, vous devez gérer une grande quantité de données : pour un site comme le Site du Zéro, il faut par exemple enregistrer et gérer les informations concernant les membres, les articles et tutoriels écrits dans les sections news et cours, les sujets et réponses écrits dans le forum, les offres d'emploi, les livres en vente, etc. Toutes ces données sont sans arrêt lues, écrites, modifiées ou supprimées, et ce serait mission impossible sans un système de stockage efficace.

Ce système miracle, c'est la **base de données** : elle permet d'enregistrer des données de façon organisée et hiérarchisée.

Structure

La base de données (BDD, ou *DB* en anglais) est un système qui enregistre des informations, mais pas n'importe comment : ces informations sont toujours classées. Et c'est ça qui fait que la BDD est si pratique : c'est un moyen extrêmement simple de ranger des informations ! Grossièrement, une BDD peut être vue comme un ensemble de tableaux, des structures contenant donc des lignes et des colonnes et dans lesquelles nos données sont rangées.

Il existe un vocabulaire spécifique pour désigner les différents éléments composant une BDD :

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

En résumé, une **base** peut contenir plusieurs **tables**, qui peuvent contenir plusieurs **entrées**, pouvant à leur tour contenir plusieurs **champs**.

Voici par exemple ce à quoi pourrait ressembler une table regroupant des informations concernant les membres d'un site :

id	pseudo	email	age
1	Coyote	coyote@bipbip.com	25

2	Thunderseb	jadorejquery@unefois.be	24
3	Kokotchy	decapsuleur@biere.org	27
4	Marcel	marcel@laposte.net	47
...

Vous voyez bien ici qu'une table se représente parfaitement par un simple tableau. Dans cet exemple :

- les **champs** sont les têtes de colonne, à savoir "id", "pseudo", "email" et "age" ;
- chaque ligne du tableau est une **entrée** de la **table** ;
- il n'y a que quatre entrées, mais une table peut très bien en contenir des millions !



À quoi sert le champ "id" ?

Il signifie identifiant, et permet de numérotter les entrées d'une table : mettre en place un tel champ n'est pas une obligation, mais si vous avez lu le chapitre sur les clés primaires du cours de MySQL que je vous ai conseillé en introduction, vous savez déjà que cette pratique nous sera très utile lors de la conception de nos tables.



Où sont stockées les données ?

En effet, c'est bien gentil de tout planquer dans la grosse boîte "base de données", mais au final où sont enregistrés les tableaux et les données qu'ils contiennent ? Eh bien il n'y a rien de magique, tout cela est sauvegardé dans... des fichiers écrits sur le disque ! Seulement, ce ne sont pas de simples fichiers texte, ils ne sont en aucun cas destinés à être édités à la main par le développeur : leur format est bien particulier et dépend du système de gestion utilisé. La représentation en tableaux utilisée précédemment pour vous faire comprendre comment fonctionne une table ne doit pas vous induire en erreur : sous la couverture, les données sont ordonnées de manière bien plus complexe !

SGBD

Dans le dernier paragraphe, je vous ai parlé de système de gestion, qu'on raccourcit en SGBD. Vous devez savoir qu'il n'existe pas qu'une seule solution pour créer et gérer des bases de données. Voici une liste des principaux acteurs de ce marché :

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



Comment choisir un SGBD ?

Comme pour tout choix de technologie, la décision peut être influencée par plusieurs contraintes : coût, performances, support, etc. Retenez simplement que dans la très grande majorité des cas, MySQL ou PostgreSQL répondront parfaitement et gratuitement à vos besoins !



Comment fonctionne un SGBD ?

Chaque système utilise un ensemble d'algorithmes pour trier et stocker les informations et pour y accéder, via des requêtes écrites en langage SQL. Un tel ensemble porte un nom bien particulier : le **moteur de stockage**. Il y a beaucoup à dire sur ce sujet

: plutôt que de paraphraser, je vais vous demander de lire ce cours écrit par un autre membre du Site du Zéro, expliquant en détail quelles sont les solutions existantes pour le système MySQL, pourquoi elles existent et en quoi elles diffèrent : [les moteurs de stockages de MySQL](#).



Dans notre projet, nous allons utiliser le moteur **InnoDB**, un moteur **relationnel** : il s'assure que les relations mises en place entre les données de plusieurs tables sont cohérentes et que si l'on modifie certaines données, ces changements seront répercutés aux tables liées.

SQL

Je vous ai expliqué que les tables et les données contenues dans une BDD étaient enregistrées dans des fichiers, mais que nous ne pouvions pas éditer ces fichiers à la main. Dans la pratique, nous n'irons en effet jamais toucher à ces fichiers directement : nous allons toujours déléguer cette tâche à MySQL. Dans les coulisses, c'est lui qui se débrouillera pour classer nos informations. Et c'est bien là tout l'avantage des bases de données : nous n'aurons jamais à nous soucier de la manière dont seront organisées nos données, nous nous contenterons de donner des ordres au SGBD.



Comment communiquer avec notre SGBD ?

Pour dialoguer avec lui, nous devons lui envoyer des requêtes écrites en langage SQL. Encore un nouveau langage à apprendre... La bonne nouvelle, c'est que le SQL est un standard, c'est-à-dire que peu importe le SGBD utilisé, vous devrez toujours lui parler en SQL. Le hic, c'est que d'un SGBD à l'autre, on observe quelques variantes dans la syntaxe des requêtes : assurez-vous toutefois, cela ne concerne généralement que certaines commandes qui sont peu utilisées.

Le langage SQL n'est absolument pas lié au langage Java : c'est un langage à part entière, uniquement destiné aux bases de données.

Pour vous donner une première idée de la syntaxe employée, voici un exemple de requête SQL :

Code : SQL - Exemple

```
SELECT id, nom, prenom, email FROM utilisateurs ORDER BY id;
```



Nous sommes ici pour apprendre le Java EE, et malheureusement pour vous si vous ne connaissez pas encore les bases du langage SQL, je ne peux pas me permettre de disserter sur le sujet.

Heureusement, si vous souhaitez en savoir plus, comme je vous l'ai déjà dit en introduction il existe un cours de MySQL complet sur le Site du Zéro, qui vous guidera pas à pas dans votre apprentissage du langage. Lorsque vous vous sentirez à l'aise avec le sujet, vous serez alors capables de donner n'importe quel ordre à votre base de données, et serez donc capables de comprendre intégralement les exemples de ce cours sans aucune difficulté ! 😊

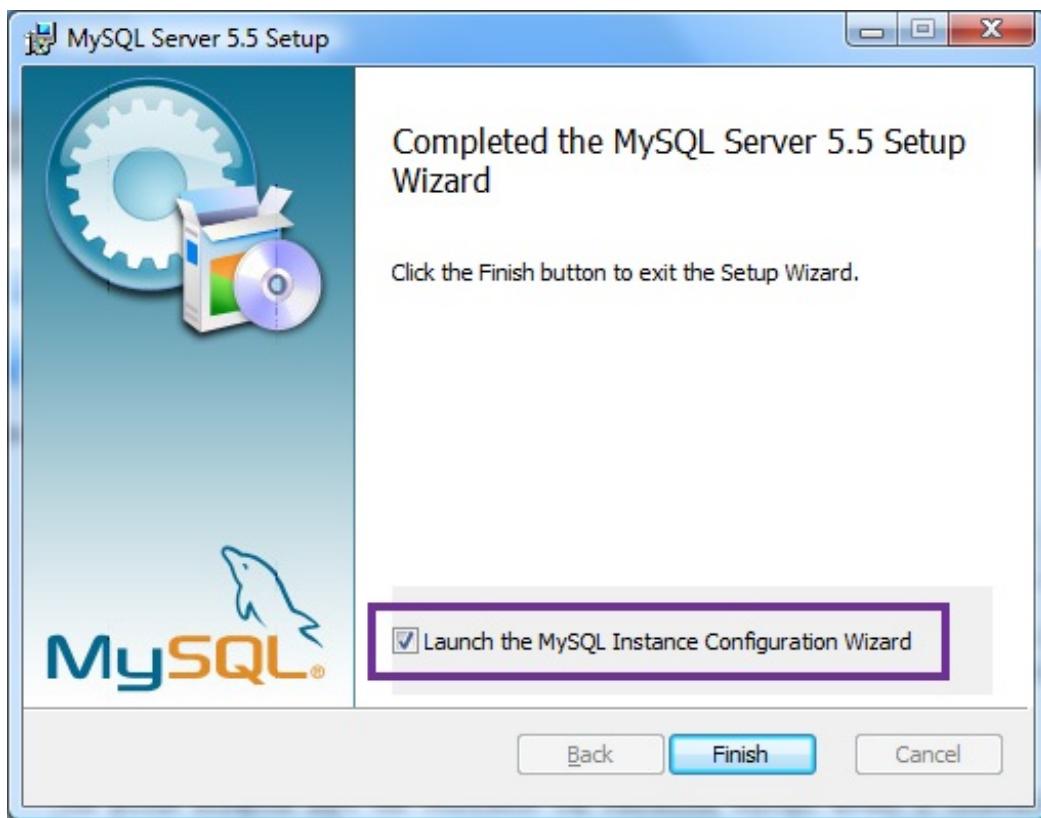
Préparation de la base avec MySQL

Installation

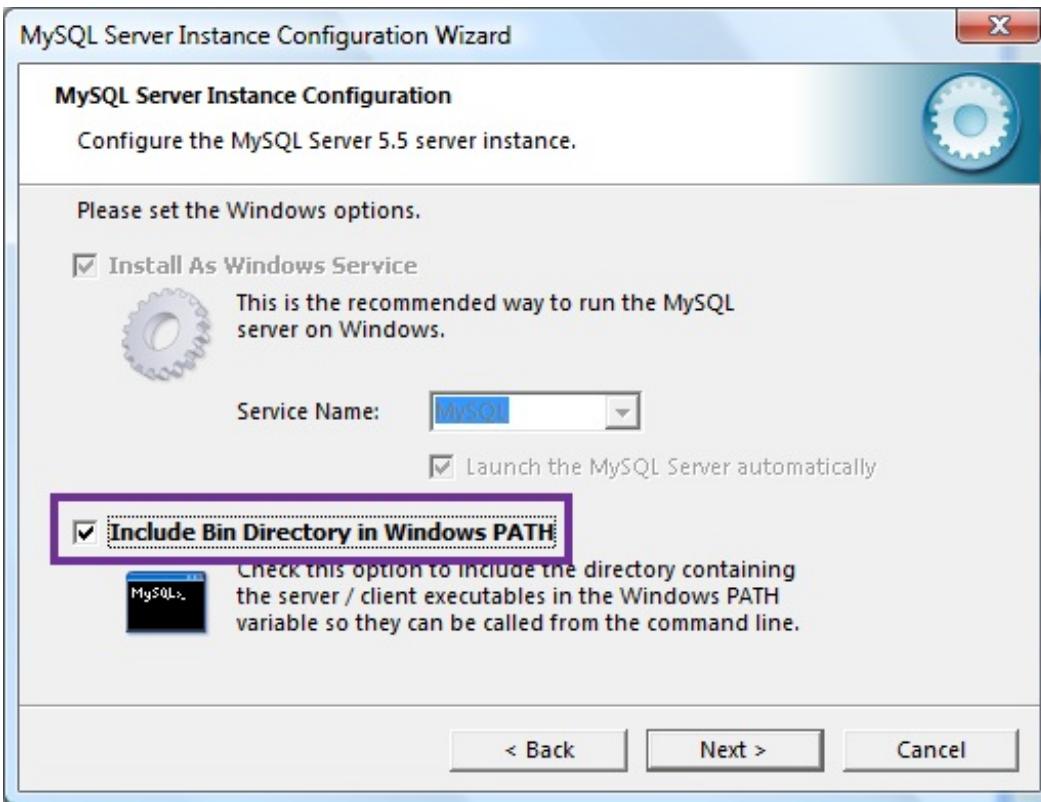
Pour commencer, vous devez vous rendre sur [la page de téléchargement](#) du site de MySQL. Sélectionnez alors le système d'exploitation sur lequel vous travaillez (Windows, Mac OS ou Linux, 32 bits ou 64 bits), et téléchargez la version de MySQL correspondante.

Sous Windows

Le plus simple est de choisir la version MySQL avec l'installateur MSI, et d'exécuter le fichier une fois téléchargé. L'assistant démarra alors et vous guidera lors de l'installation. Lorsqu'il vous demandera de choisir entre trois types d'installation, vous choisirez "Typical". Arrivés à la fin de l'installation, une fenêtre vous demandera si vous souhaitez lancer l'assistant de configuration MySQL (voir la figure suivante).



Vous veillerez bien à cliquer sur la case à cocher entourée ci-dessus avant de cliquer sur le bouton Finish. Un assistant vous demandera alors de préciser quelles options vous souhaitez activer. Choisissez la configuration standard, et à l'étape suivante, cochez l'option "Include Bin Directory in Windows PATH", comme c'est indiqué sur la figure suivante.



L'outil vous proposera alors de définir un nouveau mot de passe pour l'utilisateur "root". Ne cochez aucune autre option à cette étape, et cliquez sur Execute pour lancer la configuration.

Sous Linux

Il vous suffit d'ouvrir un terminal et d'exécuter la commande suivante pour installer MySQL, ou son équivalent sous les

distributions non *debian-like* :

Code : Console

```
sudo apt-get install mysql-server mysql-client
```

Une fois l'installation terminée, vous pourrez modifier le mot de passe par défaut de l'utilisateur "root" avec la commande suivante :

Code : Console

```
sudo mysqladmin -u root -h localhost password 'entrez ici votre mot de passe'
```

Sous Mac OS

Le plus simple pour ce système est de télécharger depuis le site de MySQL l'archive au format DMG. Ouvrez-la et vous y trouverez un fichier PKG dont le nom doit ressembler à **mysql-5.5.27-osx10.6-x86.pkg**, certains chiffres et extensions pouvant varier ou s'ajouter selon la version courante de MySQL lorsque vous procédez au téléchargement, et selon la configuration de votre poste. Il s'agit là de l'assistant d'installation de MySQL : exécutez-le, et suivez simplement les instructions qui vous sont données.

Une fois l'installation terminée, deux possibilités s'offrent à vous concernant la configuration du serveur : soit vous installez le package *MySQL Startup Item*, soit vous effectuez les manipulations à la main. Vous trouverez de plus amples détails concernant ces deux options sur [cette page du site officiel](#) dédiée à l'installation sous Mac OS X.

Création d'une base

Une fois le serveur MySQL installé sur votre poste, vous pouvez créer la base de données qui va nous servir dans toute la suite du cours. Nous allons en l'occurrence créer une base nommée **bdd_sdzee**, et initialiser l'encodage par défaut à UTF-8. Souvenez-vous, il s'agit là de l'encodage qui permettra à votre application de manipuler la plus grande variété de caractères :

Code : SQL - Requête de création de la base

```
CREATE DATABASE bdd_sdzee DEFAULT CHARACTER SET utf8 COLLATE utf8_general_ci;
```

Création d'un utilisateur

Par défaut, MySQL propose un compte *root* qui donne accès à l'intégralité du serveur. C'est une très mauvaise pratique de travailler via ce compte, nous allons donc créer un utilisateur spécifique à notre application, qui n'aura accès qu'à la base sur laquelle nous travaillons :

Code : SQL - Requête de création de l'utilisateur

```
CREATE USER 'java'@'localhost' IDENTIFIED BY 'SdZ_eE';
GRANT ALL ON bdd_sdzee.* TO 'java'@'localhost' IDENTIFIED BY 'SdZ_eE';
```

Ici le compte créé se nomme **java**, a pour mot de passe **SdZ_eE** et dispose de tous les droits sur la base **bdd_sdzee**. Vous pouvez bien entendu changer le mot de passe si celui-là ne vous plaît pas. 😊

Pour l'utiliser, il faut ensuite vous déconnecter via la commande `exit`, et vous connecter à nouveau via la commande

```
mysql -h localhost -u java -p, et enfin entrer la commande use bdd_sdzee.
```

Création d'une table

Maintenant que la base est prête et que nous avons créé et utilisons un compte, nous pouvons mettre en place une table qui va nous servir dans les exemples du chapitre à venir. Nous allons créer une table qui représente des utilisateurs, comportant un identifiant, une adresse mail, un mot de passe, un nom et une date d'inscription :

Code : SQL - Requête de création de la table

```
CREATE TABLE bdd_sdzee.Utilisateur (
    id INT( 11 ) NOT NULL AUTO_INCREMENT ,
    email VARCHAR( 60 ) NOT NULL ,
    mot_de_passe VARCHAR( 32 ) NOT NULL ,
    nom VARCHAR( 20 ) NOT NULL ,
    date_inscription DATETIME NOT NULL ,
    PRIMARY KEY ( id ),
    UNIQUE ( email )
) ENGINE = INNODB;
```

Vous pouvez ensuite vérifier la bonne création de la table nommée **Utilisateur** avec les commandes `SHOW tables;` et `DESCRIBE Utilisateur;`.

Insertion de données d'exemple

Afin de pouvoir commencer en douceur la manipulation d'une base de données depuis notre application dans le chapitre suivant, nous allons ici directement mettre en place quelques données factices, qui nous serviront d'exemple par la suite :

Code : SQL - Requêtes d'insertion de données

```
INSERT INTO Utilisateur (email, mot_de_passe, nom, date_inscription)
VALUES ('coyote@mail.acme', MD5('bipbip'), 'Coyote', NOW());
INSERT INTO Utilisateur (email, mot_de_passe, nom, date_inscription)
VALUES ('jadorejquery@unefois.be', MD5('avecdesfrites'),
'Thunderseb', NOW());
```

Vous pouvez ensuite vérifier la bonne insertion des données dans la table via la commande `SELECT * FROM Utilisateur;` (voir la figure suivante).

id	email	mot_de_passe	nom	date_inscription
1	coyote@mail.acme	7629ae53dbd1743ebbec21bc6bccbf45	Coyote	2012-08-11 21:58:52
2	jadorejquery@unefois.be	0094bcab75591116f732cc180949c16d	Thunderseb	2012-08-11 21:59:13

2 rows in set (0.00 sec)

Mise en place de JDBC dans le projet

Nous sommes capables de créer des bases et des tables dans MySQL, mais notre mission maintenant c'est bien entendu d'effectuer la liaison entre MySQL et notre projet Java EE. Car ce que nous souhaitons, c'est pouvoir interagir avec les tables de notre base **bdd_sdzee** directement depuis le code de notre application !

JDBC

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

- la connexion avec le SGBD ;
- l'envoi de requêtes SQL au SGBD depuis une application Java ;
- le traitement des données et éventuelles erreurs renvoyées par le SGBD lors des différentes étapes du dialogue

(connexion, requête, exécution, etc.).

Seulement, comme vous le savez, il existe plusieurs SGBD différents et bien qu'ils se basent tous sur le langage SQL, chacun a sa manière de gérer les données. L'avantage de JDBC, c'est qu'il est nativement prévu pour pouvoir s'adapter à n'importe quel SGBD ! Ainsi pour faire en sorte que notre application puisse dialoguer avec MySQL, nous aurons simplement besoin d'ajouter à notre projet **un driver** qui est spécifique à MySQL.

Si nous utilisions PostgreSQL, il nous faudrait, à la place, ajouter **un driver** propre à PostgreSQL, etc. Dans la pratique c'est très simple, il s'agit tout bonnement d'une archive .jar que nous allons ajouter au **build-path** de notre projet !

Pour information, sachez que JDBC ne fait pas uniquement partie de la plate-forme Java EE, c'est une brique de base de la plate-forme standard Java SE. Cependant, bien que ce cours se consacre à l'apprentissage du Java EE, j'ai jugé qu'il était préférable de s'y attarder, afin que ceux d'entre vous qui ne sont pas habitués à créer des applications Java faisant intervenir une base de données puissent découvrir et appliquer sereinement le principe par la suite. Après tout, cette brique rentre parfaitement dans le cadre de ce que nous étudions dans ce cours, à savoir la création d'une application web !

Mise en place

Pour commencer, il faut récupérer le driver sur [la page de téléchargement](#) du site de MySQL (voir la figure suivante).

The screenshot shows the MySQL Connector/J 5.1.19 download page. At the top, there's a link to 'Generally Available (GA) Releases'. Below it, the title 'Connector/J 5.1.19' is displayed. A 'Select Platform:' dropdown menu is set to 'Platform Independent' with a 'Select' button next to it. To the right, a link says 'Looking for previous GA versions?'. Two download options are listed:

Format	Version	Size	Action
Platform Independent (Architecture Independent), Compressed TAR Archive (mysql-connector-java-5.1.19.tar.gz)	5.1.19	3.7M	Download
Platform Independent (Architecture Independent), ZIP Archive (mysql-connector-java-5.1.19.zip)	5.1.19	3.9M	Download

Téléchargement du driver MySQL

Téléchargez alors l'archive au format que vous préférez. Décompressez son contenu dans un dossier sur votre poste, il contiendra alors un fichier **mysql-connector-java-xxx-bin.jar** (les xxx variant selon la version courante au moment où vous effectuez ce téléchargement).

Il suffit ensuite de copier ce fichier dans le répertoire **/lib** présent dans le dossier d'installation de votre Tomcat, et tous les projets déployés sur ce serveur pourront alors communiquer avec une base de données MySQL.

 Au lieu de déposer le driver directement dans le dossier de bibliothèques du serveur d'applications, vous pouvez également passer par Eclipse et le placer uniquement dans le répertoire **/WEB-INF/lib** d'un projet en particulier. Eclipse va alors automatiquement ajouter l'archive au *classpath* de l'application concernée.

Toutefois, je vous recommande d'utiliser la première méthode. Cela vous permettra de ne pas avoir à inclure le driver dans chacun de vos projets communiquant avec une base de données MySQL.

Création d'un bac à sable

Afin de pouvoir découvrir tranquillement le fonctionnement de JDBC, nous allons mettre en place un espace de tests dans notre application. Comme je vous l'ai dit, JDBC est une brique de la plate-forme Java SE, nous pourrions donc très bien nous créer un nouveau projet Java à part entière et y faire nos tests, avant de revenir à notre projet Java EE. Cependant, je préfère vous faire manipuler directement au sein de notre application, ça vous permettra de pratiquer une nouvelle fois la mise en place des différents composants habituels. Nous allons en l'occurrence avoir besoin :

- d'un objet Java, qui contiendra le code de nos essais de manipulation de la base de données ;
- d'une servlet, qui se contentera d'appeler les exemples présents dans l'objet Java ;
- d'une page JSP, qui affichera enfin les résultats de nos différentes manipulations.

Création de l'objet Java

Pour commencer, mettons en place un nouvel objet Java nommé **TestJDBC**, dans un nouveau package nommé `com.sdzeebdd` :

Code : Java - com.sdzeebdd.TestJDBC

```
package com.sdzeebdd;

import java.util.ArrayList;
import java.util.List;

import javax.servlet.http.HttpServletRequest;

public class TestJDBC {
    /* La liste qui contiendra tous les résultats de nos essais */
    private List<String> messages = new ArrayList<String>();

    public List<String> executerTests( HttpServletRequest request )
    {
        /* Ici, nous placerons le code de nos manipulations */
        /* ... */

        return messages;
    }
}
```

Celui-ci contient une seule méthode, vide pour le moment. C'est dans cette méthode que nous allons, par la suite, placer le code de nos manipulations, et c'est cette méthode qui sera appelée par notre servlet. La seule chose qu'elle retourne est une liste de messages, qui seront récupérés par la servlet.

Création de la servlet

Nous avons besoin d'une servlet pour appeler les tests écrits dans notre objet Java. Cette servlet sera déclenchée par un accès depuis votre navigateur à l'URL <http://localhost:8080/pro/testjdbc>.

Voici donc sa déclaration dans le fichier web.xml de notre application :

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

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

<servlet-mapping>
    <servlet-name>GestionTestJDBC</servlet-name>
    <url-pattern>/testjdbc</url-pattern>
</servlet-mapping>
```

Et voici son code :

Code : Java - com.sdzee.servlets.GestionTestJDBC

```
package com.sdzee.servlets;

import java.io.IOException;
import java.util.List;

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

import com.sdzee.bdd.TestJDBC;

public class GestionTestJDBC extends HttpServlet {
    public static final String ATT_MESSAGES = "messages";
    public static final String VUE = "/WEB-INF/test_jdbc.jsp";

    public void doGet( HttpServletRequest request,
HttpServletResponse response ) throws ServletException, IOException
{
    /* Initialisation de l'objet Java et récupération des
messages */
    TestJDBC test = new TestJDBC();
    List<String> messages = test.executerTests( request );

    /* Enregistrement de la liste des messages dans l'objet
requête */
    request.setAttribute( ATT_MESSAGES, messages );

    /* Transmission vers la page en charge de l'affichage des
résultats */
    this.getServletContext().getRequestDispatcher( VUE
).forward( request, response );
}
}
```

Elle se contentera d'appeler la méthode de notre nouvel objet Java à chaque requête GET reçue, autrement dit, chaque fois que nous accéderons depuis notre navigateur à l'URL que nous avons définie précédemment. Elle récupérera lors de cet appel la liste des messages créés par le code de nos essais, et la transmettra alors à une page JSP pour affichage.

Création de la page JSP

Enfin, nous avons besoin d'une page qui aura pour unique mission d'afficher les messages retournés par nos différents tests, que nous allons nommer **test_jdbc.jsp** et que nous allons placer sous **/WEB-INF** :

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

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Tests JDBC</title>
        <link type="text/css" rel="stylesheet" href="" />
    </head>
    <body>
        <h1>Tests JDBC</h1>

        <c:forEach items="${ messages }" var="message"
varStatus="boucle">
```

```
<p>${ boucle.count }. ${ message }</p>
</c:forEach>
</body>
</html>
```

- Une base de données permet de stocker des données de manière organisée et hiérarchisée.
- L'utilisation de l'encodage UTF-8 pour stocker les données d'une application web est très recommandée.
- Le driver JDBC permet l'interaction entre une application Java (web ou non) et une base de données.
- Il existe un driver différent pour chaque type de base de données existante.
- Il s'agit concrètement d'un simple fichier Jar, à placer dans le répertoire /lib du serveur d'applications.

Communiquez avec votre BDD

Nous allons, pour commencer, découvrir les nouveaux objets et interfaces qui doivent intervenir dans notre application afin d'établir une communication avec la base de données. Nous apprendrons ensuite à lire des données depuis la base vers notre application, et enfin à écrire des données depuis notre application vers la base.

Une fois la théorie assimilée, nous mettrons tout cela en musique dans un exemple pratique, avant d'analyser les manques et risques liés à la technique employée, et de découvrir une seconde manière de procéder qui pallie ces lacunes.

Chargement du driver

Nous avons, dans le chapitre précédent, récupéré le driver JDBC correspondant à MySQL, et nous l'avons ajouté au *classpath* du projet. Il nous est maintenant nécessaire de procéder à ce que l'on nomme **le chargement du driver** depuis le code de notre application. Voici le code minimal nécessaire :

Code : Java - Chargement du driver JDBC MySQL

```
/* Chargement du driver JDBC pour MySQL */
try {
    Class.forName( "com.mysql.jdbc.Driver" );
} catch ( ClassNotFoundException e ) {
    /* Gérer les éventuelles erreurs ici. */
}
```

Intéressons-nous à la ligne 3 de notre exemple, dont le rôle est justement de charger le driver. Le nom de ce dernier, en l'occurrence `"com.mysql.jdbc.Driver"`, est fourni par son constructeur, autrement dit ici par MySQL. Si vous utilisez un autre SGBD, vous devrez vous renseigner sur le site de son distributeur pour trouver son nom exact et ainsi pouvoir le charger depuis votre application.

Si cette ligne de code envoie une exception de type `ClassNotFoundException`, cela signifie que le fichier .jar contenant le driver JDBC pour MySQL n'a pas été correctement placé dans le *classpath*. Vous pourrez d'ailleurs faire le test vous-mêmes lorsque nous passerons à la pratique, en retirant le driver que nous avons ajouté en tant que bibliothèque externe, et constater que cette ligne envoie bien une exception ! 😊



Notez bien que dans la pratique, **il est absolument inutile de charger le driver avant chaque connexion**, une seule fois durant le chargement de l'application suffit ! Toutefois pour des raisons de simplicité, dans les exemples de ce chapitre nous ne nous préoccupons pas de ce détail et chargerons le driver à chaque exécution de nos tests.

Nous allons maintenant apprendre à communiquer avec une base de données. Pour ce faire, nous devons suivre le processus suivant :

1. nous connecter à la base ;
2. créer et exécuter une requête SQL ;
3. analyser son résultat ;
4. fermer les différentes ressources mises en jeu.

Sans plus attendre, découvrons tout cela en détail !

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

Le contenu qui va suivre est assez dense, mais il faut bien passer par la théorie pour comprendre. Pas de panique si vous n'assimilez pas tout ce que je vous présente ici, la partie suivante de ce chapitre est entièrement consacrée à la pratique ! Nous allons commencer par étudier la syntaxe et les objets à mettre en œuvre, puis nous appliquerons ensuite tout cela à travers un petit scénario de tests.

Connexion à la base de données

Identification de l'URL

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

Code : URL

```
jdbc:mysql://nomhote:port/nombdd
```

Dans cette adresse, vous remarquez plusieurs sections :

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

Ainsi, puisque notre serveur MySQL est installé sur le même poste que notre serveur d'applications, l'URL finale sera dans notre cas :

Code : URL

```
jdbc:mysql://localhost:3306/bdd_sdzee
```

Étudions maintenant la mise en place de la connexion entre notre application et notre base.

Établissement de la connexion

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

Code : Java - Connexion à la base de données

```
/* Connexion à la base de données */
String url = "jdbc:mysql://localhost:3306/bdd_sdzee";
String utilisateur = "java";
String motDePasse = "$dZ_fE";
Connection connexion = null;
try {
    connexion = DriverManager.getConnection( url, utilisateur,
motDePasse );

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

} catch ( SQLException e ) {
    /* Gérer les éventuelles erreurs ici */
} finally {
    if ( connexion != null )
        try {
            /* Fermeture de la connexion */
            connexion.close();
        } catch ( SQLException ignore ) {
            /* Si une erreur survient lors de la fermeture, il
suffit de l'ignorer. */
        }
}
```

L'établissement d'une connexion s'effectue à travers l'objet **DriverManager**. Il suffit d'appeler sa méthode statique **getConnection()** pour récupérer un objet de type **Connection**. Comme vous pouvez le voir ici, celle-ci prend en argument l'adresse de la base de données, le nom d'utilisateur et le mot de passe associé.

L'appel à cette méthode peut retourner des erreurs de type `SQLException` :

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

Si un de ces derniers cas survient, vous trouverez ci-dessous une liste des causes possibles et les pistes de résolution associées :

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

Enfin, peu importe que la connexion ait réussi ou non, retenez bien que sa fermeture dans un bloc `finally` est extrêmement importante !



Si vous ne fermez pas les connexions que vous ouvrez, et en gardez un grand nombre ouvertes sur une courte période, le serveur risque d'être saturé et de ne plus accepter aucune nouvelle connexion ; votre application risque alors de planter.



La bonne pratique à suivre est de toujours ouvrir et fermer la connexion le plus finement possible, c'est-à-dire au plus près de l'exécution de vos requêtes, dans un ensemble `try-catch-finally`. Cela permet de s'assurer qu'aucune connexion n'est ouverte ni ne reste ouverte inutilement. La mise en place d'un bloc `finally` permet de s'assurer que la connexion sera fermée quoi qu'il arrive en cours de route.

Nous n'allons pas entrer dans les détails pour le moment, puisque nous n'en sommes qu'au début de notre apprentissage, mais sachez d'ores et déjà que l'ouverture d'une connexion a un coût non négligeable en termes de performances. Dans une application très fréquentée, il devient hors de question de procéder à des ouvertures/fermetures à chaque requête effectuée, cela reviendrait à signer l'arrêt de mort de votre serveur ! Nous y reviendrons en temps voulu, pas d'inquiétudes. 😊

Création d'une requête

Avant de pouvoir créer des instructions SQL, vous devez tout d'abord créer un objet de type `Statement`. Si vous parcourrez sa documentation, vous constaterez qu'il s'agit en réalité d'une interface dont le rôle est de permettre l'exécution de requêtes. Pour initialiser cet objet, rien de plus simple, il suffit d'appeler la méthode `createStatement()` de l'objet `Connection` précédemment obtenu ! Donc si nous reprenons notre exemple, juste après l'établissement de la connexion dans notre bloc `try`, nous devons ajouter le code suivant :

Code : Java - Initialisation de l'objet Statement

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

```
Statement statement = connexion.createStatement();
```

Exécution de la requête

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

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

En outre, il existe des variantes de chaque méthode prenant en compte d'autres arguments, ainsi que deux autres méthodes nommées `execute()` et `executeBatch()`. Nous n'allons pas nous attarder sur les subtilités mises en jeu, je vous laisse le soin de lire la documentation de l'objet `Statement` si vous souhaitez en savoir davantage.

Exécution d'une requête de lecture

En parcourant la documentation de la méthode `executeQuery()`, nous apprenons qu'elle retourne un objet de type `ResultSet` contenant le résultat de la requête. Voici donc un exemple effectuant un **SELECT** sur notre table d'utilisateurs :

Code : Java - Exécution d'une requête de type SELECT

```
/* Exécution d'une requête de lecture */
ResultSet resultat = statement.executeQuery( "SELECT id, email,
mot_de_passe, nom FROM Utilisateur" );
```

Avant d'apprendre à récupérer et analyser le résultat retourné par cet appel, regardons comment effectuer une requête d'écriture dans la base.

Exécution d'une requête d'écriture

En parcourant cette fois la documentation de la méthode `executeUpdate()`, nous apprenons qu'elle retourne un entier représentant le nombre de lignes affectées par la requête réalisée. Si par exemple vous réalisez une insertion de données via un **INSERT**, cette méthode retournera 0 en cas d'échec et 1 en cas de succès. Si vous réalisez une mise à jour via un **UPDATE**, cette méthode retournera le nombre de lignes mises à jour. Idem en cas d'une suppression via un **DELETE**, etc.

Je vous propose ici un exemple effectuant un **INSERT** sur notre table d'utilisateurs :

Code : Java - Exécution d'une requête de type INSERT

```
/* Exécution d'une requête d'écriture */
int statut = statement.executeUpdate( "INSERT INTO Utilisateur
(email, mot_de_passe, nom, date_inscription) VALUES
('jmarc@mail.fr', MD5('lavieestbelle78')), 'jean-marc', NOW());" );
```

Avant de pouvoir tester ces deux méthodes, il nous faut encore découvrir comment manipuler les résultats retournés.

Accès aux résultats de la requête

Retour d'une requête de lecture

Je vous l'ai déjà soufflé, l'exécution d'une requête de lecture via la méthode `statement.executeQuery()` retourne un

objet de type `ResultSet`. Vous pouvez le voir comme un tableau, qui contient les éventuelles données retournées par la base de données sous forme de lignes. Pour accéder à ces lignes de données, vous avez à votre disposition un curseur, que vous pouvez déplacer de ligne en ligne. Notez bien qu'il ne s'agit pas d'un [curseur au sens base de données du terme](#), mais bien d'un curseur propre à l'objet `ResultSet`. Voyons cela dans un exemple, puis commentons :

Code : Java - Exécution d'une requête de type SELECT et récupération du résultat

```
/* Exécution d'une requête de lecture */
ResultSet resultat = statement.executeQuery( "SELECT id, email,
mot_de_passe, nom FROM Utilisateur" );

/* Récupération des données du résultat de la requête de lecture */
while ( resultat.next() ) {
    int idUtilisateur = resultat.getInt( "id" );
    String emailUtilisateur = resultat.getString( "email" );
    String motDePasseUtilisateur = resultat.getString(
"mot_de_passe" );
    String nomUtilisateur = resultat.getString( "nom" );

    /* Traiter ici les valeurs récupérées. */
}
```

Pour commencer, à la ligne 2 nous récupérons le retour de l'appel à la méthode `statement.executeQuery()` dans un objet `ResultSet`.

Ensuite, afin de pouvoir accéder aux lignes contenues dans cet objet, nous effectuons un appel à la méthode `next()`, qui permet de déplacer le curseur à la ligne suivante. Elle retourne un booléen, initialisé à `true` tant qu'il reste des données à parcourir.



Pourquoi ne lisons-nous pas la première ligne avant de déplacer le curseur à la ligne suivante ?

Tout simplement parce que lors de la création d'un objet `ResultSet`, son curseur est par défaut positionné **avant** la première ligne de données. Ainsi, il est nécessaire de se déplacer d'un cran vers l'avant pour pouvoir commencer à lire les données contenues dans l'objet.



Si vous essayez de lire des données avant d'avoir déplacé le curseur, votre code enverra une `SQLException`.

Une fois le curseur positionné correctement, il ne nous reste plus qu'à récupérer les contenus des différents champs via une des nombreuses méthodes de récupération proposées par l'objet `ResultSet`. Je ne vais pas vous en faire la liste exhaustive, je vous laisse parcourir la documentation pour les découvrir en intégralité. Sachez simplement qu'il en existe une par type de données récupérables :

- une méthode `resultat.getInt()` pour récupérer un entier ;
- une méthode `resultat.getString()` pour récupérer une chaîne de caractères ;
- une méthode `resultat.getBoolean()` pour récupérer un booléen ;
- etc.

Chacune de ces méthodes existe sous deux formes différentes :

- soit elle prend en argument le nom du champ visé dans la table de la base de données ;
- soit elle prend en argument l'index du champ visé dans la table de la base de données.

En l'occurrence, nous avons ici utilisé le nom des champs : `"id"`, `"email"`, `"mot_de_passe"` et `"nom"`. Si nous avions voulu utiliser leurs index, alors nous aurions dû remplacer les lignes 6 à 9 du code précédent par :

Code : Java - Ciblage des champs par leurs index

```
int idUtilisateur = resultat.getInt( 1 );
String emailUtilisateur = resultat.getString( 2 );
String motDePasseUtilisateur = resultat.getString( 3 );
```

```
String nomUtilisateur = resultat.getString( 4 );
```

En effet, la colonne nommée "id" est bien la colonne n°1 de notre table, "email" la colonne n°2, etc.



Peu importe que vous utilisez l'index ou le nom des champs, vous devez vous assurer qu'ils existent bel et bien dans la table. Si vous précisez un nom de champ qui n'existe pas, ou un index de champ qui dépasse l'index de la dernière colonne, alors la méthode de récupération vous enverra une `SQLException`.

Dernière information importante : afin de parcourir toutes les lignes de données renvoyées dans le résultat de la requête, il est nécessaire de boucler sur l'appel à la méthode de déplacement du curseur. Lorsque ce curseur atteindra la dernière ligne contenue dans l'objet, la méthode `resultat.next()` renverra `false`, et nous sortirons ainsi automatiquement de notre boucle une fois le parcours des données achevé.

Retour d'une requête d'écriture

Lorsque vous effectuez une modification sur une table de votre base de données via la méthode `statement.executeUpdate()`, celle-ci renvoie des informations différentes selon le type de la requête effectuée :

- l'exécution d'un `INSERT` renvoie 0 en cas d'échec de la requête d'insertion, et 1 en cas de succès ;
- l'exécution d'un `UPDATE` ou d'un `DELETE` renvoie le nombre de lignes respectivement mises à jour ou supprimées ;
- l'exécution d'un `CREATE`, ou de toute autre requête ne retournant rien, renvoie 0.

Voici par exemple comment récupérer le statut d'une requête d'insertion de données :

Code : Java - Exécution d'une requête de type INSERT et récupération du statut

```
/* Exécution d'une requête d'écriture */
int statut = statement.executeUpdate( "INSERT INTO Utilisateur
(email, mot_de_passe, nom, date_inscription) VALUES
('jmarc@mail.fr', MD5('lavieestbelle78'), 'jean-marc', NOW());" );
```

Libération des ressources

Tout comme il est nécessaire de fermer proprement une connexion ouverte, il est extrêmement recommandé de disposer proprement des objets `Statement` et `ResultSet` initialisés au sein d'une connexion :

Code : Java - Cycle d'ouverture/fermeture des ressources impliquées dans une communication avec la BDD

```
Connection connexion = null;
Statement statement = null;
ResultSet resultat = null;
try {
    /*
     * Ouverture de la connexion, initialisation d'un Statement,
     * initialisation d'un ResultSet, etc.
    */
} catch ( SQLException e ) {
    /* Traiter les erreurs éventuelles ici. */
} finally {
    if ( resultat != null ) {
        try {
            /* On commence par fermer le ResultSet */
            resultat.close();
        } catch ( SQLException ignore ) {
        }
    }
}
if ( statement != null ) {
```

```

        try {
            /* Puis on ferme le Statement */
            statement.close();
        } catch ( SQLException ignore ) {
        }
    }
    if ( connexion != null ) {
        try {
            /* Et enfin on ferme la connexion */
            connexion.close();
        } catch ( SQLException ignore ) {
        }
    }
}

```

Ceux-ci doivent obligatoirement être fermés, du plus récemment ouvert au plus ancien. Ainsi il faut commencer par fermer le ResultSet, puis le Statement et enfin l'objet Connection. Les exceptions éventuelles, envoyées en cas de ressources déjà fermées ou non disponibles, peuvent être ignorées comme c'est le cas dans ce code d'exemple. Vous pouvez bien évidemment choisir de les prendre en compte, par exemple en les enregistrant dans un fichier de logs.



Tout comme pour la connexion, ces ressources doivent être fermées au sein d'un bloc **finally** afin de s'assurer que, quoi qu'il arrive en cours de route, les ressources soient proprement libérées.

Mise en pratique

Nous venons d'étudier les étapes principales de la communication avec une base de données, qui pour rappel sont :

1. la connexion à la base ;
2. la création et l'exécution d'une requête SQL ;
3. la récupération de son résultat ;
4. la fermeture des différentes ressources mises en jeu.

Il est maintenant temps de pratiquer, en utilisant le bac à sable que nous avons mis en place.

Afficher le contenu de la table Utilisateur

En premier lieu, nous allons effectuer une requête de type **SELECT** afin de récupérer ce que contient notre table **Utilisateur**. Voici le code à mettre en place dans la méthode `executerTests()` de notre objet `TestJDBC` :

Code : Java - com.sdzee.bdd.TestJDBC

```

public List<String> executerTests( HttpServletRequest request ) {
    /* Chargement du driver JDBC pour MySQL */
    try {
        messages.add( "Chargement du driver..." );
        Class.forName( "com.mysql.jdbc.Driver" );
        messages.add( "Driver chargé !" );
    } catch ( ClassNotFoundException e ) {
        messages.add( "Erreur lors du chargement : le driver n'a pas
été trouvé dans le classpath ! <br/>" +
                    + e.getMessage() );
    }

    /* Connexion à la base de données */
    String url = "jdbc:mysql://localhost:3306/bdd_sdzee";
    String utilisateur = "java";
    String motDePasse = "$dZ_fE";
    Connection connexion = null;
    Statement statement = null;
    ResultSet resultat = null;
    try {
        messages.add( "Connexion à la base de données..." );
        connexion = DriverManager.getConnection( url, utilisateur,
motDePasse );
        messages.add( "Connexion réussie !" );
    }
}

```

```

    /* Crédation de l'objet gérant les requêtes */
    statement = connexion.createStatement();
    messages.add( "Objet requête créé !" );

    /* Exécution d'une requête de lecture */
    resultat = statement.executeQuery( "SELECT id, email,
mot_de_passe, nom FROM Utilisateur;" );
    messages.add( "Requête \"SELECT id, email, mot_de_passe, nom
FROM Utilisateur;\" effectuée !" );

    /* Récupération des données du résultat de la requête de
lecture */
    while ( resultat.next() ) {
        int idUtilisateur = resultat.getInt( "id" );
        String emailUtilisateur = resultat.getString( "email" );
        String motDePasseUtilisateur = resultat.getString(
"mot_de_passe" );
        String nomUtilisateur = resultat.getString( "nom" );
        /* Formatage des données pour affichage dans la JSP
finale. */
        messages.add( "Données retournées par la requête : id =
" + idUtilisateur + ", email = " + emailUtilisateur
+ ", motdepasse = "
+ motDePasseUtilisateur + ", nom = " +
nomUtilisateur + "." );
    }
} catch ( SQLException e ) {
    messages.add( "Erreur lors de la connexion : <br/>" +
e.getMessage() );
} finally {
    messages.add( "Fermeture de l'objet ResultSet." );
    if ( resultat != null ) {
        try {
            resultat.close();
        } catch ( SQLException ignore ) {
        }
    }
    messages.add( "Fermeture de l'objet Statement." );
    if ( statement != null ) {
        try {
            statement.close();
        } catch ( SQLException ignore ) {
        }
    }
    messages.add( "Fermeture de l'objet Connection." );
    if ( connexion != null ) {
        try {
            connexion.close();
        } catch ( SQLException ignore ) {
        }
    }
}
}

return messages;
}

```

Vous retrouvez ici sans surprise les étapes que nous avons découvertes et analysées précédemment. Les lignes 28 à 42 correspondent à l'exécution de la requête **SELECT** et à la récupération de son résultat.

Vous pouvez remarquer l'utilisation de la liste **messages** que nous avions mise en place lors de la création de notre bac à sable. Nous y insérons ici, et y insérerons dans la suite de nos exemples, tous les messages indiquant les résultats et erreurs de nos tests. À chaque nouvel essai, il nous suffira alors d'accéder à l'URL de la servlet, qui je le rappelle est <http://localhost:8080/pro/testjdbc>, pour que notre JSP finale nous affiche le contenu de cette liste et nous permette ainsi de suivre simplement la progression au sein du code.

Une fois ce code mis en place, il ne vous reste plus qu'à démarrer votre serveur Tomcat si ce n'est pas déjà fait, et à accéder à l'URL de test. Vous pourrez alors visualiser toutes les informations placées dans la liste **messages**, qui est je vous le rappelle

parcourue par notre page JSP. Vous obtiendrez le résultat affiché à la figure suivante si vous avez correctement préparé votre base de données et si vous avez démarré votre serveur SQL.



Tests JDBC

1. Chargement du driver...
2. Driver chargé !
3. Connexion à la base de données...
4. Connexion réussie !
5. Requête "SELECT id, email, mot_de_passe, nom FROM Utilisateur;" effectuée !
6. Données retournées par la requête : id = 1, email = coyote@mail.acme, motdepasse = 7629ae53dbd1743ebbec21bc6bccbf45, nom = Coyote.
7. Données retournées par la requête : id = 2, email = jadorejquery@unefois.be, motdepasse = 0094bcab75591116f732cc180949c16d, nom = Thunderseb.
8. Fermeture de l'objet ResultSet.
9. Fermeture de l'objet PreparedStatement.
10. Fermeture de l'objet Connection.

Tout fonctionne comme prévu, notre application a réussi à :

- charger le driver ;
- se connecter à la base de données avec le compte utilisateur que nous y avions créé ;
- effectuer une requête de sélection et récupérer son résultat ;
- fermer les ressources.

Insérer des données dans la table Utilisateur

Le principe est sensiblement le même que pour une requête de lecture ; il suffit d'utiliser un appel à `statement.executeUpdate()` à la place d'un appel à `statement.executeQuery()`. Dans notre exemple, nous allons insérer des données d'exemple dans la table **Utilisateur**, puis effectuer une requête de lecture pour visualiser le contenu de la table après insertion. Pour ce faire, il faut simplement ajouter le code suivant, **juste avant** les lignes 28 à 42 du code précédent :

Code : Java - com.sdzee.bdd.TestJDBC

```
/* Exécution d'une requête d'écriture */
int statut = statement.executeUpdate( "INSERT INTO Utilisateur
(email, mot_de_passe, nom, date_inscription) VALUES
('jmarc@mail.fr', MD5('lavieestbelle78'), 'jean-marc', NOW());" );

/* Formatage pour affichage dans la JSP finale. */
messages.add( "Résultat de la requête d'insertion : " + statut + " ." );

```

Nous récupérons ici le retour de la requête **INSERT** dans l'entier **statut**, qui pour rappel vaut 1 en cas de succès de l'insertion et 0 en cas d'échec, et le stockons dans la liste **messages** pour affichage final dans notre JSP.

Accédez à nouveau à la page de test depuis votre navigateur, et constatez le résultat à la figure suivante.



Tests JDBC

1. Chargement du driver...
2. Driver chargé !
3. Connexion à la base de données...
4. Connexion réussie !
5. Résultat de la requête d'insertion : 1.
6. Requête "SELECT id, email, mot_de_passe, nom FROM Utilisateur;" effectuée !
7. Données retournées par la requête : id = 1, email = coyote@mail.acme, motdepasse = 7629ae53dbd1743ebbec21bc6bccbf45, nom = Coyote.
8. Données retournées par la requête : id = 2, email = jadorejquery@unefois.be, motdepasse = 0094bcab75591116f732cc180949c16d, nom = Thunderseb.
9. Données retournées par la requête : id = 3, email = jmarc@mail.fr, motdepasse = fe9bdbba3a8df1a2397a5132b3276457a, nom = jean-marc.
10. Fermeture de l'objet ResultSet.
11. Fermeture de l'objet PreparedStatement.
12. Fermeture de l'objet Connection.

La requête de sélection retourne bien une ligne supplémentaire, et il s'agit bien des données que nous venons d'insérer. Seulement il nous manque un petit quelque chose...



Comment récupérer l'id auto-généré par la table lors de l'insertion d'une ligne ?

Dans l'exemple, nous pouvons observer que l'id de la nouvelle ligne insérée vaut 3 parce que nous effectuons un **SELECT** par la suite, mais dans une vraie application nous n'allons pas nous amuser à effectuer une requête de lecture après chaque insertion simplement pour récupérer cette information ! Le problème, c'est que la méthode `executeUpdate()` ne retourne pas cette valeur, elle retourne seulement un entier dont la valeur indique le succès ou non de l'insertion.

Eh bien rassurez-vous, car il existe **une variante de la méthode `executeUpdate()`** qui remplit exactement cette fonction ! Regardez sa documentation, vous observerez qu'elle prend un argument supplémentaire en plus de la requête SQL à effectuer. Il s'agit d'un entier qui peut prendre uniquement deux valeurs, qui sont définies par les deux constantes `Statement.RETURN_GENERATED_KEYS` et `Statement.NO_GENERATED_KEYS` (qui en réalité valent respectivement 1 et 2). Leurs noms parlent d'eux-mêmes : la première permet la récupération de l'id généré, et la seconde ne le permet pas. Mais attention, il ne suffit pas de passer la valeur `Statement.RETURN_GENERATED_KEYS` en tant que second argument à la méthode `executeUpdate()` pour qu'elle se mette à retourner l'id souhaité. Non, la méthode continue à ne retourner qu'un simple entier indiquant le succès ou non de l'insertion. Ce qui change par contre, c'est qu'il devient alors possible de récupérer un `ResultSet`, le même objet que celui utilisé pour récupérer le retour d'une requête de lecture. Voyez plutôt :

Code : Java - com.sdzee.bdd.TestJDBC

```

/* Exécution d'une requête d'écriture avec renvoi de l'id auto-
généré */
int statut = statement.executeUpdate( "INSERT INTO Utilisateur
(email, mot_de_passe, nom, date_inscription) VALUES
('jmarc2@mail.fr', MD5('lavieestbelle78'), 'jean-marc', NOW())" ,
Statement.RETURN_GENERATED_KEYS);

/* Formatage pour affichage dans la JSP finale. */
messages.add( "Résultat de la requête d'insertion : " + statut + " ." );
;

/* Récupération de l'id auto-généré par la requête d'insertion. */
resultat = statement.getGeneratedKeys();
/* Parcours du ResultSet et formatage pour affichage de la valeur
qu'il contient dans la JSP finale. */
while ( resultat.next() ) {
    messages.add( "ID retourné lors de la requête d'insertion :" +
resultat.getInt( 1 ) );
}

```

En remplaçant la requête d'insertion précédente par ce nouvel appel, vous obtiendrez alors cette ligne (voir figure suivante) parmi les messages affichés.

6. ID retourné lors de la requête d'insertion : 4

Vous observez que l'id créé par MySQL pour l'entrée insérée en base est bien présent dans le ResultSet retourné par statement.getGeneratedKeys() !



Si vous n'obtenez pas ces résultats et/ou si votre code génère des exceptions, reportez-vous aux causes éventuelles et aux pistes de résolution exposées précédemment.

Les limites du système

Nous sommes dorénavant capables d'interroger notre base de données, mais comme toujours, il y a un "mais" ! Dans notre mise en pratique, tout fonctionne à merveille, mais nous allons découvrir un cas d'utilisation qui peut poser de gros problèmes de sécurité.

Insérer des données saisies par l'utilisateur

Reprenons notre exemple d'insertion de données précédent, mais cette fois au lieu d'insérer des données écrites en dur dans le code, nous allons confier à l'utilisateur la tâche de saisir ces données. Nous n'allons pas nous embêter à mettre en place un formulaire pour quelque chose d'aussi basique, nous allons nous contenter de paramètres placés directement dans l'URL de la requête. Concrètement, nous allons modifier légèrement le code de notre objet Java pour que lorsque le client saisit une URL de la forme `http://localhost:8080/pro/testjdbc?nom=aaa&motdepasse=bbb&email=ccc`, notre application soit capable de récupérer et utiliser les données ainsi transmises.

C'est très simple, il nous suffit pour cela de remplacer la ligne d'appel à la méthode statement.executeUpdate() par les lignes suivantes :

Code : Java - com.sdzee.bdd.TestJDBC

```
/* Récupération des paramètres d'URL saisis par l'utilisateur */
String paramEmail = request.getParameter( "email" );
String paramMotDePasse = request.getParameter( "motdepasse" );
String paramNom = request.getParameter( "nom" );

/* Exécution d'une requête d'écriture */
int statut = statement.executeUpdate( "INSERT INTO Utilisateur
(email, mot_de_passe, nom, date_inscription)
VALUES ('" + paramEmail + "', MD5('" + paramMotDePasse +
"'), '" + paramNom + "', NOW());" );
```

Avec tout ce que nous avons codé dans les chapitres précédents, vous devez comprendre ces lignes aisément ! Nous récupérons ici simplement les paramètres d'URL via la méthode request.getParameter(), et les concaténons directement à notre INSERT pour former une requête valide.

Le problème des valeurs nulles

Voyons pour commencer un premier inconvénient majeur avec cette technique. Afin de générer une requête valide, nous sommes obligés de préciser des paramètres dans l'URL d'appel de notre page de test. Si nous nous contentons d'appeler bêtement l'URL `http://localhost:8080/pro/testjdbc` sans paramètres, les appels aux méthodes request.getParameter() vont retourner des chaînes de caractères indéfinies, c'est-à-dire initialisées à null.

Si vous avez prêté attention à la requête CREATE que nous avions utilisée pour mettre en place notre table Utilisateur dans la base de données, vous devez vous souvenir que nous avions précisé une contrainte NOT NULL sur les champs nom, motdepasse et email. Ainsi, notre base de données doit nous retourner une erreur si nous tentons d'insérer des valeurs indéfinies dans la table Utilisateur.

Faites bêtement le test, et appelez votre URL de test sans paramètres. Vous observez alors au sein de la page affichée par votre JSP cette ligne (voir figure suivante).

10. Données retournées par la requête : id = 4, email = null, motdepasse = 37a6259cc0c1dae299a7866489dff0bd, nom = null.



Que s'est-il passé ? Pourquoi est-ce qu'une ligne a été insérée avec des valeurs indéfinies ?

Ne soyez pas leurrés ici, les valeurs que vous avez insérées dans votre base ne valent pas `null` au sens Java du terme ! En réalité, les paramètres que vous récupérez dans votre application Java sont bien indéfinis, mais la construction de la requête par concaténation produit la chaîne suivante :

Code : SQL

```
INSERT INTO Utilisateur (email, mot_de_passe, nom, date_inscription)
VALUES ('null',MD5('null'),'null',NOW());
```

Ainsi, si des valeurs apparaissent comme étant égales à `null` ce n'est pas parce que la contrainte `NOT NULL` n'a pas été respectée, c'est parce que notre requête a été mal formée ! Alors que les champs `email` et `nom` devraient effectivement être indéfinis, ils sont ici initialisés avec le contenu '`null`' et la requête d'insertion fonctionne donc comme si ces champs étaient correctement renseignés. La requête correctement formée devrait être la suivante :

Code : SQL

```
INSERT INTO Utilisateur (email, mot_de_passe, nom, date_inscription)
VALUES (null,MD5(null),null,NOW());
```

Notez bien l'absence des apostrophes autour des `null`. Si notre base de données recevait une telle requête, elle la refuserait, car elle y détecterait correctement les valeurs indéfinies.



Par ailleurs, vous remarquerez peut-être sur l'image que le champ `mot_de_passe` ne contient pas '`null`'. Cela provient tout simplement du fait que nous utilisons la fonction `MD5()` dans notre requête SQL, qui génère un *hash* de la chaîne contenant '`null`'. Si nous n'utilisions pas cette fonction, la valeur de ce champ serait, elle aussi, initialisée à '`null`' !



Comment pallier ce problème d'insertion de valeurs "nulles" ?

Pour éviter ce désagrément, plusieurs solutions s'offrent à nous. La plus simple pour le moment, c'est de mettre en place une vérification dans le code de notre objet Java sur le retour de chaque appel à la méthode `request.getParameter()`, juste avant l'exécution de la requête. Voici ce que devient alors notre code :

Code : Java - com.sdzeebdd.TestJDBC

```
/* Récupération des paramètres d'URL saisis par l'utilisateur */
String paramEmail = request.getParameter( "email" );
String paramMotDePasse = request.getParameter( "motdepasse" );
String paramNom = request.getParameter( "nom" );

if ( paramEmail != null && paramMotDePasse != null && paramNom != null ) {
    /* Exécution d'une requête d'écriture */
    int statut = statement.executeUpdate( "INSERT INTO Utilisateur
(email, mot_de_passe, nom, date_inscription)
VALUES ('" + paramEmail + "', MD5('" +
paramMotDePasse + "'), '" + paramNom + "', NOW())" );
}

/* Formatage pour affichage dans la JSP finale. */
messages.add( "Résultat de la requête d'insertion : " + statut +
". " );
}
```

Grâce à ce simple bloc `if`, si un des appels à `request.getParameter()` retourne une chaîne indéfinie, alors la requête n'est pas effectuée. Appelez à nouveau la page de test sans paramètres, vous constaterez que la requête n'est plus effectuée.



Cela dit, c'est très contraignant de procéder de cette manière : si demain la structure de notre table change et qu'il devient par exemple possible de ne pas transmettre le paramètre **nom**, alors il faudra penser à modifier cette condition dans notre code pour autoriser une chaîne **paramNom** indéfinie...

Le cas idéal

Essayons maintenant d'insérer des valeurs correctement, en saisissant une URL qui contient bien les trois paramètres attendus. Appelez par exemple la page de test avec l'URL suivante :

Code : URL

```
http://localhost:8080/pro/testjdbc?  
nom=Marcel&motdepasse=pastèque&email=marcel@mail.fr
```

Vous observerez alors une nouvelle ligne dans la page affichée par votre JSP (voir la figure suivante).

11. Données retournées par la requête : `id = 5, email = marcel@mail.fr, motdepasse = 98951f4898ab8da14b2c8ff93a953b98, nom = Marcel.`

Tout va bien, notre code a bien inséré dans la table les données récupérées depuis l'URL transmise par l'utilisateur.



Que va-t-il se passer si nous omettons de renseigner un ou plusieurs paramètres ?

Bonne question, il reste effectivement un cas que nous n'avons pas encore traité. Si vous accédez à la page de test avec l'URL suivante :

Code : URL

```
http://localhost:8080/pro/testjdbc?nom=&motdepasse=&email=
```

Alors les appels aux méthodes `request.getParameter()` ne vont cette fois plus retourner **null** comme c'était le cas lorsque nous accédions à la page sans aucun paramètre, mais des chaînes vides. À la figure suivante, le résultat alors affiché.

12. Données retournées par la requête : `id = 6, email = , motdepasse = d41d8cd98f00b204e9800998ecf8427e, nom = .`

Vous observez ici que les valeurs vides sont insérées correctement dans la table. Cela est dû au fait que MySQL sait faire la distinction entre un champ vide et un champ nul.



Sachez toutefois que ce n'est pas le cas de toutes les solutions existantes sur le marché. Certains systèmes, comme les bases de données Oracle par exemple, considèrent par défaut qu'un champ vide est équivalent à un champ nul. Ainsi, notre dernier exemple serait refusé par la base si nous travaillions avec le SGBD Oracle.

Les injections SQL

Dans ce dernier exemple tout va bien, car aucune embûche ne gêne notre parcours, nous évoluons dans un monde peuplé de bisounours. 😊 Mais dans la vraie vie, vous savez très bien que tout n'est pas aussi merveilleux, et vous devez toujours vous rappeler la règle d'or que je vous ai déjà communiquée à plusieurs reprises : « **il ne faut jamais faire confiance à l'utilisateur** ».



Quel est le problème exactement ?

Ok... Puisque vous semblez encore une fois sceptiques, changeons l'URL de notre exemple précédent par celle-ci :

Code : URL

```
http://localhost:8080/pro/testjdbc?  
nom=Marcel'&motdepasse=pastèque&email=marcel@mail.fr
```

Nous avons simplement ajouté une apostrophe au contenu du paramètre **nom**. Faites le test, et vous observerez alors cette ligne dans la page affichée par votre JSP (voir la figure suivante).

5. Erreur lors de la connexion :
You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near "Marcel", NOW())' at line 1



Que s'est-il passé ?

Ce type de failles porte un nom, il s'agit d'une **injection SQL**. En envoyant une chaîne contenant le caractère apostrophe, qui pour rappel est le caractère qui sert à délimiter les valeurs dans notre requête d'insertion, l'utilisateur a fait échouer notre requête. En effet, après concaténation du contenu des paramètres, notre requête devient :

Code : SQL

```
INSERT INTO Utilisateur (email, mot_de_passe, nom, date_inscription)  
VALUES ('marcel@mail.fr', MD5('pastèque'), 'Marcel', NOW());
```

Vous voyez bien où est le problème ici : dans votre requête, deux apostrophes se suivent, ce qui rend sa syntaxe erronée et provoque l'envoi d'une SQLException.



D'accord, l'utilisateur peut provoquer l'envoi d'une exception. Ce n'est pas catastrophique non plus...

En l'occurrence non, ça ne nous embête pas plus que ça, car ça ne pénalise que l'utilisateur, qui se retrouve avec un message d'erreur devant les yeux. Mais rendez-vous bien compte que **ce type de failles peut être très dangereux** pour d'autres types de requêtes ! L'objet de ce chapitre n'est pas de faire de vous des pirates en herbe, nous n'allons donc pas nous attarder sur les cas qui posent de gros problèmes de sécurité, sachez simplement que si vous ne protégez pas efficacement vos requêtes, il peut devenir possible pour un utilisateur averti de modifier une requête à votre insu ou pire, d'effectuer des requêtes sur la base à votre insu ! En clair, votre application peut devenir une porte ouverte conduisant tout droit aux données de votre base...



Quelle est la parade contre ce type de risques ?

A priori, nous pourrions tenter de créer une méthode de vérification qui se chargerait d'analyser les paramètres saisis par l'utilisateur, et d'y éliminer ou échapper les caractères problématiques (notamment l'apostrophe que nous avons utilisée dans notre exemple). Rassurez-vous, il existe bien plus propre, bien plus standard et surtout bien plus efficace : **les requêtes préparées** !

Les requêtes préparées

Pourquoi préparer ses requêtes ?

Nous allons ici découvrir un nouvel objet, nommé `PreparedStatement`. En parcourant sa documentation, vous vous apercevez qu'il s'agit en réalité d'une nouvelle interface qui implémente l'interface `Statement` utilisée dans nos précédents exemples. Comme son nom l'indique, cet objet permet de créer des **requêtes préparées**.



Pourquoi devrions-nous utiliser de telles requêtes ?

En étudiant un peu plus attentivement la documentation de l'objet, vous découvrirez alors qu'il présente trois différences majeures avec un `Statement` classique :

- l'utilisation de `PreparedStatement` peut permettre de pré-compiler une requête SQL ;
- une requête SQL ainsi créée peut être paramétrée, c'est-à-dire contenir des trous ou jokers qui ne seront comblés que lors de son exécution ;
- une requête SQL ainsi créée est protégée contre les injections SQL, et contre d'éventuelles erreurs sur les types des paramètres.

1. Des requêtes pré-compilées

Le premier avantage se situe au niveau des performances. Dès lors que vous souhaitez exécuter une requête d'un même objet `Statement` plusieurs fois, il devient intéressant d'utiliser un `PreparedStatement` à la place, afin de réduire le temps d'exécution. En effet, la principale fonctionnalité de ce nouvel objet est qu'à la différence d'un classique objet `Statement`, il prend en argument une requête SQL dès sa création.



Quel est l'intérêt de passer une requête à l'objet dès sa création ?

L'avantage, c'est que dans la plupart des cas (*), la requête SQL sera, dès la création de l'objet, directement envoyée au SGBD, où elle sera compilée. Pour être exact, un objet `PreparedStatement` ne contient donc pas simplement une requête SQL, mais une requête SQL pré-compilée. Concrètement, cela signifie que lorsque ce `PreparedStatement` sera exécuté, le SGBD n'aura plus qu'à exécuter la requête sans avoir besoin de la compiler au préalable.

(*). Note : tous les drivers JDBC ne procèdent pas de cette manière, certains n'effectuent pas l'envoi de la requête vers le serveur SQL pour pré-compilation lors de sa création via un `PreparedStatement`. Ces différences de comportement d'un driver à l'autre existent parce que cette fonctionnalité n'est pas définie noir sur blanc dans les spécifications de l'API JDBC.

2. Des requêtes paramétrées

Le second avantage réside dans le fait qu'à travers un tel objet, il est possible de créer des requêtes qui prennent en compte des paramètres. Ceux-ci sont représentés par des points d'interrogation ? dans la requête qui sert de modèle, et doivent être précisés avant l'exécution. Si vous n'êtes pas familiers avec ce concept, vous en apprendrez davantage dans [ce chapitre du cours de MySQL](#). Le principe est simple, il s'agit de créer une requête modèle qui contient un ou plusieurs trous. Exemple d'une requête attendant un seul paramètre :

Code : SQL - Exemple de requête paramétrée

```
SELECT * FROM Utilisateur WHERE email = ?
```

Vous observez ici le caractère joker ? dont je viens de vous parler. Lorsque vous passez une telle requête à un objet `PreparedStatement`, celui-ci va la faire pré-compiler et se chargera ensuite de remplacer le paramètre manquant par la valeur que vous souhaitez lui donner au moment de l'exécution.



Ainsi, la pré-compilation, couplée à la prise en compte de paramètres au moment de l'exécution, permet d'améliorer considérablement les performances de requêtes paramétrées destinées à être exécutées plusieurs fois. Cela dit, ces conditions ne sont pas un prérequis à l'utilisation de requêtes préparées : il est tout à fait possible de préparer des requêtes qui n'attendent pas de paramètres ou qui ne seront exécutées qu'une seule fois.

3. Des requêtes protégées !

Le dernier avantage et le plus important de tous, c'est bien évidemment celui que je vous ai annoncé en conclusion du paragraphe sur les injections SQL : en utilisant des requêtes préparées, vous prévenez tout risque de failles de ce type ! Et cette fonctionnalité, contrairement à l'étape de pré-compilation, est disponible quel que soit le driver JDBC utilisé.

En outre, non seulement vos requêtes seront protégées contre les injections SQL, mais le passage des paramètres s'en retrouvera également grandement facilité, et ce quel que soit le type du paramètre passé : `String`, `Date`, etc., et même la valeur `null` ! Nous allons revenir sur ce point dans la mise en pratique à venir.

Comment préparer ses requêtes ?

Vous n'allez pas être dépayrés, le fonctionnement est assez semblable à celui d'un Statement classique.

Initialisation de l'objet

La première différence se situe là où, auparavant, nous effectuions cette initialisation :

Code : Java - Initialisation de l'objet Statement

```
/* Cr ation de l'objet g rant les requ etes */
Statement statement = connexion.createStatement();
```

Nous allons désormais directement préciser la requête SQL dans cette nouvelle initialisation :

Code : Java - Initialisation de l'objet PreparedStatement

```
/* Cr ation de l'objet g rant la requ ete pr par e d finie */
PreparedStatement preparedStatement = connexion.prepareStatement(
    "SELECT id, email, mot_de_passe, nom FROM Utilisateur");
```

La seule différence est l'appel à la méthode `prepareStatement()` de l'objet `Connection` qui attend, comme je vous l'ai déjà annoncé, une requête SQL en argument.

Ex ecution de la requ ete

Puisque votre requête est déjà déclarée lors de l'initialisation, la seconde différence avec l'utilisation d'un Statement classique se situe au niveau de l'appel à la méthode d'exécution. Alors qu'auparavant nous effectuions par exemple :

Code : Java - Ex ecution d'une requ ete de type SELECT avec un Statement classique

```
/* Ex ecution d'une requ ete de lecture */
statement.executeQuery( "SELECT id, email, mot_de_passe, nom FROM
Utilisateur");
```

Nous allons cette fois simplement appeler :

Code : Java - Ex ecution d'une requ ete de type SELECT avec un PreparedStatement

```
preparedStatement.executeQuery();
```

Vous l'aurez déjà remarqué si vous avez attentivement parcouru la documentation de l'objet `PreparedStatement`, ses méthodes `executeQuery()` et `executeUpdate()` n'attendent logiquement aucun argument.

Et avec des param tres ?

En effet tout cela est bien joli, mais préparer ses requêtes est surtout utile lorsque des paramètres interviennent. Pour rappel, voici comment nous procédions avec un Statement classique :

Code : Java - Ex ecution d'une requ ete param tr e avec un Statement classique

```

/* Création de l'objet gérant les requêtes */
statement = connexion.createStatement();

/* Récupération des paramètres d'URL saisis par l'utilisateur */
String paramEmail = request.getParameter( "email" );
String paramMotDePasse = request.getParameter( "motdepasse" );
String paramNom = request.getParameter( "nom" );

if ( paramEmail != null && paramMotDePasse != null && paramNom != null ) {
    /* Exécution d'une requête d'écriture */
    int statut = statement.executeUpdate( "INSERT INTO Utilisateur
(email, mot_de_passe, nom, date_inscription)
VALUES ('" + paramEmail + "', MD5('" +
paramMotDePasse + "'), '" + paramNom + "', NOW());" );
}

```

Et voilà comment procéder avec une requête préparée. Je vous donne le code, et vous commente le tout ensuite :

Code : Java - Exécution d'une requête paramétrée avec un PreparedStatement

```

/* Création de l'objet gérant les requêtes préparées */
PreparedStatement = connexion.prepareStatement( "INSERT INTO
Utilisateur (email, mot_de_passe, nom, date_inscription) VALUES (?,?
MD5(?), ?, NOW());" );

/* Récupération des paramètres d'URL saisis par l'utilisateur */
String paramEmail = request.getParameter( "email" );
String paramMotDePasse = request.getParameter( "motdepasse" );
String paramNom = request.getParameter( "nom" );

/*
 * Remplissage des paramètres de la requête grâce aux méthodes
 * setXXX() mises à disposition par l'objet PreparedStatement.
 */
PreparedStatement.setString( 1, paramEmail );
PreparedStatement.setString( 2, paramMotDePasse );
PreparedStatement.setString( 3, paramNom );

/* Exécution de la requête */
int statut = preparedStatement.executeUpdate();

```

Pour commencer, à la ligne 2 vous observez la définition de la requête SQL dès l'initialisation du PreparedStatement. Celle-ci attend trois paramètres, signalés par les trois caractères ?.

Vous remarquez ensuite l'étape de remplissage des paramètres de la requête, dans les lignes 13 à 15. Il s'agit en réalité tout simplement de remplacer les ? par les valeurs des paramètres attendus. Cela se fait par l'intermédiaire des méthodes setXXX () mises à disposition par l'objet PreparedStatement, dont je vous laisse le loisir de parcourir la documentation si vous ne l'avez pas déjà fait. Celles-ci devraient grandement vous rappeler la flopée de méthodes getXXX () que nous avions découvertes dans l'objet ResultSet, puisqu'il en existe là encore une pour chaque type géré :

- une méthode preparedStatement.setInt () pour définir un entier ;
- une méthode preparedStatement.setString () pour définir une chaîne de caractères ;
- une méthode preparedStatement.setBoolean () pour définir un booléen ;
- ...

La plupart de ces méthodes attendent simplement deux arguments :

- un entier définissant le paramètre à remplacer ;
- un objet du type concerné destiné à remplacer le paramètre dans la requête SQL.

En l'occurrence, notre requête attend, dans l'ordre, un email, un mot de passe et un nom. Il s'agit là de trois chaînes de caractères, voilà pourquoi dans notre exemple nous utilisons à trois reprises la méthode preparedStatement.setString (),

appliquée respectivement aux paramètres numérotés 1, 2 et 3, représentant la position du ? à remplacer dans la requête. Ainsi, le contenu de notre chaîne **paramEmail** remplace le premier ?, le contenu de **paramMotDePasse** le second, et le contenu de **paramNom** le troisième.



Facile, n'est-ce pas ? Vous comprenez maintenant mieux pourquoi je vous avais annoncé qu'un `PreparedStatement` facilitait le passage de paramètres : finie la concaténation barbare directement au sein de la requête SQL, tout passe désormais par des méthodes standard, et c'est bien plus propre ainsi ! 😊

Enfin, vous remarquez à la ligne 18 l'exécution de la requête d'insertion via un simple appel à la méthode `preparedStatement.executeUpdate()`.

Mise en pratique

Nous pouvons maintenant utiliser des requêtes préparées dans notre classe d'exemple, afin de tester leur bon fonctionnement et leurs différences avec les requêtes non préparées. Vous devez remplacer le code suivant dans notre objet :

Code : Java - Code faisant intervenir un Statement classique

```
/* Création de l'objet gérant les requêtes */
statement = connexion.createStatement();

/* Récupération des paramètres d'URL saisis par l'utilisateur */
String paramEmail = request.getParameter( "email" );
String paramMotDePasse = request.getParameter( "motdepasse" );
String paramNom = request.getParameter( "nom" );

if ( paramEmail != null && paramMotDePasse!= null && paramNom != null ) {
    /* Exécution d'une requête d'écriture */
    int statut = statement.executeUpdate( "INSERT INTO Utilisateur
(email, mot_de_passe, nom, date_inscription)
VALUES ('" + paramEmail + "', MD5('" +
paramMotDePasse + "'), '" + paramNom + "', NOW());" );

    /* Formatage pour affichage dans la JSP finale. */
    messages.add( "Résultat de la requête d'insertion : " + statut +
".." );
}
```

Par ce nouveau code :

Code : Java - Code faisant intervenir un PreparedStatement

```
/* Création de l'objet gérant les requêtes préparées */
PreparedStatement = connexion.prepareStatement( "INSERT INTO
Utilisateur (email, mot_de_passe, nom, date_inscription) VALUES (?,?
MD5(?), ?, NOW());" );
messages.add( "Requête préparée créée !" );

/* Récupération des paramètres d'URL saisis par l'utilisateur */
String paramEmail = request.getParameter( "email" );
String paramMotDePasse = request.getParameter( "motdepasse" );
String paramNom = request.getParameter( "nom" );

/*
 * Remplissage des paramètres de la requête grâce aux méthodes
 * setXXX() mises à disposition par l'objet PreparedStatement.
 */
PreparedStatement.setString( 1, paramEmail );
PreparedStatement.setString( 2, paramMotDePasse );
PreparedStatement.setString( 3, paramNom );
```

```

/* Exécution de la requête */
int statut = preparedStatement.executeUpdate();

/* Formatage pour affichage dans la JSP finale. */
messages.add( "Résultat de la requête d'insertion préparée : " +
statut + "." );

```

De même, n'oubliez pas de remplacer la création de la requête de sélection :

Code : Java - Ancien mode d'établissement de la requête de sélection

```

/* Exécution d'une requête de lecture */
resultat = statement.executeQuery( "SELECT id, email, mot_de_passe,
nom FROM Utilisateur;" );
messages.add( "Requête \"SELECT id, email, mot_de_passe, nom FROM
Utilisateur;\" effectuée !" );

```

Par le code suivant :

Code : Java - Nouveau mode d'établissement de la requête de sélection

```

/* Crédit à l'objet gérant les requêtes préparées */
PreparedStatement = connexion.prepareStatement( "SELECT id, email,
mot_de_passe, nom FROM Utilisateur;" );
messages.add( "Requête préparée créée !" );

/* Exécution d'une requête de lecture */
resultat = preparedStatement.executeQuery();
messages.add( "Requête \"SELECT id, email, mot_de_passe, nom FROM
Utilisateur;\" effectuée !" );

```

Remarquez au passage que contrairement à un Statement classique, qui peut être réutilisé plusieurs fois pour exécuter des requêtes différentes, il est nécessaire avec un PreparedStatement de réinitialiser l'objet via un nouvel appel à la méthode connexion.prepareStatement(). Ceci est tout simplement dû au fait que la requête SQL est passée lors de la création de l'objet, et non plus lors de l'appel aux méthodes d'exécution.

Enfin, il faut bien entendu remplacer l'initialisation du Statement par un PreparedStatement ; et la fermeture de la ressource Statement par la fermeture de la ressource PreparedStatement :

Code : Java - Fermeture du Statement

```

messages.add( "Fermeture de l'objet Statement." );
if ( statement != null ) {
    try {
        statement.close();
    } catch ( SQLException ignore ) {
    }
}

```

Code : Java - Fermeture du PreparedStatement

```

messages.add( "Fermeture de l'objet PreparedStatement." );
if ( preparedStatement != null ) {
    try {
        preparedStatement.close();
    } catch ( SQLException ignore ) {
    }
}

```

```

    }
}

```

Une fois toutes ces modifications effectuées et enregistrées, suivez alors ce bref scénario de tests :

- accédez à l'URL sans aucun paramètre `http://localhost:8080/pro/testjdbc`, et observez le résultat à la figure suivante.

6. Erreur lors de la connexion :
Column 'email' cannot be null

- accédez à l'URL correctement renseignée `http://localhost:8080/pro/testjdbc?nom=Raymond&motdepasse=knySna&email=raymond@anpe.fr`, et observez le résultat à la figure suivante.

15. Données retournées par la requête : id = 8, email = raymond@anpe.fr, motdepasse = efb5d722dc8283932a43e0ec71e93caf, nom = Raymond.

- accédez à l'URL contenant des paramètres vides `http://localhost:8080/pro/testjdbc?nom=&motdepasse=&email=`, et observez le résultat à la figure suivante.

6. Erreur lors de la connexion :
Duplicate entry " for key 'email'

- accédez à l'URL contenant une tentative d'injection SQL `http://localhost:8080/pro/testjdbc?nom=Charles'&motdepasse=doucefrance&email=charles@lamer.fr`, et observez le résultat à la figure suivante.

16. Données retournées par la requête : id = 11, email = charles@lamer.fr, motdepasse = 3ca1313820d5ba507202c91a65910e22, nom = Charles'.

Vous remarquez ici deux différences majeures avec les requêtes non préparées :

- dans le premier test, le passage des valeurs indéfinies est cette fois correctement réalisé. La base de données refuse alors logiquement la requête, en précisant ici que le champ **email** ne peut pas contenir **null** ;
- dans le dernier test, le paramètre contenant une apostrophe est cette fois correctement inséré dans la table, la requête n'échoue plus !

En outre, vous remarquez dans le troisième test qu'une erreur est renvoyée lors de la tentative d'insertion des valeurs vides. Cela signifie que la requête préparée autorise logiquement les valeurs vides tout comme le faisait le Statement classique. Mais, puisque dans notre table il y a une contrainte d'unicité sur le champ **email** et qu'il existe déjà une ligne contenant une adresse vide, le SGBD n'accepte pas d'insérer une nouvelle ligne contenant encore une telle adresse.



Vous devez maintenant être convaincus de l'intérêt des requêtes préparées : elles facilitent le passage de valeurs à une requête paramétrée, et protègent automatiquement les requêtes contre tout type d'injections !

Dorénavant, je ne veux plus vous voir utiliser de Statement classique. En trois mots : préparez vos requêtes !

Avant de passer à la suite, il nous reste encore à découvrir comment récupérer l'id auto-généré après exécution d'une requête d'insertion. En effet, vous savez qu'avec un Statement classique nous précisons un paramètre supplémentaire lors de l'exécution de la requête :

Code : Java

```

/* Exécution d'une requête d'écriture avec renvoi de l'id auto-
généré */
int statut = statement.executeUpdate( "INSERT INTO Utilisateur
(email, mot_de_passe, nom, date_inscription) VALUES
('jmarc@mail.fr', MD5('lavieestbelle78'), 'jean-marc', NOW())" );

```

```
Statement.RETURN_GENERATED_KEYS);
```

Eh bien de la même manière que nous devons préciser la requête SQL, non pas à l'exécution mais dès la création d'un PreparedStatement, nous allons devoir préciser dès la création si nous souhaitons récupérer l'id auto-généré ou non, en utilisant une [cousine de la méthode connexion.prepareStatement\(\)](#), qui attend en second argument les mêmes constantes que celles utilisées avec le Statement :

Code : Java

```
/* Création d'un PreparedStatement avec renvoi de l'id auto-généré */
PreparedStatement preparedStatement = connexion.prepareStatement(
    "INSERT INTO Utilisateur (email, mot_de_passe, nom,
    date_inscription) VALUES ('jmarc@mail.fr', MD5('lavieestbelle78'),
    'jean-marc', NOW());", Statement.RETURN_GENERATED_KEYS );
```

Le reste ne change pas : il faut parcourir le ResultSet retourné par la méthode `preparedStatement.getGeneratedKeys()` et y récupérer le premier champ, tout comme nous l'avions fait avec un Statement classique.

- Il est nécessaire de charger le driver JDBC une seule et unique fois, au démarrage de l'application par exemple.
- JDBC fonctionne par un système d'URL. Dans le cas de MySQL :
`jdbc:mysql://hôte:port/nom_de_la_bdd`.
- Il est nécessaire d'établir une connexion entre l'application et la base de données, via un appel à `DriverManager.getConnection()` qui retourne un objet `Connection`.
- Un appel à `connexion.createStatement()` retourne un objet `Statement`, qui permet d'effectuer des requêtes via notamment ses méthodes `executeQuery()` et `executeUpdate()`.
- L'emploi de cet objet est à éviter, celui-ci étant sujet à de nombreux problèmes dont les dangereuses injections SQL.
- Préparer ses requêtes permet une protection contre ces injections, une gestion simplifiée des paramètres et dans certains cas de meilleures performances.
- Un appel à `connexion.prepareStatement()` retourne un objet `PreparedStatement`, qui permet d'effectuer des requêtes de manière sécurisée, via notamment ses méthodes `executeQuery()` et `executeUpdate()`.
- Un ensemble de *setters* facilite l'utilisation de paramètres dans les requêtes effectuées.
- Il est nécessaire de **libérer les ressources** utilisées lors d'un échange avec la base de données, en fermant les différents objets intervenant dans un ordre précis via des appels à leur méthode `close()`.

Le modèle DAO

Nous avons fait un grand pas en avant en découvrant la technologie JDBC, mais il est déjà nécessaire de nous poser certaines questions. Dans ce chapitre, nous allons :

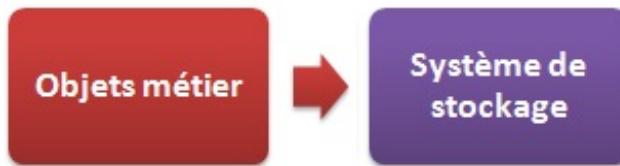
- lister les problèmes liés à la liaison directe entre nos objets métier et notre base de données ;
- découvrir et mettre en place le *design pattern* DAO, qui répond parfaitement à cette problématique ;
- apprendre à l'intégrer proprement dans notre application web.

Objectifs

Inconvénients de notre solution

Dans le précédent chapitre, nous avons uniquement testé JDBC dans un bac à sable, sans vraiment nous soucier de son intégration dans le cas d'une vraie application. Nous avons certes pris la peine de découper un minimum notre exemple, en séparant nettement le trio vue, contrôleur et modèle, mais ce n'est pas suffisant.

Voilà à la figure suivante une représentation globale de ce que nous avons actuellement, avec en rouge les objets de notre couche modèle directement en contact avec le système de stockage.



Pourquoi n'est-ce pas suffisant ? La gestion des données est bien effectuée dans le modèle !

En effet, en théorie MVC est ainsi bien respecté. Mais dans la pratique, dans une application qui est constituée de plusieurs centaines voire milliers de classes, cela constitue un réel problème de conception : si nous codons une vraie application de cette manière, nous lions alors très fortement - pour ne pas dire mélangeons - le code responsable des traitements métier au code responsable du stockage des données. Si bien qu'en fin de compte, il devient impossible d'exécuter séparément l'un ou l'autre. Et ceci est fâcheux pour plusieurs raisons :

- il est impossible de mettre en place des tests unitaires :
 - impossible de tester le code métier de l'application sans faire intervenir le stockage (BDD, etc.) ;
 - impossible de ne tester que le code relatif au stockage des données, obligation de lancer le code métier.
- il est impossible de changer de mode de stockage. Que ce soit vers un autre SGBD, voire vers un système complètement différent d'une base de données, cela impliquerait une réécriture complète de tout le modèle, car le code métier est mêlé avec et dépendant du code assurant le stockage.

Vous trouverez toujours quelqu'un qui vous dira que si chaque composant n'est pas testable séparément ce n'est pas un drame, quelqu'un d'autre qui vous dira que lorsque l'on crée une application on ne change pas de mode de stockage du jour au lendemain, etc. Ces gens-là, je vous conseille de ne les écouter que d'une oreille : écrire un code orienté objet et bien organisé est une excellente pratique, et c'est ce que je vais vous enseigner dans cette partie du cours.

Je pourrais continuer la liste des inconvénients, en insistant notamment sur le fait qu'un code mélangé dans une couche "modèle" monolithique est bien plus difficile à maintenir et à faire évoluer qu'un code proprement découpé et organisé, mais je pense que vous avez déjà compris et êtes déjà convaincus. 😊



Pour information, ce constat n'est absolument pas limité à la plate-forme Java. Vous trouverez des solutions qui répondent à ce besoin dans n'importe quelle technologie web, citons par exemple les « *fat model* » avec Ruby on Rails.

Isoler le stockage des données

L'idée est qu'au lieu de faire communiquer directement nos objets métier avec la base de données, ou le système de fichiers, ou

les *webservices*, ou peu importe ce qui fait office de système de stockage, ceux-ci vont parler avec la couche DAO. Et c'est cette couche DAO qui va ensuite de son côté communiquer avec le système de stockage.

L'objectif de l'architecture que nous devons mettre en place n'est donc rien d'autre que l'isolement pur et simple du code responsable du stockage des données. Nous souhaitons en effet littéralement encapsuler ce code dans une couche plus ou moins hermétique, de laquelle aucune information concernant le mode de stockage utilisé ne s'échappe. En d'autres termes, notre objectif est de cacher la manière dont sont stockées les données au reste de l'application.

Voilà cette fois une représentation de ce que nous souhaitons mettre en place (voir figure suivante).



Vous visualisez bien que seul le DAO est en contact avec le système de stockage, et donc que seul lui en a connaissance. Le revers de la médaille, si c'en est un, c'est qu'afin de réaliser le cloisonnement du stockage des données, la création d'une nouvelle couche est nécessaire, c'est-à-dire l'écriture de codes supplémentaires et répétitifs.



En résumé, l'objectif du pattern DAO tient en une phrase : il permet de faire la distinction entre les données auxquelles vous souhaitez accéder, et la manière dont elles sont stockées.

Principe Constitution

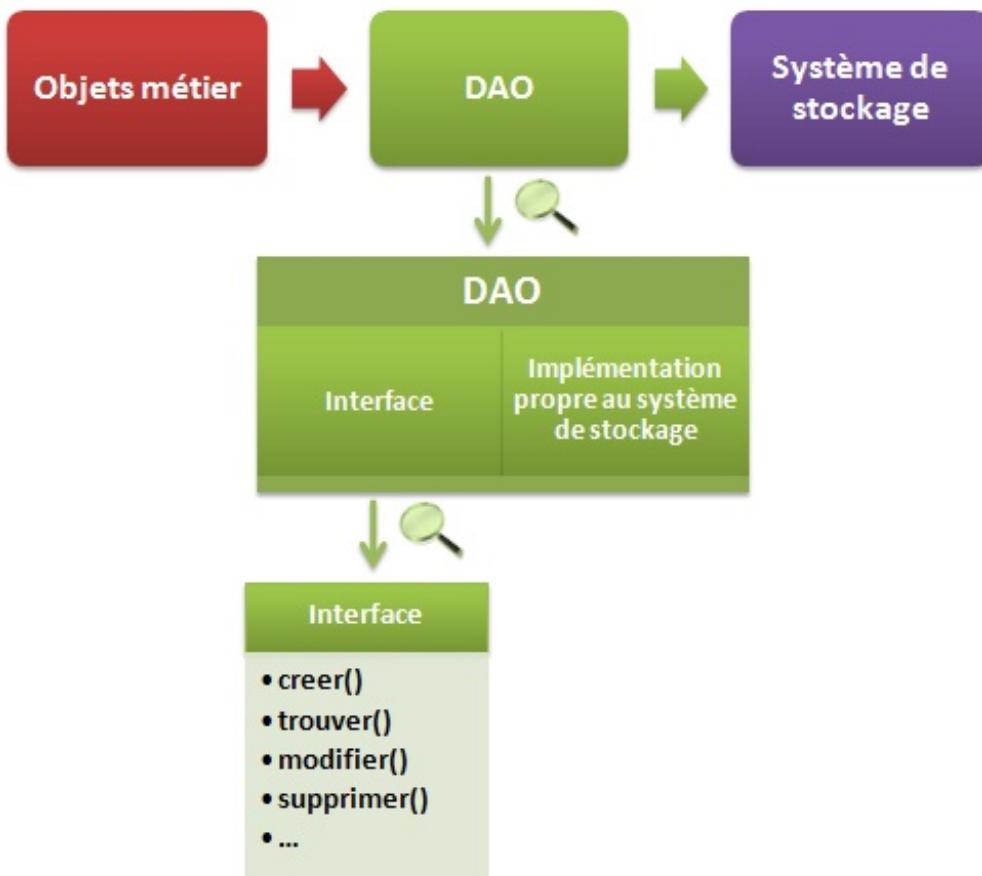
Le principe du pattern DAO est de séparer la couche modèle d'une application en deux sous-couches distinctes :

- une couche gérant les traitements métier appliqués aux données, souvent nommée couche service ou métier. Typiquement, tout le travail de validation réalisé dans nos objets **InscriptionForm** et **ConnexionForm** en fait partie ;
- une couche gérant le stockage des données, logiquement nommée couche de données. Il s'agit là des opérations classiques de stockage : la création, la lecture, la modification et la suppression. Ces quatre tâches basiques sont souvent raccourcies à l'anglaise en *CRUD*.

Pour réaliser efficacement une telle opération, il est nécessaire d'encapsuler les exceptions spécifiques au mode de stockage dans des exceptions personnalisées et propres à la couche DAO. Dans notre cas par exemple, nous allons devoir faire en sorte que les exceptions propres à SQL ou à JDBC ne soient pas vues comme telles par nos objets métier, mais uniquement comme des exceptions émanant de la « boîte noire » qu'est notre DAO.

De même, il va falloir masquer le code responsable du stockage au code « extérieur », et l'exposer uniquement via des interfaces. Dans notre cas, il s'agira donc de faire en sorte que le code basé sur JDBC soit bien à l'abri dans des implémentations de DAO, et que nos objets métier n'aient connaissance que des interfaces qui les décrivent.

Reprendons le schéma précédent et zoomons sur la couche DAO (voir la figure suivante).



La couche modèle est constituée de la couche métier (en rouge) et de la couche données (en vert). La couche métier n'a connaissance que des interfaces décrivant les objets de la couche données. Ainsi, peu importe le système de stockage final utilisé, du point de vue du code métier les méthodes à appeler ne changent pas, elles seront toujours celles décrites dans l'interface. C'est uniquement l'implémentation qui sera spécifique au mode de stockage.

Intégration

Ne soyez pas leurrés par le schéma précédent, la couche DAO ne va pas seulement contenir les interfaces et implémentations des méthodes *CRUD*. Elle va également renfermer quelques classes dédiées à l'isolement des concepts liés au mode de stockage, comme les exceptions dont nous avons déjà brièvement parlé, mais également le chargement du driver et l'obtention d'une connexion. Pour ce faire, nous allons créer une *Factory* (une fabrique) qui sera unique dans l'application, ne sera instanciée que si les informations de configuration sont correctes et aura pour rôle de fournir les implémentations des différents DAO.

En ce qui concerne la relation entre la couche métier et le DAO, c'est très simple : les objets métier appellent les méthodes *CRUD*, qui ont pour rôle de communiquer avec le système de stockage et de peupler les beans représentant les données.

Assez gambergé, du code vaut mieux qu'un long discours ! 😊

Création

Pour commencer, nous allons laisser tomber le bac à sable dans lequel nous avons évolué dans le chapitre précédent, et allons reprendre les systèmes d'inscription et de connexion que nous avions utilisés jusqu'à présent. Je vous demande donc de supprimer de votre projet les classes **GestionTestJDBC.java** et **TestJDBC.java** ainsi que la JSP **test_jdbc.jsp**, car nous n'allons pas les réutiliser.

Modification de la table Utilisateur

En ce qui concerne la BDD, nous allons conserver la base de données **bdd_sdzee** et le compte **java** que nous y avons créé, mais allons modifier la table d'exemple **Utilisateur**. Plus précisément, nous allons modifier le type du champ **mot_de_passe**.

 Pourquoi ? Qu'est-ce qui ne va pas avec le type utilisé dans notre bac à sable ?

Je ne vous en ai volontairement pas parlé jusqu'ici pour ne pas compliquer votre apprentissage de JDBC, mais maintenant que vous êtes à l'aise, discutons-en ! 😊

Dans l'exemple du chapitre précédent, j'ai choisi de déléguer le chiffrement du mot de passe de l'utilisateur à la fonction MD5 () de MySQL afin de ne pas encombrer inutilement le code. Seulement c'est une mauvaise pratique, notamment parce que :

- d'une part, c'est une fonction propre à MySQL. Si vous optez pour un autre SGBD, voire pour un autre système de stockage, vous n'êtes pas certains de pouvoir lui trouver un équivalent ;
- d'autre part, et c'est là le point le plus important, c'est un algorithme rapide et peu sûr qu'il est déconseillé d'utiliser pour la sécurisation de mots de passe.

Je ne vais pas vous faire un cours de cryptographie, vous trouverez à ce sujet d'excellentes bases sur le Site du Zéro et plus largement sur le web. Sachez simplement que pour sécuriser de manière efficace les mots de passe de vos utilisateurs, il est préférable aujourd'hui :

- d'utiliser un algorithme de *hashage* relativement fort et lent. Typiquement, une des variantes de SHA-2 (SHA-224, SHA-256, SHA-384 ou SHA-512) est un excellent choix ;
- d'associer au mot de passe un « grain de sel » aléatoire et suffisamment long : 1) pour faire en sorte que deux mots de passe identiques aient une empreinte différente ; 2) afin d'empêcher un éventuel pirate d'utiliser des structures de données comme les fameuses *rainbow tables* pour déchiffrer rapidement une empreinte ;
- d'itérer récursivement la fonction de *hashage* un très grand nombre de fois, mais surtout pas uniquement sur l'empreinte résultant du *hashage* précédent afin de ne pas augmenter les risques de collisions, en veillant bien à réintroduire à chaque itération des données comme le sel et le mot de passe d'origine, avec pour objectif de rendre le travail d'un éventuel pirate autant de fois plus lent.

Bref, vous l'aurez compris, pour un débutant c'est presque mission impossible de protéger efficacement les mots de passe de ses utilisateurs ! Heureusement, il existe des solutions simplifiant grandement cette opération, et notre choix va se porter sur la bibliothèque Jasypt. En quelques mots, il s'agit d'une surcouche aux API de cryptographie existant nativement dans Java, qui fournit des objets et méthodes très faciles d'accès afin de chiffrer des données.



Très bien, mais en quoi cela va-t-il impacter le type de notre champ **mot_de_passe** dans notre table **Utilisateur** ?

Eh bien comme je vous l'ai dit, nous n'allons plus utiliser l'algorithme MD5, mais lui préférer cette fois SHA-256. Le premier génératit des empreintes longues de 32 caractères, voilà pourquoi nous les stockions dans un champ SQL de taille 32. Le second génère des empreintes longues de 64 caractères, que Jasypt encode pour finir à l'aide de Base64 en chaînes longues de 56 caractères. Celles-ci ne vont donc pas rentrer dans le champ **mot_de_passe** que nous avions défini, et nous devons donc le modifier via la commande suivante à exécuter depuis l'invite de commandes de votre serveur MySQL :

Code : SQL - Modification du type du champ mot_de_passe dans la table des utilisateurs

```
ALTER TABLE Utilisateur CHANGE mot_de_passe mot_de_passe CHAR(56)
NOT NULL;
```

Bien entendu, avant d'effectuer cette modification, vous n'oublierez pas de vous positionner sur la bonne base en exécutant la commande `USE bdd_sdzee;` après connexion à votre serveur MySQL.

Reprise du bean Utilisateur

Nous disposons déjà d'un bean **Utilisateur**, sur lequel nous avons travaillé jusqu'à présent, mais celui-ci ne contient pour l'instant qu'une adresse mail, un nom et un mot de passe. Afin qu'il devienne utilisable pour effectuer la correspondance avec les données stockées dans notre table **Utilisateur**, il nous faut réaliser quelques ajouts :

Code : Java - com.sdzee.beans.Utilisateur

```

package com.sdzee.beans;

import java.sql.Timestamp;

public class Utilisateur {

    private Long id;
    private String email;
    private String motDePasse;
    private String nom;
    private Timestamp dateInscription;

    public Long getId() {
        return id;
    }

    public void setId( Long id ) {
        this.id = id;
    }

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

    public String getEmail() {
        return email;
    }

    public void setMotDePasse( String motDePasse ) {
        this.motDePasse = motDePasse;
    }

    public String getMotDePasse() {
        return motDePasse;
    }

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

    public String getNom() {
        return nom;
    }

    public Timestamp getDateInscription() {
        return dateInscription;
    }

    public void setDateInscription( Timestamp dateInscription ) {
        this.dateInscription = dateInscription;
    }
}

```

C'est simple, nous avons simplement créé deux nouvelles propriétés **id** et **dateInscription** stockant logiquement l'**id** et la date d'inscription, et nous disposons maintenant d'un bean qui représente parfaitement une ligne de notre table **Utilisateur**.



Pourquoi avoir utilisé un objet `Long` pour stocker l'**id** ?

En effet, nous aurions pu a priori nous contenter d'un type primitif `long`. Seulement dans une base de données les valeurs peuvent être initialisées à `NULL`, alors qu'un type primitif en Java ne peut pas valoir `null`. Voilà pourquoi il est déconseillé de travailler directement avec les types primitifs, et de leur préférer les objets « enveloppeurs » (les fameux *Wrapper*) : ceux-ci peuvent en effet être initialisés à `null`.



En l'occurrence, les champs de notre table SQL se voient tous appliquer une contrainte `NOT NULL`, il n'y a donc pas de risque de valeurs nulles. Cependant, c'est une bonne pratique de toujours utiliser les objets *Wrapper* dans les beans dont les propriétés correspondent à des champs d'une base de données, afin d'éviter ce genre d'erreurs !

Maintenant que nous avons créé les différentes représentations d'un utilisateur dans notre application, nous allons pouvoir nous attaquer au fameux cloisonnement de la couche de données, autrement dit à la création à proprement parler de notre DAO.

Création des exceptions du DAO

Afin de cacher la nature du mode de stockage des données au reste de l'application, c'est une bonne pratique de masquer les exceptions spécifiques (celles qui surviennent au *runtime*, c'est-à-dire lors de l'exécution) derrière des exceptions propres au DAO. Je m'explique. Typiquement, nous allons dans notre application avoir besoin de gérer deux types d'exceptions concernant les données :

- celles qui sont liées à la configuration du DAO et du driver JDBC ;
- celles qui sont liées à l'interaction avec la base de données.

Dans la couche modèle actuelle de notre système d'inscription, nous nous apprêtons à introduire le stockage des données. Puisque nous avons décidé de suivre le modèle de conception DAO, nous n'allons pas réaliser les manipulations sur la base de données directement depuis les traitements métier, nous allons appeler des méthodes de notre DAO, qui à leur tour la manipuleront. Nous obtiendrons ainsi un modèle divisé en deux sous-couches : une couche métier et une couche de données.

Seulement vous vous en doutez, lors d'une tentative de lecture ou d'écriture dans la base de données, il peut survenir de nombreux types d'incidents : des soucis de connexions, des requêtes incorrectes, des données absentes, la base qui ne répond plus, etc. Et à chacune de ces erreurs correspond une exception SQL ou JDBC particulière. Eh bien notre objectif ici, c'est de faire en sorte que depuis l'extérieur de la couche de données, aucune de ces exceptions ne sorte directement sous cette forme.

Pour ce faire, c'est extrêmement simple, il nous suffit de créer une exception personnalisée qui va encapsuler les exceptions liées à SQL ou JDBC. Voici donc le code de nos deux nouvelles exceptions :

Code : Java - com.sdzee.dao.DAOException

```
package com.sdzee.dao;

public class DAOException extends RuntimeException {
    /*
     * Constructeurs
     */
    public DAOException( String message ) {
        super( message );
    }

    public DAOException( String message, Throwable cause ) {
        super( message, cause );
    }

    public DAOException( Throwable cause ) {
        super( cause );
    }
}
```

Code : Java - com.sdzee.dao.DAOConfigurationException

```
package com.sdzee.dao;

public class DAOConfigurationException extends RuntimeException {
    /*
     * Constructeurs
     */
    public DAOConfigurationException( String message ) {
        super( message );
    }

    public DAOConfigurationException( String message, Throwable
cause ) {
        super( message, cause );
    }
}
```

```

public DAOConfigurationException( Throwable cause ) {
    super( cause );
}
}

```

Comme vous pouvez le constater, il s'agit uniquement de classes héritant de `RuntimeException`, qui se contentent de redéfinir les constructeurs. Ne vous inquiétez pas si vous ne saisissez pas encore bien pourquoi nous avons besoin de ces deux exceptions, vous allez très vite vous en rendre compte dans les codes qui suivent !

Création d'un fichier de configuration

Dans le chapitre précédent, nous avions directement stocké en dur l'adresse et les identifiants de connexion à notre base de données, dans le code Java de notre objet. C'était pratique pour l'exemple, mais vous vous doutez bien que nous n'allons pas procéder de manière aussi brute dans une vraie application.

Afin de séparer les informations de configuration du reste de l'application, il est recommandé de les placer dans un endroit accessible par le code Java et aisément modifiable à la main. Une bonne pratique très courante dans ce genre de cas est la mise en place d'un fichier `properties`, qui n'est rien d'autre qu'un fichier texte dont les lignes respectent un certain format.

Nous allons donc créer un fichier nommé `dao.properties`, que nous allons placer dans le package `com.sdzee.dao` :

Code : Fichier properties - `dao.properties`

```

url = jdbc:mysql://localhost:3306/bdd_sdzee
driver = com.mysql.jdbc.Driver
nomutilisateur = java
motdepasse = $dZ_£E

```

Vous retrouvez dans ce fichier les informations de connexion à notre base. Chacune d'elles est associée à une **clé**, représentée par la chaîne de caractères placée à gauche du signe égal sur chacune des lignes.



Puisque ce fichier contient des informations confidentielles, comme le mot de passe de connexion à la base de données, il est hors de question de le placer dans un répertoire public de l'application ; il est impératif de le placer sous `/WEB-INF` ou un de ses sous-répertoires ! Si vous le mettiez par exemple directement à la racine, sous le répertoire `WebContent` d'Eclipse, alors ce fichier deviendrait accessible depuis le navigateur du client...



Dans une vraie application, les fichiers de configuration ne sont pas stockés au sein de l'application web, mais en dehors. Il devient ainsi très facile d'y modifier une valeur sans avoir à naviguer dans le contenu de l'application. En outre, mais nous étudierons cela en annexe de ce cours, cela permet de ne pas avoir besoin de redéployer l'application à chaque modification, et de se contenter de simplement redémarrer le serveur. Toutefois dans notre cas, nous n'en sommes qu'au stade de l'apprentissage et n'allons donc pas nous embêter avec ces considérations : placer le fichier sous `/WEB-INF` nous convient très bien !

Création d'une Factory

Nous arrivons maintenant à une étape un peu plus délicate de la mise en place de notre couche de données. Il nous faut créer la **Factory** qui va être en charge de l'instanciation des différents DAO de notre application. Alors certes, pour le moment nous n'avons qu'une seule table en base et donc un seul DAO à mettre en place. Mais rappelez-vous : l'important est de créer un code facile à maintenir et à faire évoluer. En outre, maintenant que nous avons placé nos informations de connexion dans un fichier à part, cette **Factory** va être responsable de :

- lire les informations de configuration depuis le fichier `properties` ;
- charger le driver JDBC du SGBD utilisé ;
- fournir une connexion à la base de données.

Je vous donne le code, relativement bref, et je vous commente le tout ensuite :

Code : Java - com.sdzee.dao.DAOFactory

```
package com.sdzee.dao;

import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStream;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.Properties;

public class DAOFactory {

    private static final String FICHIER_PROPERTIES      =
"/com/sdzee/dao/dao.properties";
    private static final String PROPERTY_URL             =
"url";
    private static final String PROPERTY_DRIVER          =
"driver";
    private static final String PROPERTY_NOM_UTILISATEUR =
"nomutilisateur";
    private static final String PROPERTY_MOT_DE_PASSE     =
"motdepassé";

    private String           url;
    private String           username;
    private String           password;

    DAOFactory( String url, String username, String password ) {
        this.url = url;
        this.username = username;
        this.password = password;
    }

    /*
     * Méthode chargée de récupérer les informations de connexion à la
     * base de
     * données, charger le driver JDBC et retourner une instance de la
     * Factory
     */
    public static DAOFactory getInstance() throws
DAOConfigurationException {
        Properties properties = new Properties();
        String url;
        String driver;
        String nomUtilisateur;
        String motDePasse;

        ClassLoader classLoader =
Thread.currentThread().getContextClassLoader();
        InputStream fichierProperties =
classLoader.getResourceAsStream( FICHIER_PROPERTIES );

        if ( fichierProperties == null ) {
            throw new DAOConfigurationException( "Le fichier
properties " + FICHIER_PROPERTIES + " est introuvable." );
        }

        try {
            properties.load( fichierProperties );
            url = properties.getProperty( PROPERTY_URL );
            driver = properties.getProperty( PROPERTY_DRIVER );
            nomUtilisateur = properties.getProperty(
PROPERTY_NOM_UTILISATEUR );
            motDePasse = properties.getProperty(
PROPERTY_MOT_DE_PASSE );
        } catch ( IOException e ) {
            throw new DAOConfigurationException( "Impossible de
charger le fichier properties " + FICHIER_PROPERTIES, e );
        }
    }
}
```

```
}

try {
    Class.forName( driver );
} catch ( ClassNotFoundException e ) {
    throw new DAOConfigurationException( "Le driver est
introuvable dans le classpath.", e );
}

DAOFactory instance = new DAOFactory( url, nomUtilisateur,
motDePasse );
return instance;
}

/* Méthode chargée de fournir une connexion à la base de
données */
/* package */ Connection getConnection() throws SQLException {
    return DriverManager.getConnection( url, username, password
);
}
}

/*
* Méthodes de récupération de l'implémentation des différents DAO
(un seul
* pour le moment)
*/
public UtilisateurDao getUtilisateurDao() {
    return new UtilisateurDaoImpl( this );
}
}
```

Pour commencer, analysons la méthode `getInstance()` qui, comme son nom l'indique, a pour principal objectif d'instancier la classe **DAOFactory**. En premier lieu, remarquez qu'il s'agit là d'une méthode statique. Il n'y a ici rien de compliqué, ça tient tout simplement du bon sens : si nous n'avions pas déclaré cette méthode **static**, alors nous ne pourrions pas l'appeler avant d'avoir instancié la classe **DAOFactory**... Alors que c'est justement là le but de la méthode !



Pourquoi avoir utilisé une telle méthode statique, et pas simplement un constructeur public ?

En effet, la plupart du temps lorsque vous souhaitez créer un objet, vous vous contentez d'appeler un de ses constructeurs publics. Typiquement, pour créer un objet de type A, vous écrivez `A monObjet = new A();`. La limite de cette technique, c'est qu'à chaque appel un nouvel objet va être créé quoi qu'il arrive.

Dans notre cas, ce n'est pas ce que nous souhaitons ! Je vous l'ai déjà expliqué, nous voulons instancier notre **DAOFactory** uniquement sous certaines conditions :

- si le fichier **dao.properties** est accessible ;
 - si les données qu'il contient sont valides ;
 - si le driver JDBC est bien présent dans l'application.

Voilà pourquoi nous utilisons une méthode statique et pas un simple constructeur. Pour information, sachez que c'est une pratique courante dans énormément de projets et de bibliothèques. Voilà tout pour cet aparté, revenons à notre analyse du code.

À la ligne 34, nous initialisons un objet `Properties` qui, comme son nom l'indique, va nous permettre de gérer notre fichier de configuration. Ensuite, nous procédons à l'ouverture du fichier `dao.properties` aux lignes 40 et 41.



Qu'est-ce que c'est que cet objet `ClassLoader` et cette méthode `getResourcesAsStream()` ? Pourquoi ne pas avoir utilisé un traditionnel `FileInputStream` pour ouvrir notre fichier `properties` ?

En effet, nous aurions pu confier cette tâche à un simple `FileInputStream`, seulement il nous aurait alors fallu :

- donner le chemin complet vers le fichier dans le système, pour procéder à l'ouverture du flux ;
 - gérer une éventuelle FileNotFoundException dans un bloc **catch**, et procéder à la fermeture du flux dans un bloc **finally**.

En lieu et place de cette technique, j'utilise ici une méthode plus sophistiquée à première vue mais qui en réalité est très simple : elle consiste à appeler la méthode `getResourceAsStream()` de l'objet `ClassLoader`, qui se charge pour nous d'ouvrir le flux demandé et de retourner `null` en cas d'erreur. Comme vous pouvez l'observer dans le code, nous récupérons le `ClassLoader` depuis le *thread* courant grâce à la méthode `getContextClassLoader()`.

Nous aurions pu écrire le tout en une ligne, mais la décomposition en deux actions distinctes vous permet de comprendre plus facilement de quoi il s'agit. Si vous n'êtes pas familiers avec les objets manipulés ici, qui font partie de la plate-forme Java SE, la Javadoc est là pour vous renseigner : n'hésitez pas à parcourir les liens que je vous donne !



Une fois l'appel à la méthode `getResourceAsStream()` réalisé, nous vérifions à la ligne 43 si elle a retourné `null`, ce qui signifierait alors que le flux n'a pas pu être ouvert. Si tel est le cas, nous envoyons une exception personnalisée `DAOConfigurationException` stipulant que le fichier n'a pas été trouvé.

Nous procédons ensuite au chargement des propriétés contenues dans le fichier à la ligne 48, puis à leur lecture aux lignes 49 à 52. Je vous laisse parcourir les documentations des méthodes `Properties.load()` et `Properties.getProperty()` si vous n'êtes pas à l'aise avec ces manipulations. Après tout, ceci n'est pas l'objet de ce cours et vous devriez déjà connaître cette fonctionnalité si vous avez correctement étudié Java SE !



Observez alors attentivement ce que nous effectuons au niveau du bloc `catch` : nous interceptons l'exception éventuellement envoyée en cas d'erreur lors du chargement des propriétés (format du fichier *properties* incorrect), et l'encapsulons dans une exception `DAOConfigurationException`. Vous devez ici commencer à mieux comprendre pourquoi nous avons pris la peine de créer des exceptions personnalisées, mais ne vous inquiétez pas si vous n'avez pas encore bien saisi, nous y reviendrons plus tard.

Une fois les informations lues avec succès, nous tentons de charger le driver JDBC dont le nom est précisé dans le fichier `dao.properties`, en l'occurrence dans notre cas il s'agit du driver pour MySQL. Le principe est le même que dans le chapitre précédent, nous effectuons un simple appel à `Class.forName()`. Là encore, observez l'encapsulation de l'exception envoyée en cas d'erreur (driver introuvable) dans une exception personnalisée de type `DAOConfigurationException`.

Pour en terminer avec cette méthode `getInstance()`, en cas de succès des étapes précédentes nous instancions la `DAOFactory` en faisant appel au constructeur défini aux lignes 23 à 27.

Nous créons ensuite la méthode `getConnection()` chargée de fournir une connexion à la base de données, aux lignes 68 à 70. Là encore pas de surprise, nous utilisons le même principe que dans le chapitre précédent, à savoir un appel à `DriverManager.getConnection()`.



Par ailleurs, vous avez là un bel exemple de l'utilisation d'une méthode statique à la place d'un constructeur public. En effet, nous ne récupérons pas un objet de type `Connection` en appelant un constructeur public, mais en faisant appel à une méthode statique de l'objet `DriverManager` !

Enfin, nous devons écrire les *getters* retournant les différentes implémentations de DAO contenues dans notre application, ce qui, ne l'oublions pas est le rôle d'origine de notre `DAOFactory`. Comme je vous l'ai déjà précisé, pour le moment nous ne travaillons que sur une seule table et n'allons donc créer qu'un seul DAO ; voilà pourquoi nous n'avons qu'un seul *getter* à mettre en place, aux lignes 76 à 78.



Ne vous préoccuez pas du code de ce *getter* pour le moment, j'y reviendrai après vous avoir expliqué les interfaces et implémentations du DAO Utilisateur. De même, ne vous inquiétez pas si le code ne compile pas encore, il compilera lorsque nous aurons créé les classes mises en jeu !

Création de l'interface du DAO Utilisateur

Avant de créer une implémentation de notre DAO Utilisateur, il nous faut écrire le contrat qui va définir toutes les actions qui devront être effectuées sur les données, c'est-à-dire sur le bean `Utilisateur`. Qui dit contrat dit interface, et c'est donc une interface que nous allons mettre en place.

En ce qui concerne le nommage des classes et interfaces, il existe différentes bonnes pratiques. Pour ma part, je nomme `AbcDao`

l'interface d'un DAO correspondant à la table Abc, et **AbcDaoImpl** son implémentation. Je vous conseille de suivre cette règle également, cela permet de s'y retrouver rapidement dans une application qui contient beaucoup de DAO différents ! Nous créons ici un DAO correspondant à la table Utilisateur, nous allons donc créer une interface nommée **UtilisateurDao** :

Code : Java - com.sdzee.dao.UtilisateurDao

```
package com.sdzee.dao;

import com.sdzee.beans.Utilisateur;

public interface UtilisateurDao {
    void creer( Utilisateur utilisateur ) throws DAOException;
    Utilisateur trouver( String email ) throws DAOException;
}
```

Vous retrouvez ici les deux méthodes mises en jeu par nos formulaires d'inscription et de connexion :

- la création d'un utilisateur, lors de son inscription ;
- la recherche d'un utilisateur, lors de la connexion.

Ces méthodes font partie du fameux CRUD dont je vous ai parlé auparavant ! Notre système se limitant à ces deux fonctionnalités, nous n'allons volontairement pas implémenter les méthodes de mise à jour et de suppression d'un utilisateur. Dans une vraie application toutefois, il va de soi que l'interface de chaque DAO se doit d'être complète.

Petite information au passage : en Java, les méthodes d'une interface sont obligatoirement publiques et abstraites, inutile donc de préciser les mots-clés **public** et **abstract** dans leurs signatures. L'écriture reste permise, mais elle est déconseillée dans les **spécifications Java SE** publiées par Oracle, dans le chapitre concernant les interfaces.

Création de l'implémentation du DAO

Nous voilà arrivés à l'étape finale : la création de l'implémentation de notre DAO Utilisateur. Il s'agit de la classe qui va manipuler la table **Utilisateur** de notre base de données. C'est donc elle qui va contenir le code des méthodes `creer()` et `trouver()` définies dans le contrat que nous venons tout juste de mettre en place.

La première chose à faire, c'est de créer une classe qui implémente l'interface **UtilisateurDao**. Nous allons bien entendu suivre la convention de nommage que nous avons adoptée, et nommer cette implémentation **UtilisateurDaoImpl** :

Code : Java - com.sdzee.dao.UtilisateurDaoImpl

```
public class UtilisateurDaoImpl implements UtilisateurDao {
    /* Implémentation de la méthode trouver() définie dans
    l'interface UtilisateurDao */
    @Override
    public Utilisateur trouver( String email ) throws DAOException {
        return null;
    }

    /* Implémentation de la méthode creer() définie dans
    l'interface UtilisateurDao */
    @Override
    public void creer( Utilisateur utilisateur ) throws
IllegalArgumentException, DAOException {
    }
}
```

Vous le savez très bien, en Java lorsqu'une classe implémente une interface, elle doit impérativement définir toutes les méthodes qui y sont décrites. En l'occurrence notre interface **UtilisateurDao** contient deux méthodes `creer()` et `trouver()`, voilà pourquoi nous devons écrire leur code dans notre implémentation.

Nous devons maintenant réfléchir un peu. Dans l'architecture que nous sommes en train de construire, nous avons mis en place une *Factory*. En plus d'assurer sa fonction principale, c'est-à-dire créer le DAO via le *getter* `getUtilisateurDao()`, elle joue le rôle de pierre angulaire : c'est par son intermédiaire que le DAO va pouvoir acquérir une connexion à la base de données, en appelant sa méthode `getConnection()`.



Pour pouvoir appeler cette méthode et ainsi récupérer une connexion, le DAO doit donc avoir accès à une instance de la **DAOFactory**. Comment faire ?

C'est simple, nous avons déjà préparé le terrain à la ligne 79 du code de notre **DAOFactory** ! Si vous regardez bien, nous passons une instance de la classe (via le mot-clé **this**) à l'implémentation du DAO lors de sa construction. Ainsi, il nous suffit de créer un constructeur qui prend en argument un objet de type **DAOFactory** dans notre DAO :

Code : Java - com.sdzee.dao.UtilisateurDaoImpl

```
public class UtilisateurDaoImpl implements UtilisateurDao {
    private DAOFactory daoFactory;
    UtilisateurDaoImpl( DAOFactory daoFactory ) {
        this.daoFactory = daoFactory;
    }
    ...
}
```

Jusque-là, rien de bien compliqué. Mais ne vous endormez surtout pas, le gros du travail reste à faire ! Nous devons maintenant écrire le code des méthodes `creer()` et `trouver()`, autrement dit le code à travers lequel nous allons communiquer avec notre base de données ! Vous connaissez déjà le principe : connexion, requête préparée, etc. ! 😊

Plutôt que d'écrire bêtement du code à la chaîne en recopiant et adaptant ce que nous avions écrit dans le chapitre précédent, nous allons étudier les besoins de nos méthodes et réfléchir à ce qu'il est possible de factoriser dans des méthodes utilitaires. Mieux encore, nous n'allons pas nous concentrer sur ces deux méthodes en particulier, mais réfléchir plus globalement à ce que toute méthode communiquant avec une base de données doit faire intervenir. Nous pouvons d'ores et déjà noter :

- initialiser une requête préparée avec des paramètres ;
- récupérer une ligne d'une table et enregistrer son contenu dans un bean ;
- fermer proprement les ressources ouvertes (`Connection`, `PreparedStatement`, `ResultSet`).

Pour accomplir ces différentes tâches, a priori nous allons avoir besoin de trois méthodes utilitaires :

1. une qui récupère une liste de paramètres et les ajoute à une requête préparée donnée ;
2. une qui récupère un `ResultSet` et enregistre ses données dans un bean ;
3. une qui ferme toutes les ressources ouvertes.

Étudions tout cela étape par étape.

1. Initialisation d'une requête préparée

Nous avons besoin d'une méthode qui va prendre en argument une connexion, une requête SQL et une liste d'objets, et s'en servir pour initialiser une requête préparée via un appel à `connexion.prepareStatement()`. À ce propos, rappelez-vous de ce que nous avons découvert dans le chapitre précédent : c'est lors de cet appel qu'il faut préciser si la requête doit retourner un champ auto-généré ou non ! Le problème, c'est que nous ne pouvons pas savoir à l'avance si notre requête a besoin de retourner cette information ou non. Voilà pourquoi nous allons ajouter un quatrième argument à notre méthode, dont voici le code complet :

Code : Java

```
/*
 * Initialise la requête préparée basée sur la connexion passée en
 * argument,
 * avec la requête SQL et les objets donnés.
 */
```

```

public static PreparedStatement initialisationRequetePreparee(
    Connection connexion, String sql, boolean returnGeneratedKeys,
    Object... objets ) throws SQLException {
    PreparedStatement preparedStatement =
        connexion.prepareStatement( sql, returnGeneratedKeys ?
            Statement.RETURN_GENERATED_KEYS : Statement.NO_GENERATED_KEYS );
    for ( int i = 0; i < objets.length; i++ ) {
        preparedStatement.setObject( i + 1, objets[i] );
    }
    return preparedStatement;
}

```

Observez les quatre arguments passés à la méthode :

- la connexion, dont nous avons besoin pour appeler la méthode `connexion.prepareStatement()` ;
- la requête SQL, que nous passons en argument lors de l'appel pour construire l'objet `PreparedStatement` ;
- un booléen, indiquant s'il faut ou non retourner d'éventuelles valeurs auto-générées. Nous l'utilisons alors directement au sein de l'appel à `connexion.prepareStatement()` grâce à une simple expression ternaire ;
- une succession d'objets... de tailles variables ! Eh oui, là encore nous n'avons aucun moyen d'anticiper et de savoir à l'avance combien de paramètres attend notre requête préparée.



Que signifient ces '...' dans le type du dernier argument de la méthode ?

C'est une notation purement Java, dite ***varargs***. Vous pouvez la voir comme un joker : en déclarant ainsi votre méthode, vous pourrez ensuite l'appeler avec autant de paramètres que vous voulez (tant qu'ils respectent le type déclaré), et l'appel fonctionnera toujours ! La seule différence sera la taille de l'objet passé en tant que quatrième et dernier argument ici. Exemples :

Code : Java - Exemples d'appels avec un nombre variable d'arguments

```

initialisationRequetePreparee( connexion, requeteSQL, true, email );
initialisationRequetePreparee( connexion, requeteSQL, true, email,
    motDePasse );
initialisationRequetePreparee( connexion, requeteSQL, true, email,
    motDePasse, nom );
initialisationRequetePreparee( connexion, requeteSQL, true, email,
    motDePasse, nom, dateInscription );

```

Bien qu'ils ne présentent pas tous le même nombre d'arguments, tous ces appels font bien référence à une seule et même méthode : celle que nous avons précédemment définie, et dont la signature ne mentionne que quatre arguments ! En réalité sous la couverture, le compilateur regroupera lui-même tous les arguments supplémentaires dans un simple tableau. En fin de compte, cette notation ***varargs*** s'apparente en quelque sorte à un tableau implicite.



Dans ce cas, pourquoi ne pas utiliser directement un tableau, en déclarant à la place de `Object...` un argument de type `Object[]` ?

Tout simplement parce que cela compliquerait l'utilisation de la méthode. En effet, si nous devions absolument passer un objet de type tableau, alors il faudrait prendre la peine d'initialiser un tableau avant d'appeler la méthode. Nos exemples précédents deviendraient alors :

Code : Java - Exemples d'appels avec un tableau

```

Object[] objets = { email };
initialisationRequetePreparee( connexion, requeteSQL, true, objets );
objets = { email, motDePasse };
initialisationRequetePreparee( connexion, requeteSQL, true, objets );
objets = { email, motDePasse, nom };
initialisationRequetePreparee( connexion, requeteSQL, true, objets );
objets = { email, motDePasse, nom, dateInscription };
initialisationRequetePreparee( connexion, requeteSQL, true, objets );

```

Observez la différence avec l'exemple précédent : les appels à la méthode sont, cette fois, tous identiques, mais le fait de devoir initialiser des tableaux avant chaque appel est pénible et complique la lecture du code.



Au sujet de la nature de ce tableau implicite, pourquoi le déclarer de type Object ?

Eh bien parce que là encore, nous n'avons aucun moyen d'anticiper et de déterminer à l'avance ce que notre requête attend en guise de paramètres ! Voilà pourquoi nous devons utiliser le type le plus global possible, à savoir `Object`.

D'ailleurs, en conséquence nous n'allons pas pouvoir faire appel aux méthodes `PreparedStatement.setString()`, `PreparedStatement.setInt()`, etc. que nous avions découvertes et utilisées dans le chapitre précédent, puisque nous n'aurons ici aucun moyen simple de savoir de quel type est chaque objet. Heureusement, il existe une méthode `PreparedStatement.setObject()` qui prend en argument un objet de type `Object`, et qui s'occupe ensuite derrière les rideaux d'effectuer la conversion vers le type SQL du paramètre attendu avant d'envoyer la requête à la base de données.

Sympathique, n'est-ce pas ? 😊

Nous y voilà : ce grand aparté sur les `varargs` vous donne toutes les clés pour comprendre ce qui se passe dans la boucle `for` qui conclut notre méthode. Nous y parcourons le tableau implicite des arguments passés à la méthode lors de son appel, et plaçons chacun d'eux dans la requête préparée via un appel à la méthode `PreparedStatement.setObject()`.



En fin de compte, c'est ici une méthode courte que nous avons réalisée, mais qui fait intervenir des concepts intéressants et qui va nous être très utile par la suite. Par ailleurs, notez qu'il s'agit d'une méthode purement utilitaire, que nous pourrons réutiliser telle quelle dans n'importe quel DAO !

2. Mapping d'un ResultSet dans un bean

Abordons maintenant notre seconde méthode utilitaire, qui va nous permettre de faire la correspondance entre une ligne d'un `ResultSet` et un bean. Contrairement à notre précédente méthode, nous n'allons cette fois pas pouvoir totalement découpler la méthode de notre DAO en particulier, puisque nous travaillons directement sur un `ResultSet` issu de notre table d'utilisateurs et sur un bean de type **Utilisateur**. Voici le code :

Code : Java

```
/*
 * Simple méthode utilitaire permettant de faire la correspondance
 * (le
 * mapping) entre une ligne issue de la table des utilisateurs (un
 * ResultSet) et un bean Utilisateur.
 */
private static Utilisateur map( ResultSet resultSet ) throws
SQLException {
    Utilisateur utilisateur = new Utilisateur();
    utilisateur.setId( resultSet.getLong( "id" ) );
    utilisateur.setEmail( resultSet.getString( "email" ) );
    utilisateur.setMotDePasse( resultSet.getString( "mot_de_passe" ) );
    utilisateur.setNom( resultSet.getString( "nom" ) );
    utilisateur.setDateInscription( resultSet.getTimestamp(
"date_inscription" ) );
    return utilisateur;
}
```

Rien de bien compliqué ici : notre méthode prend en argument un `ResultSet` dont le curseur a déjà été correctement positionné, et place chaque champ lu dans la propriété correspondante du nouveau bean créé.

3. Fermeture des ressources

Pour terminer, nous allons gérer proprement la fermeture des différentes ressources qui peuvent intervenir dans une communication avec la base de données. Plutôt que de tout mettre dans une seule et même méthode, nous allons écrire une

méthode pour fermer chaque type de ressource :

Code : Java

```

/* Fermeture silencieuse du resultset */
public static void fermetureSilencieuse( ResultSet resultSet ) {
    if ( resultSet != null ) {
        try {
            resultSet.close();
        } catch ( SQLException e ) {
            System.out.println( "Échec de la fermeture du ResultSet
: " + e.getMessage() );
        }
    }
}

/* Fermeture silencieuse du statement */
public static void fermetureSilencieuse( Statement statement ) {
    if ( statement != null ) {
        try {
            statement.close();
        } catch ( SQLException e ) {
            System.out.println( "Échec de la fermeture du Statement
: " + e.getMessage() );
        }
    }
}

/* Fermeture silencieuse de la connexion */
public static void fermetureSilencieuse( Connection connexion ) {
    if ( connexion != null ) {
        try {
            connexion.close();
        } catch ( SQLException e ) {
            System.out.println( "Échec de la fermeture de la
connexion : " + e.getMessage() );
        }
    }
}

/* Fermetures silencieuses du statement et de la connexion */
public static void fermeturesSilencieuses( Statement statement,
Connection connexion ) {
    fermetureSilencieuse( statement );
    fermetureSilencieuse( connexion );
}

/* Fermetures silencieuses du resultset, du statement et de la
connexion */
public static void fermeturesSilencieuses( ResultSet resultSet,
Statement statement, Connection connexion ) {
    fermetureSilencieuse( resultSet );
    fermetureSilencieuse( statement );
    fermetureSilencieuse( connexion );
}

```

Vous connaissez déjà le principe des trois premières méthodes, puisque nous avons effectué sensiblement la même chose dans le chapitre précédent. Nous créons ensuite deux méthodes qui font appel à tout ou partie des trois premières, tout bonnement parce que certaines requêtes font intervenir un ResultSet, d'autres non ! En outre, notez que tout comme la méthode initialisationRequetePreparee(), ces méthodes sont purement utilitaires et nous pouvons les réutiliser dans n'importe quel DAO.

4. Récapitulons

Les méthodes purement utilitaires peuvent être placées dans une classe à part entière, une classe utilitaire donc, et être utilisées depuis n'importe quel DAO. Voilà pourquoi j'ai donc créé une classe finale que j'ai nommée **DAOUtilitaire** et placée dans le

package com.sdzee.dao. Vous pouvez la mettre en place vous-mêmes, ou bien la télécharger en cliquant sur [ce lien](#) et l'ajouter à votre projet.

En ce qui concerne la méthode `map()`, celle-ci doit simplement être placée dans la classe **UtilisateurDaoImpl**.

La méthode `trouver()`

Maintenant que nos méthodes utilitaires sont prêtes, nous pouvons finalement écrire le code de la méthode `trouver()` :

Code : Java - com.sdzee.dao.UtilisateurDaoImpl

```
private static final String SQL_SELECT_PAR_EMAIL = "SELECT id, email, nom, mot_de_passe, date_inscription FROM Utilisateur WHERE email = ?";

/* Implémentation de la méthode définie dans l'interface UtilisateurDao */
@Override
private Utilisateur trouver( String email ) throws DAOException {
    Connection connexion = null;
    PreparedStatement preparedStatement = null;
    ResultSet resultSet = null;
    Utilisateur utilisateur = null;

    try {
        /* Récupération d'une connexion depuis la Factory */
        connexion = daoFactory.getConnection();
        preparedStatement = initialisationRequetePreparee( connexion,
            SQL_SELECT_PAR_EMAIL, false, email );
        resultSet = preparedStatement.executeQuery();
        /* Parcours de la ligne de données de l'éventuel ResultSet retourné */
        if ( resultSet.next() ) {
            utilisateur = map( resultSet );
        }
    } catch ( SQLException e ) {
        throw new DAOException( e );
    } finally {
        fermeturesSilencieuses( resultSet, preparedStatement, connexion );
    }

    return utilisateur;
}
```

Vous retrouvez ici les principes découverts et appliqués dans le chapitre précédent : l'obtention d'une connexion, la préparation d'une requête de lecture, son exécution, puis la récupération et l'analyse du `ResultSet` retourné, et enfin la fermeture des ressources mises en jeu.

Les trois lignes surlignées font intervenir chacune des méthodes utilitaires que nous avons développées. À ce propos, vous n'oublierez pas d'importer le contenu de la classe **DAOUtilitaire** afin de rendre disponibles ses méthodes ! Pour ce faire, il vous suffit d'ajouter la ligne suivante dans les imports de votre classe **UtilisateurDaoImpl** :

Code : Java - Import des méthodes utilitaires

```
import static com.sdzee.dao.DAOUtilitaire.*;
```

En précisant directement dans l'import le mot-clé **static**, vous pourrez appeler vos méthodes comme si elles faisaient directement partie de votre classe courante. Par exemple, vous n'aurez pas besoin d'écrire `DAOUtilitaire.fermeturesSilencieuses(...)`, mais simplement `fermeturesSilencieuses(...)`. Pratique, n'est-ce pas ? 😊

En outre, vous retrouvez également dans ce code la bonne pratique que je vous ai enseignée en début de cours, à savoir la mise

en place d'une constante. Elle détient en l'occurrence l'instruction SQL utilisée pour préparer notre requête.

La méthode creer()

Dernière pierre à notre édifice, nous devons écrire le code de la méthode `creer()` :

Code : Java - com.sdzee.dao.UtilisateurDaoImpl

```

private static final String SQL_INSERT = "INSERT INTO Utilisateur
(email, mot_de_passe, nom, date_inscription) VALUES (?, ?, ?, ?
NOW())";

/* Implémentation de la méthode définie dans l'interface
UtilisateurDao */
@Override
public void creer( Utilisateur utilisateur ) throws DAOException {
    Connection connexion = null;
    PreparedStatement preparedStatement = null;
    ResultSet valeursAutoGenerees = null;

    try {
        /* Récupération d'une connexion depuis la Factory */
        connexion = daoFactory.getConnection();
        preparedStatement = initialisationRequetePreparee( connexion,
SQL INSERT, true, utilisateur.getEmail(),
utilisateur.getMotDePasse(), utilisateur.getNom() );
        int statut = preparedStatement.executeUpdate();
        /* Analyse du statut retourné par la requête d'insertion */
        if ( statut == 0 ) {
            throw new DAOException( "Échec de la création de
l'utilisateur, aucune ligne ajoutée dans la table." );
        }
        /* Récupération de l'id auto-généré par la requête
d'insertion */
        valeursAutoGenerees = preparedStatement.getGeneratedKeys();
        if ( valeursAutoGenerees.next() ) {
            /* Puis initialisation de la propriété id du bean
Utilisateur avec sa valeur */
            utilisateur.setId( valeursAutoGenerees.getLong( 1 ) );
        } else {
            throw new DAOException( "Échec de la création de
l'utilisateur en base, aucun ID auto-généré retourné." );
        }
    } catch ( SQLException e ) {
        throw new DAOException( e );
    } finally {
        fermeturesSilencieuses( valeursAutoGenerees, preparedStatement,
connexion );
    }
}

```

Là encore, vous retrouvez les principes découverts jusqu'à présent : l'obtention d'une connexion, la préparation d'une requête d'insertion avec demande de renvoi de l'id auto-généré grâce au booléen à `true` passé à notre méthode utilitaire `DAOUtilitaire.initialisationRequetePreparee()`, son exécution et la récupération de son statut, la récupération de l'id auto-généré via l'appel à la méthode `PreparedStatement.getGeneratedKeys()`, et enfin la fermeture des ressources mises en jeu.



Dans ces deux méthodes, vous observerez bien l'utilisation de l'exception personnalisée `DAOException` que nous avons créée auparavant. Si vous avez encore un peu de mal à voir où nous voulons en venir avec nos exceptions, ne vous inquiétez pas : vous allez comprendre dans la suite de ce chapitre !

Intégration

Nous allons maintenant tâcher d'intégrer proprement cette nouvelle couche dans notre application. Pour l'exemple, nous allons travailler avec notre système d'inscription et laisser de côté le système de connexion pour le moment.

Chargement de la DAOFactory

Notre **DAOFactory** est un objet que nous ne souhaitons instancier qu'une seule fois, le démarrage de l'application semble donc l'instant approprié pour procéder à son initialisation.



Dans une application Java classique, il nous suffirait de placer quelques lignes de code en tête de la méthode `main()`, et le tour serait joué. Mais dans une application Java EE, comment faire ?

La solution, c'est l'interface `ServletContextListener`. Lisez sa très courte documentation, vous observerez qu'elle fournit une méthode `contextInitialized()` qui est appelée dès le démarrage de l'application, avant le chargement des servlets et filtres du projet. Il s'agit exactement de ce dont nous avons besoin !

Création du Listener

Nous allons donc mettre en place un nouveau package `com.sdzee.config` et y créer une classe nommée `InitialisationDaoFactory` qui implémente l'interface `ServletContextListener` :

Code : Java - `com.sdzee.config.InitialisationDaoFactory`

```
package com.sdzee.config;

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

import com.sdzee.dao.DAOFactory;

public class InitialisationDaoFactory implements
ServletContextListener {
    private static final String ATT_DAO_FACTORY = "daofactory";

    private DAOFactory           daoFactory;

    @Override
    public void contextInitialized( ServletContextEvent event ) {
        /* Récupération du ServletContext lors du chargement de
l'application */
        ServletContext servletContext = event.getServletContext();
        /* Instanciation de notre DAOFactory */
        this.daoFactory = DAOFactory.getInstance();
        /* Enregistrement dans un attribut ayant pour portée toute
l'application */
        servletContext.setAttribute( ATT_DAO_FACTORY,
this.daoFactory );
    }

    @Override
    public void contextDestroyed( ServletContextEvent event ) {
        /* Rien à réaliser lors de la fermeture de l'application...
*/
    }
}
```

Comme d'habitude en Java, puisque nous implementons une interface nous devons définir toutes ses méthodes, en l'occurrence elle en contient deux : une qui est lancée au démarrage de l'application, et une autre à sa fermeture. Vous l'aurez deviné, seule la méthode appelée lors du démarrage nous intéresse ici.

Le code est très simple et très court, comme vous pouvez l'observer. Si vous avez été très assidus, vous devez vous souvenir que le `ServletContext` n'est rien d'autre que l'objet qui agit dans les coulisses de l'objet implicite `application` ! Sachant cela, vous devez maintenant comprendre sans problème ce que nous faisons ici :

- nous récupérons le `ServletContext` à la ligne 16 ;
- nous obtenons une instance de notre `DAOFactory` via un appel à sa méthode statique `DAOFactory.getInstance()` ;
- nous plaçons cette instance dans un attribut du `ServletContext` via sa méthode `setAttribute()`, qui a donc pour portée l'application entière !



Il faut être très prudent lors de la manipulation d'objets de portée **application**. En effet, puisque de tels objets sont partagés par tous les composants durant toute la durée de vie de l'application, il est recommandé de les utiliser uniquement en lecture seule. Autrement dit, il est recommandé de les initialiser au démarrage de l'application et de ne plus jamais les modifier ensuite - que ce soit depuis une servlet, un filtre ou une JSP - afin d'éviter les modifications concurrentes et les problèmes de cohérence que cela pourrait impliquer. Dans notre cas, il n'y a aucun risque : notre objet est bien créé dès le démarrage de l'application, et il sera ensuite uniquement lu.

Configuration du Listener

Pour que notre **Listener** fraîchement créé soit pris en compte lors du démarrage de notre application, il nous faut ajouter une section au fichier `web.xml` :

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

```
<listener>
    <listener-
    class>com.sdzee.config.InitialisationDaoFactory</listener-class>
</listener>
```

Voilà tout ce qu'il est nécessaire d'écrire. N'oubliez pas de redémarrer Tomcat pour que la modification soit prise en compte !

Utilisation depuis la servlet

Notre **DAOFactory** étant prête à l'emploi dans notre projet, nous pouvons maintenant récupérer une instance de notre DAO Utilisateur depuis notre servlet d'inscription.

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.dao.DAOFactory;
import com.sdzee.dao.UtilisateurDao;
import com.sdzee.forms.InscriptionForm;

public class Incription extends HttpServlet {
    public static final String CONF_DAO_FACTORY = "daofactory";
    public static final String ATT_USER = "utilisateur";
    public static final String ATT_FORM = "form";
    public static final String VUE = "/WEB-
INF/incription.jsp";

    private UtilisateurDao utilisateurDao;

    public void init() throws ServletException {
```

```

/* Récupération d'une instance de notre DAO Utilisateur */
this.utilisateurDao = ( DAOFactory )
getServletContext().getAttribute( CONF.DAO_FACTORY )
).getUtilisateurDao();
}

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( utilisateurDao );

    /* Traitement 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 );
}
}

```

La première étape consiste à récupérer une instance de notre **DAOFactory**. Puisque nous en avons placé une dans un attribut de portée application lors de son chargement, il nous suffit depuis notre servlet de récupérer le `ServletContext` et d'appeler sa méthode `getAttribute()`. En réalité, le principe est exactement le même que pour un attribut de requête ou de session, seul l'objet de référence change.

Cette méthode retournant un objet de type `Object`, nous devons ensuite effectuer un *cast* vers le type **DAOFactory** pour pouvoir appeler sa méthode `getUtilisateurDao()`. Nous enregistrons finalement l'instance de notre DAO Utilisateur dans un attribut de notre servlet.



Quelle est cette méthode nommée `init()` ?

Cette méthode fait partie de celles que je ne vous avais volontairement pas expliquées lorsque je vous ai présenté les servlets. Je vous avais d'ailleurs précisé alors, que vous n'étiez pas encore assez à l'aise avec le concept de servlet lui-même, pour avoir besoin d'intervenir sur son cycle de vie. Le temps est venu pour vous d'en savoir un petit peu plus.

Pour rappel, une servlet n'est créée qu'une seule et unique fois par le conteneur, lors du démarrage de l'application ou bien lors de son premier accès (ceci dépendant de la présence ou non d'une section **<load-on-startup>** dans la déclaration de la servlet au sein du fichier `web.xml`). Eh bien sachez que lorsque la servlet est instanciée, cette fameuse méthode `init()` va être appelée, une seule et unique fois donc. Ainsi, vous avez la possibilité à travers cette méthode d'effectuer des tâches uniques, qui ne sont pas destinées à être lancées à chaque appel aux méthodes `doGet()` ou `doPost()` par exemple.



Pourquoi créer notre instance de DAO depuis cette méthode `init()` ?

Eh bien tout simplement parce que si nous en créions une depuis nos méthodes `doXXX()`, une nouvelle instance serait créée à chaque requête reçue ! En l'occurrence, ce n'est pas nécessaire puisque notre DAO est bien écrit : il manipule les ressources dont il a besoin au plus près de la requête SQL à effectuer, et les ferme correctement de la même manière.

Par contre, notre DAO pourrait très bien être construit d'une manière différente et nécessiter alors une nouvelle instance à chaque utilisation. Par exemple, imaginez que notre DAO partage une seule et même connexion à la base de données pour l'ensemble de ses méthodes CRUD. Eh bien dans ce cas, il serait impensable de ne créer qu'une seule instance du DAO et de la partager à tous les clients qui enverraient des requêtes vers la servlet, car cela reviendrait à partager une seule connexion à la BDD pour tous les clients ! 

 Ce type de problématiques porte un nom : il s'agit de la **thread-safety**. En quelques mots, il s'agit de la possibilité ou non de partager des ressources à l'ensemble des *threads* intervenant sur un objet ou une classe. C'est un concept particulièrement important dans le cas d'une application web, parce qu'il y a bien souvent un nombre conséquent de clients utilisant une seule et même application centralisée.



Qu'est-ce qu'un attribut de servlet a de particulier face à un attribut déclaré au sein d'une méthode `doXXX()` ?

La différence est majeure : un attribut de servlet est partagé par tous les *threads* utilisant l'instance de la servlet, autrement dit par tous les clients qui y font appel ! Alors qu'un simple attribut déclaré par exemple dans une méthode `doGet()` n'existe que pour un seul *thread*, et se matérialise ainsi par un objet différent d'un *thread* à l'autre.

Voilà pourquoi dans notre cas, nous enregistrons notre instance de DAO dans un attribut de servlet : il peut tout à fait être partagé par l'ensemble des clients faisant appel à la servlet, puisque je vous le rappelle il est correctement écrit et gère proprement les ressources. L'intérêt est avant tout d'économiser les ressources du serveur : ré-instancier un nouveau DAO à chaque requête reçue serait un beau gâchis en termes de mémoire et de performances !



Bref, vous comprenez maintenant mieux pourquoi je n'ai pas souhaité vous présenter le cycle de vie d'une servlet trop tôt. Il implique des mécanismes qui ne sont pas triviaux, et il est nécessaire de bien comprendre le contexte d'utilisation d'une servlet avant d'intervenir sur son cycle de vie. Là encore, je ne vous dis pas tout et me limite volontairement à cette méthode `init()`. Ne vous inquiétez pas, le reste du mystère sera levé en annexe !

Pour finir, il faut transmettre notre instance DAO Utilisateur à l'objet métier **InscriptionForm**. Eh oui, c'est bien lui qui va gérer l'inscription d'un utilisateur, et c'est donc bien lui qui va en avoir besoin pour communiquer avec la base de données ! Nous lui passons ici à la ligne 35 par l'intermédiaire de son constructeur, qu'il va donc nous falloir modifier par la suite pour que notre code compile...

Reprise de l'objet métier

Comme je viens de vous le dire, la première étape consiste ici à modifier le constructeur de l'objet **InscriptionForm** pour qu'il prenne en compte le DAO transmis. En l'occurrence, nous n'avions jusqu'à présent créé aucun constructeur, et nous étions contentés de celui par défaut. Nous allons donc devoir ajouter le constructeur suivant, ainsi que la variable d'instance associée permettant de stocker le DAO passé en argument :

Code : Java - com.sdzee.forms.InscriptionForm

```
private UtilisateurDao utilisateurDao;
public InscriptionForm( UtilisateurDao utilisateurDao ) {
    this.utilisateurDao = utilisateurDao;
}
```

Le DAO étant maintenant disponible au sein de notre objet, nous allons pouvoir l'utiliser dans notre méthode d'inscription pour appeler la méthode `creer()` définie dans l'interface **UtilisateurDao**. Il faudra alors bien penser à intercepter une éventuelle **DAOException** envoyée par le DAO, et agir en conséquence !

Code : Java - com.sdzee.forms.InscriptionForm

```
public Utilisateur inscrireUtilisateur( HttpServletRequest request )
{
```

```

String email = getValeurChamp( request, CHAMP_EMAIL );
String motDePasse = getValeurChamp( request, CHAMP_PASS );
String confirmation = getValeurChamp( request, CHAMP_CONF );
String nom = getValeurChamp( request, CHAMP_NOM );

Utilisateur utilisateur = new Utilisateur();
try {
    traiterEmail( email, utilisateur );
    traiterMotsDePasse( motDePasse, confirmation, utilisateur );
    traiterNom( nom, utilisateur );

    if ( erreurs.isEmpty() ) {
        utilisateurDao.creer( utilisateur );
        resultat = "Succès de l'inscription.";
    } else {
        resultat = "Échec de l'inscription.";
    }
} catch ( DAOException e ) {
    resultat = "Échec de l'inscription : une erreur imprévue est
survenue, merci de réessayer dans quelques instants.";
    e.printStackTrace();
}

return utilisateur;
}

```

Vous pouvez observer plusieurs choses ici :

- j'appelle la méthode `utilisateurDao.creer()` à la ligne 14, uniquement si aucune erreur de validation n'a eu lieu. En effet, inutile d'aller faire une requête sur la BDD si les critères de validation des champs du formulaire n'ont pas été respectés ;
- il est alors nécessaire de mettre en place un `try/catch` pour gérer une éventuelle **DAOException** retournée par cet appel ! En l'occurrence, j'initialise la chaîne `resultat` avec un message d'échec ;
- enfin, j'ai regroupé le travail de validation des paramètres et d'initialisation des propriétés du bean dans des méthodes `traiterXXX()`. Cela me permet d'aérer le code, qui commençait à devenir sérieusement chargé avec tout cet enchevêtrement de blocs `try/catch`, sans oublier les ajouts que nous devons y apporter !



Quels ajouts ? Qu'est-il nécessaire de modifier dans nos méthodes de validation ?

La gestion de l'adresse mail doit subir quelques modifications, car nous savons dorénavant qu'une adresse doit être unique en base, et le mot de passe doit absolument être chiffré avant d'être envoyé en base. En plus de cela, nous allons mettre en place une exception personnalisée pour la validation des champs afin de rendre notre code plus clair. Voici donc le code des méthodes gérant ces fonctionnalités :

Code : Java - com.sdzee.forms.InscriptionForm

```

private static final String ALGO_CHIFFREMENT = "SHA-256";

...
/*
 * Appel à la validation de l'adresse email reçue et initialisation
 * de la
 * propriété email du bean
 */
private void traiterEmail( String email, Utilisateur utilisateur ) {
    try {
        validationEmail( email );
    } catch ( FormValidationException e ) {
        setErreur( CHAMP_EMAIL, e.getMessage() );
    }
}

```

```

        utilisateur.setEmail( email );
    }

/*
 * Appel à la validation des mots de passe reçus, chiffrement du mot
 * de
 * passe et initialisation de la propriété motDePasse du bean
 */
private void traiterMotsDePasse( String motDePasse, String
confirmation, Utilisateur utilisateur ) {
    try {
        validationMotsDePasse( motDePasse, confirmation );
    } catch ( FormValidationException e ) {
        setErreur( CHAMP_PASS, e.getMessage() );
        setErreur( CHAMP_CONF, null );
    }

    /*
     * Utilisation de la bibliothèque Jasypt pour chiffrer le mot de
     * passe
     * efficacement.
     *
     * L'algorithme SHA-256 est ici utilisé, avec par défaut un salage
     * aléatoire et un grand nombre d'itérations de la fonction de
     * hashage.
     *
     * La String retournée est de longueur 56 et contient le hash en
     * Base64.
     */
    ConfigurablePasswordEncryptor passwordEncryptor = new
ConfigurablePasswordEncryptor();
    passwordEncryptor.setAlgorithm( ALGO_CHIFFREMENT );
    passwordEncryptor.setPlainDigest( false );
    String motDePasseChiffre = passwordEncryptor.encryptPassword(
motDePasse );

    utilisateur.setMotDePasse( motDePasseChiffre );
}

/* Validation de l'adresse email */
private void validationEmail( String email ) throws
FormValidationException {
    if ( email != null ) {
        if ( !email.matches(
"([^.@]+)(\\.[^.@]+)*@[^.@]+\\" ) ) {
            throw new FormValidationException( "Merci de saisir une
adresse mail valide." );
        } else if ( utilisateurDao.trouver( email ) != null ) {
            throw new FormValidationException( "Cette adresse email
est déjà utilisée, merci d'en choisir une autre." );
        }
    } else {
        throw new FormValidationException( "Merci de saisir une
adresse mail." );
    }
}

```

Pour commencer, remarquez la structure des méthodes `traiterXXX()` créées : comme je vous l'ai déjà annoncé, celles-ci renferment simplement l'appel à la méthode de validation d'un champ et l'initialisation de la propriété du bean correspondante.

Ensuite dans la méthode de traitement des mots de passe, j'ai ajouté la procédure de chiffrement à l'aide de la bibliothèque [Jasypt](#) aux lignes 39 à 42. Vous pouvez la télécharger directement en cliquant sur [ce lien](#). De la même manière que pour les bibliothèques que nous avons déjà manipulées dans le passé, il suffit de placer le fichier .jar récupéré sous **/WEB-INF/lib**.

Je vous laisse parcourir sa documentation et découvrir les méthodes et objets disponibles, en l'occurrence le code que je vous donne est suffisamment commenté pour que vous compreniez comment ce que j'ai mis en place fonctionne.

 Il est important de réaliser cette sécurisation du mot de passe en amont, afin d'initialiser la propriété du bean avec l'empreinte ainsi générée et non pas directement avec le mot de passe en clair. Par la suite, lorsque nous souhaiterons vérifier si un utilisateur entre le bon mot de passe lors de sa connexion, il nous suffira de le comparer directement à l'empreinte stockée grâce à la méthode `passwordEncryptor.checkPassword()`. De manière générale, la règle veut qu'on ne travaille jamais directement sur les mots de passe en clair.

Enfin, j'ai ajouté dans la méthode de validation de l'adresse email un appel à la méthode `trouver()` du DAO, afin de renvoyer une exception en cas d'adresse déjà existante. Cela veut dire que nous effectuons deux requêtes sur la base lors d'une inscription utilisateur : une pour vérifier si l'adresse n'existe pas déjà via l'appel à `trouver()`, et une pour l'insérer via l'appel à `creer()`.

 Est-ce bien nécessaire d'effectuer deux requêtes ? Économiser une requête vers la BDD ne serait pas préférable ?

Eh bien oui, c'est bien plus pratique pour nous d'effectuer une vérification avant de procéder à l'insertion. Si nous ne le faisons pas, alors nous devrions nous contenter d'analyser le statut retourné par notre requête `INSERT` et d'envoyer une exception en cas d'un statut valant zéro. Le souci, c'est qu'un échec de l'insertion peut être dû à de nombreuses causes différentes, et pas seulement au fait qu'une adresse email existe déjà en base ! Ainsi en cas d'échec nous ne pourrions pas avertir l'utilisateur que le problème vient de son adresse mail, nous pourrions simplement l'informer que la création de son compte a échoué. Il tenterait alors à nouveau de s'inscrire avec les mêmes coordonnées en pensant que le problème vient de l'application, et recevrait une nouvelle fois la même erreur, etc.

En testant en amont la présence de l'adresse en base via une bête requête de lecture, nous sommes capables de renvoyer une erreur précise à l'utilisateur. Si l'appel à `trouver()` retourne quelque chose, alors cela signifie que son adresse existe déjà : nous envoyons alors une exception contenant un message lui demandant d'en choisir une autre, qui provoque dans `traiterEmail()` la mise en place du message dans la Map des erreurs, et qui ainsi permet de ne pas appeler la méthode `creer()` en cas d'adresse déjà existante.

En outre, chercher à économiser ici une requête SQL relève du détail. Premièrement parce qu'il s'agit ici d'un simple `SELECT` très peu gourmand, et deuxièmement parce qu'à moins de travailler sur un site extrêmement populaire, l'inscription reste une opération effectuée relativement peu fréquemment en comparaison du reste des fonctionnalités. Bref, pas de problème ! 😊

Création d'une exception dédiée aux erreurs de validation

Enfin, vous remarquez que nos méthodes renvoient dorénavant des `FormValidationException`, et non plus de banales `Exception`. J'ai en effet pris soin de créer une nouvelle exception personnalisée, qui hérite d'`Exception` et que j'ai placée dans `com.sdzee.forms` :

Code : Java - `com.sdzee.forms.FormValidationException`

```
package com.sdzee.forms;

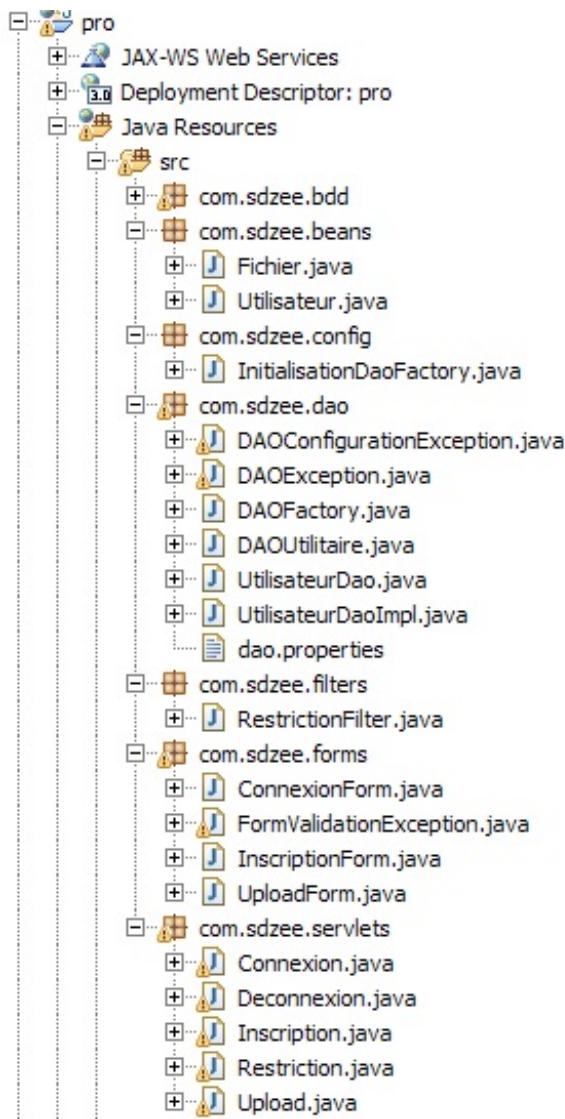
public class FormValidationException extends Exception {
    /*
     * Constructeur
     */
    public FormValidationException( String message ) {
        super( message );
    }
}
```

Ce n'est pas une modification nécessaire en soit, mais cela rend le code plus compréhensible à la lecture, car nous savons ainsi quel type d'erreur est manipulé par une méthode au premier coup d'œil.

Vérifications

Le code final

Avant tout, je vous propose de vérifier que vous avez bien reporté toutes les modifications nécessaires au code existant, et réalisé tous les ajouts demandés. Voici sur la figure suivante l'arborescence que vous êtes censés obtenir.



Certaines classes ayant subi beaucoup de retouches, j'ai dû fractionner leur code en plusieurs sections, afin de vous l'expliquer le plus clairement possible et ainsi de vous en faciliter la compréhension. Ne souhaitant pas vous voir vous emmêler les pinceaux, je vous propose de télécharger les codes complets des deux classes les plus massives :

- [UtilisateurDaoImpl.java](#)
- [InscriptionForm.java](#)

Vous pourrez ainsi vous assurer que vous n'avez omis aucun changement.

Le scénario de tests

Après toutes ces nouveautés, tous ces ajouts et toutes ces modifications, il est temps de tester le bon fonctionnement de notre application. Le scénario est relativement léger, il suffit de tester ce qui se passe lorsqu'un utilisateur tente de s'inscrire deux fois avec la même adresse email. Je vous laisse reprendre les scénarios des chapitres précédents, afin de vous assurer que tout ce que nous avions mis en place dans les chapitres précédents fonctionne toujours.



Pour information, cela s'appelle un **test de régression**. Cela consiste à vérifier que les modifications apportées sur le code n'ont pas modifié le fonctionnement de la page ou de l'application, et c'est bien entendu un processus récurrent et extrêmement important dans le développement d'une application !

Voici à la figure suivante le formulaire avant soumission des données.

Vous pouvez vous inscrire via ce formulaire.

Adresse email *

Mot de passe *

Confirmation du mot de passe *

Nom d'utilisateur

Inscription

Voici à la figure suivante le formulaire après soumission la première fois.

Vous pouvez vous inscrire via ce formulaire.

Adresse email *

Mot de passe *

Confirmation du mot de passe *

Nom d'utilisateur

Inscription

Succès de l'inscription.

Voici à la figure suivante le formulaire après soumission une seconde fois.

Vous pouvez vous inscrire via ce formulaire.

Adresse email * Cette adresse email est déjà utilisée, merci d'en choisir une autre.

Mot de passe *

Confirmation du mot de passe *

Nom d'utilisateur

Inscription

Échec de l'inscription.

Comme prévu, la méthode de validation de l'adresse email a correctement fait son travail. Elle a reçu un résultat lors de l'appel à la méthode `trouver()`, et a par conséquent placé un message d'erreur sur le champ `email`.

En outre, vous pouvez regarder le contenu de votre table **Utilisateur** depuis la console de votre serveur MySQL, via un simple `SELECT * FROM Utilisateur;`. Vous constaterez alors que les informations que vous avez saisies depuis votre navigateur ont bien été enregistrées dans la base de données, et que le mot de passe a bien été chiffré.

Voilà tout ce qu'il est nécessaire de vérifier pour le moment. En apparence, cela paraît peu par rapport à tout le code que nous avons dû mettre en place, je vous l'accorde. Mais ne vous en faites pas, nous n'avons pas uniquement rendu notre premier DAO opérationnel, nous avons préparé le terrain pour tous les futurs DAO de notre application ! 😊

- Le pattern DAO permet d'isoler l'accès aux données et leur stockage du reste de l'application.
- Mettre en place des exceptions spécifiques au DAO permet de masquer le type du stockage sous-jacent.
- Mettre en place une Factory initialisée au démarrage de l'application via un `ServletContextListener` permet de ne charger qu'une seule et unique fois le driver JDBC.
- Mettre en place des utilitaires pour la préparation des requêtes et la libération des ressources permet de limiter la duplication et d'alléger grandement le code.
- Notre DAO libérant proprement les ressources qu'il met en jeu, une seule et unique instance de notre DAO peut être partagée par toutes les requêtes entrantes.
- Récupérer une telle instance de DAO de manière unique depuis une servlet, et non pas à chaque requête entrante, peut se faire simplement en utilisant sa méthode `init()`.
- Notre servlet transmet alors simplement l'instance du DAO à l'objet métier, qui sera responsable de la gestion des données.
- Notre objet métier ne connaît pas le système de stockage final utilisé : il ne fait qu'appeler les méthodes définies dans l'interface de notre DAO.

TP Fil rouge - Étape 6

Sixième étape du fil rouge, le temps est enfin venu pour vous d'intégrer une base de données à votre application ! Création de la base, des tables, des DAO associés et intégration dans l'application existante vous attendent.

Objectifs

Fonctionnalités

Vous allez mettre en place une base de données dans votre application. Ce changement va impliquer que :

- lors de la création d'un client ou d'une commande par l'utilisateur, les données ne seront maintenant plus seulement mises en session, mais également enregistrées dans la base ;
 - vous allez rendre possible la création de clients portant le même nom, et de commandes passées à la même date ;
 - les pages listant les clients et commandes existants continueront à se baser uniquement sur les données présentes en session ;
 - la suppression d'un client ou d'une commande ne supprimera pas seulement les données de la session, mais également de la base.

C'est « tout » ce que je vous demande...

Conseils

Ne vous fiez pas à la faible longueur du sujet : énormément de travail et de réflexion vous attendent !

Création de la base de données

La première étape consiste à créer la base et les tables représentant les données de votre application. Ceci ne faisant pas directement l'objet de ce cours, je vais vous accompagner en vous détaillant chaque instruction. Suivez le guide !

Connexion au serveur MySQL

Connectez-vous à votre serveur via la commande `mysql -h localhost -u root -p`, puis entrez votre mot de passe le cas échéant.

Création d'une base

Mettez en place une nouvelle base de données nommée **tp_sdzee** à l'aide de l'instruction suivante :

Code : SQL

```
CREATE DATABASE tp_sdzee DEFAULT CHARACTER SET utf8 COLLATE  
utf8_general_ci;
```

Mise en place des droits

Vous savez qu'utiliser le compte **root** est une mauvaise pratique, vous devez donc accorder des droits sur la nouvelle table à l'utilisateur **java** créé dans le cadre du cours, via l'instruction suivante :

Code : SQL

```
GRANT ALL ON tp_sdzee.* TO 'java'@'localhost' IDENTIFIED BY  
'SdZ_eE';
```

Vous devez ensuite vous déconnecter de l'utilisateur `root` via la commande `exit`, puis vous reconnecter avec votre compte `java` via la commande `mysql -h localhost -u java -p`, entrer le mot de passe et enfin entrer la commande `use tp_sdzee` pour définir la base sur laquelle vous allez travailler.

Création des tables

Vous devez alors créer une table **Client** contenant les champs **id**, **nom**, **prenom**, **adresse**, **telephone**, **email** et **image**. Voici un exemple d'instruction que vous pouvez utiliser pour cela :

Code : SQL

```
CREATE TABLE tp_sdzee.Client (
    id INT( 11 ) NOT NULL AUTO_INCREMENT ,
    nom VARCHAR( 20 ) NOT NULL ,
    prenom VARCHAR( 20 ) ,
    adresse VARCHAR( 200 ) NOT NULL ,
    telephone VARCHAR( 10 ) NOT NULL ,
    email VARCHAR( 60 ) ,
    image VARCHAR( 200 ) ,
    PRIMARY KEY ( id )
) ENGINE = INNODB;
```

Rien de particulier ici, remarquez simplement la mise en place d'un champ **id** pour identifier chaque client.

De même, vous devez créer une table **Commande** contenant les champs **id**, **id_client**, **date**, **montant**, **modePaiement**, **statutPaiement**, **modeLivraison** et **statutLivraison** :

Code : SQL

```
CREATE TABLE tp_sdzee.Commande (
    id INT( 11 ) NOT NULL AUTO_INCREMENT ,
    id_client INT( 11 ) ,
    date DATETIME NOT NULL ,
    montant DEC( 11 ) NOT NULL ,
    mode_paiement VARCHAR( 20 ) NOT NULL ,
    statut_paiement VARCHAR( 20 ) ,
    mode_livraison VARCHAR( 20 ) NOT NULL ,
    statut_livraison VARCHAR( 20 ) ,
    PRIMARY KEY ( id ) ,
    CONSTRAINT fk_id_client
        FOREIGN KEY (id_client)      -- On donne un nom à notre clé
                                         -- Colonne sur laquelle on crée la cl
                                         é
                                         -- Colonne de référence (celle de la
                                         REFERENCES Client(id)      -- Action à effectuer lors de la
                                         table Client)                suppression d'une référence
                                         -- ON DELETE SET NULL
                                         -- ENGINE = INNODB;
```

Première chose, vous remarquez ici que vous n'allez pas enregistrer les données des clients directement dans la table **Commande**, mais uniquement leur **id** dans le champ **id_client**. Les clients existant déjà dans la table **Client**, il serait idiot de dupliquer les données !

Deuxième chose, observez en fin d'instruction la mise en place d'une **contrainte de clé étrangère** sur le champ **id_client**. Grâce à cette clé, le SGBD sait que le contenu du champ **id_client** correspond à un **id** existant dans la table **Client**.

Enfin, ne manquez pas l'**option de la clé étrangère** ! Puisqu'une relation est définie par le SGBD entre une commande et son client, il devient impossible par défaut de supprimer un client s'il a déjà passé une commande. Dans le cadre de ce TP, ce n'est pas très ergonomique : vous allez donc devoir changer ce comportement, en précisant que lors de la suppression d'un client, les champs **id_client** de la table **Commande** qui y font référence devront passer automatiquement à **NULL**. C'est le rôle de la ligne **ON DELETE SET NULL**.

 Si vous le souhaitez, vous pouvez changer le comportement adopté ici. Vous pouvez par exemple faire en sorte qu'il soit impossible de supprimer un client s'il a déjà passé une commande, ou bien encore faire en sorte que lors de la suppression d'un client, toutes les commandes qu'il a passées soient supprimées. Tout cela est simplement paramétrable via l'option appliquée sur la clé étrangère mise en place !

En construisant ainsi vos tables, vous allez pouvoir vous assurer que les données écrites dans votre base sont cohérentes.

Mise en place de JDBC

Si ce n'est pas déjà fait, vous allez devoir placer le jar du driver JDBC pour MySQL dans le dossier **/lib** de Tomcat, afin que votre application puisse communiquer avec votre base de données.

Réutilisation de la structure DAO développée dans le cadre du cours

Votre base de données est en place. Vous pouvez maintenant attaquer le développement de votre application. Pour commencer, une bonne nouvelle : vous allez pouvoir réutiliser certaines classes du package `com.sdzee.dao` que vous avez développées dans le cadre du cours ! Les voici :

- **DAOConfigurationException.java**
- **DAOException.java**
- **DAOUtilitaire.java**

En effet, inutile de réinventer la roue pour toutes celles-là : elles fonctionnent, vous avez passé du temps à les mettre en place, autant s'en resservir ! Vous pouvez donc créer un package `com.sdzee.tp.dao` dans votre projet et y copier chacune d'elles.

De même, vous allez pouvoir réutiliser la classe **InitialisationDaoFactory.java** et la placer dans un nouveau package `com.sdzee.tp.config`.

Enfin, il reste deux fichiers que vous allez pouvoir réutiliser, moyennant quelques petits ajustements :

- vous pouvez reprendre le fichier **dao.properties**, mais il faudra bien penser à changer le contenu de son entrée **url**, pour cibler la nouvelle base **tp_sdzee** que vous avez créée, et non plus **bdd_sdzee** comme c'était le cas dans le cadre du cours ;
- vous pouvez de même reprendre la classe **DAOFactory.java**, mais vous allez devoir y changer le chemin vers le fichier **dao.properties** défini dans la constante **FICHIER_PROPERTIES**.

Prenez bien le temps de mettre cette structure en place, n'allez pas trop vite et vérifiez que tout est bien au bon endroit avant de poursuivre ! 

Création des interfaces et implémentations du DAO

Maintenant que le cadre est en place, vous allez devoir mettre la main à la pâte et coder les classes du DAO spécifiques à votre projet :

- l'interface **ClientDao** et son implémentation **ClientDaoImpl** ;
- l'interface **CommandeDao** et son implémentation **CommandeDaoImpl**.

Les interfaces

Dans le cadre de ce TP, vous n'avez pas développé de formulaires pour la modification des données existantes. Ainsi, vous n'avez pas besoin de définir toutes les méthodes *CRUD* dans vos DAO. Vous pouvez vous contenter des méthodes `creer()`, `trouver()`, `lister()` et `supprimer()`. Inspirez-vous de ce que vous avez mis en place dans le chapitre précédent si vous ne vous souvenez plus comment procéder.

Les implémentations

Vous allez devoir définir les requêtes SQL à effectuer sur vos tables, créer et manipuler des requêtes préparées depuis vos différentes méthodes, tout cela bien entendu en réutilisant les méthodes développées dans la classe **DAOUtilitaire**. Vous pouvez, là encore, vous inspirer grandement de ce que vous avez développé dans le chapitre précédent, mais vous devrez prendre garde à modifier les méthodes `map()`, à créer une méthode `lister()` et à bien ajuster les méthodes `trouver()` et `creer()`, et enfin `supprimer()` !

Afin de vous permettre de partir du bon pied, je vous donne ici les requêtes à utiliser respectivement dans les classes **ClientDaoImpl** et **CommandeDaoImpl** :

Code : Java - Constantes définissant les requêtes dans ClientDaoImpl

```
private static final String SQL_SELECT      = "SELECT id, nom,
prenom, adresse, telephone, email, image FROM Client ORDER BY id";
private static final String SQL_SELECT_PAR_ID = "SELECT id, nom,
prenom, adresse, telephone, email, image FROM Client WHERE id = ?";
private static final String SQL_INSERT      = "INSERT INTO Client
(nom, prenom, adresse, telephone, email, image) VALUES (?, ?, ?, ?, ?, ?)";
private static final String SQL_DELETE_PAR_ID = "DELETE FROM Client
WHERE id = ?";
```

Code : Java - Constantes définissant les requêtes dans CommandeDaoImpl

```
private static final String SQL_SELECT      = "SELECT id,
id_client, date, montant, mode_paiement, statut_paiement,
mode_livraison, statut_livraison FROM Commande ORDER BY id";
private static final String SQL_SELECT_PAR_ID = "SELECT id,
id_client, date, montant, mode_paiement, statut_paiement,
mode_livraison, statut_livraison FROM Commande WHERE id = ?";
private static final String SQL_INSERT      = "INSERT INTO
Commande (id_client, date, montant, mode_paiement, statut_paiement,
mode_livraison, statut_livraison) VALUES (?, ?, ?, ?, ?, ?, ?, ?)";
private static final String SQL_DELETE_PAR_ID = "DELETE FROM
Commande WHERE id = ?";
```

Avec ceci en poche, vous n'avez plus à vous soucier de l'aspect SQL du développement, et vous pouvez vous concentrer sur l'aspect... Java EE ! 😊

Intégration dans le code existant

Afin d'intégrer correctement vos DAO dans l'application, vous allez devoir procéder à de nombreux changements dans le code existant...

Modification des beans

Vous allez devoir apporter deux changements :

- ajouter un champ **id** de type Long aux deux beans, pour refléter l'**id** auto-généré existant dans vos tables ;
- modifier le type du champ **date** dans le bean **Commande**. Jusqu'à présent vous stockiez cette information sous forme d'une chaîne de caractères pour ne pas vous compliquer la tâche, mais cette fois vous allez devoir faire correspondre ce champ à celui créé dans la table **Commande**. Puisque nous utilisons la bibliothèque JodaTime, je vous conseille de changer le type du champ **date** de String vers DateTime.

Modification des servlets

Vous allez devoir adapter vos servlets **CreationClient** et **CreationCommande**, toujours sur le modèle de ce que vous avez développé dans le cours :

- récupérer la ou les implémentations de DAO depuis la *factory*, dans la méthode `init()` de chaque servlet ;
- passage du ou des DAO à l'objet métier lors de sa construction.

En outre, vous allez devoir apporter un autre changement, qui va avoir un impact considérable sur le reste de l'application. Jusqu'à présent, vous manipuliez des Map indexées sur les noms et dates des clients et commandes, ce qui rendait impossible la création de deux clients portant le même nom et la création de deux commandes au même instant.

Maintenant que vos données sont identifiables par un **id** dans vos tables, vous allez pouvoir modifier vos Map pour qu'elles indexent les id des clients et commandes. Vos Map vont donc changer de `Map<String, Client>` et `Map<String, Commande>`.

Commande> vers Map<Long, Client> et Map<Long, Commande>. Vous allez donc devoir prendre garde, partout où ces maps sont manipulées, à ne plus travailler sur les noms et dates, mais bien sur les id. Ne prenez pas ce changement à la légère, il va impliquer pas mal de petites modifications ça et là... !

Vous allez également devoir modifier vos servlets de suppression :

- pour qu'elles se basent sur l'**id** lors de la suppression d'une entrée dans une Map, suite à la modification apportée précédemment ;
- pour qu'elles suppriment également les données en base, ceci implique donc la récupération du DAO depuis la méthode `init()` et un appel à la méthode `supprimer()` du DAO.

Modification des objets métier

Vous allez devoir reprendre vos deux objets métier **CreationClientForm** et **CreationCommandeForm** pour qu'ils interagissent avec la BDD. Là encore, vous pouvez vous inspirer grandement de ce que vous avez développé dans le chapitre précédent. Dans les grandes lignes, il va vous falloir :

- récupérer la ou les instances de DAO nécessaires depuis le constructeur ;
- effectuer un appel à la méthode `creer()` du DAO une fois la validation des champs passée avec succès, et gérer les éventuelles erreurs ou exceptions retournées.

Attention dans le code de **CreationCommandeForm** ! Vous devrez prendre garde à bien adapter la manipulation de la Map des clients, suite aux changements que vous avez apportés au maps définies dans vos servlets !

Modification des JSP

Premièrement, il va falloir adapter vos JSP aux changements effectués sur vos maps de clients et commandes. Dans les pages **listerClients.jsp** et **listerCommandes.jsp**, vous allez donc devoir modifier les liens de suppression pour qu'ils ne transmettent plus le nom d'un client ou la date d'une commande, mais leur **id**.

Deuxièmement, vous allez devoir modifier la manière dont vous affichez le champ **date** du bean **Commande** ! Eh oui, maintenant que vous l'avez remplacé par un objet Joda **DateTime**, vous ne pouvez plus vous contenter de l'afficher bêtement via la balise `<c:out>`... Pas de panique, les développeurs de Joda ont pensé à ce cas de figure et ont créé une bibliothèque de balises destinées à manipuler les objets Joda !

Vous devez donc télécharger son jar en cliquant sur [ce lien](#), puis le placer dans le répertoire **/WEB-INF/lib** de votre projet. Ensuite, vous allez pouvoir utiliser les balises comme vous utilisez celles de la JSTL. Voici un exemple d'utilisation, qui se charge de formater un objet **DateTime** pour affichage à l'utilisateur selon un *pattern* défini :

Code : JSP - Exemple d'utilisation d'une balise de la bibliothèque Joda

```
<%@ taglib prefix="joda" uri="http://www.joda.org/joda/time/tags" %>
...
<joda:format value="${ commande.date }" pattern="dd/MM/yyyy
HH:mm:ss"/>
```

Je vous invite bien entendu à parcourir la documentation du projet plus en profondeur pour découvrir les différentes balises et attributs disponibles.

Création d'un filtre

Enfin, il reste un aspect important à changer dans le mode de fonctionnement de votre application. Je vous ai demandé de faire en sorte que vos pages listant les commandes et clients existants continuent à se baser sur les infos présentes en session. Il va donc falloir réfléchir un petit peu à la manière de procéder ici. En effet, au tout premier lancement de l'application, aucun problème : aucune donnée n'existe, et donc rien n'existe en session.

Imaginez maintenant que l'utilisateur crée tout un tas de clients et commandes, puis quitte l'application et ne revient pas pendant

une semaine. À son retour, la session n'existera plus depuis belle lurette sur le serveur, qui aura tôt fait de la supprimer. Ainsi, les pages listant les clients et commandes n'afficheront rien, alors qu'en réalité il existe des données, non pas dans la session, mais dans la base de données !

Ce qu'il vous faut gérer, c'est donc le préchargement des données existant en base dans la session de l'utilisateur lors de son arrivée, et uniquement lors de son arrivée, sur n'importe quelle page de l'application.



Comment réaliser une telle opération ?

En principe, vous avez déjà la réponse à cette question ! Réfléchissez bien. Quel composant est idéal pour ce type de fonctionnalité ? Eh oui, c'est le filtre ! Vous allez donc créer un filtre appliqué à toutes les pages de votre application, qui se chargera de vérifier si une session utilisateur existe, et qui la remplira avec les données présentes en base si elle n'existe pas. En somme, vous allez y récupérer une implémentation de DAO comme vous l'avez déjà fait pour les servlets, effectuer un appel à sa méthode `lister()`, et peupler les maps de clients et de commandes avec les données contenues dans les listes récupérées !

N'oubliez pas de bien déclarer le filtre dans le fichier **web.xml** !

Voilà tout ce dont vous avez besoin. Comme vous le voyez, le sujet était très court, mais en réalité le travail est conséquent !

Correction

Faites attention à bien reprendre les fichiers du cours qui restent inchangés, à corriger les quelques classes qui nécessitent des ajustements, et surtout ne baissez pas les bras devant la charge de travail impliquée par l'introduction d'une base de données dans votre programme ! Cet exercice est l'occasion parfaite pour vous familiariser avec ces concepts si importants. Ne la gâchez pas en abandonnant dès le moindre petit obstacle ! 😊

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 de coder par vous-mêmes. Si besoin, n'hésitez pas à relire le sujet ou à retourner lire les chapitres précédents. La pratique est très importante, ne vous ruez pas sur la solution !

Code de la structure DAO

Voici les liens directs vers les fichiers que vous devez reprendre tels quels depuis le chapitre précédent :

- [InitialisationDaoFactory \(inchangé\)](#)
- [DAOConfigurationException \(inchangé\)](#)
- [DAOException \(inchangé\)](#)
- [DAOUtilitaire \(inchangé\)](#)

Voilà le contenu du nouveau fichier **dao.properties**, à placer dans `com.sdzee.tp.dao` :

Code : Java - dao.properties

```
url = jdbc:mysql://localhost:3306/tp_sdzee
driver = com.mysql.jdbc.Driver
nomutilisateur = java
motdepasse = SdZ_eE
```

Et voilà le code de la classe **DAOFactory** adapté à ce nouveau fichier *Properties* et au nouveau projet :

Code : Java - com.sdzee.tp.dao.DAOFactory

```
package com.sdzee.tp.dao;
```

```
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStream;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.Properties;

public class DAOFactory {

    private static final String FICHIER_PROPERTIES      =
"/com/sdzee/tp/dao/dao.properties";
    private static final String PROPERTY_URL             =
"url";
    private static final String PROPERTY_DRIVER          =
"driver";
    private static final String PROPERTY_NOM_UTILISATEUR =
"nomutilisateur";
    private static final String PROPERTY_MOT_DE_PASSE     =
"motdepassé";

    private String           url;
    private String           username;
    private String           password;

    /* package */ DAOFactory( String url, String username, String
password ) {
        this.url = url;
        this.username = username;
        this.password = password;
    }

    /*
     * Méthode chargée de récupérer les informations de connexion à la
     * base de
     * données, charger le driver JDBC et retourner une instance de la
     * Factory
     */
    public static DAOFactory getInstance() throws
DAOConfigurationException {
        Properties properties = new Properties();
        String url;
        String driver;
        String nomUtilisateur;
        String motDePasse;

        ClassLoader classLoader =
Thread.currentThread().getContextClassLoader();
        InputStream fichierProperties =
classLoader.getResourceAsStream( FICHIER_PROPERTIES );

        if ( fichierProperties == null ) {
            throw new DAOConfigurationException( "Le fichier
properties " + FICHIER_PROPERTIES + " est introuvable." );
        }

        try {
            properties.load( fichierProperties );
            url = properties.getProperty( PROPERTY_URL );
            driver = properties.getProperty( PROPERTY_DRIVER );
            nomUtilisateur = properties.getProperty(
PROPERTY_NOM_UTILISATEUR );
            motDePasse = properties.getProperty(
PROPERTY_MOT_DE_PASSE );
        } catch ( FileNotFoundException e ) {
            throw new DAOConfigurationException( "Le fichier
properties " + FICHIER_PROPERTIES + " est introuvable.", e );
        } catch ( IOException e ) {
            throw new DAOConfigurationException( "Impossible de
charger le fichier properties " + FICHIER_PROPERTIES, e );
        }
    }
}
```

```

        try {
            Class.forName( driver );
        } catch ( ClassNotFoundException e ) {
            throw new DAOConfigurationException( "Le driver est
introuvable dans le classpath.", e );
        }

        DAOFactory instance = new DAOFactory( url, nomUtilisateur,
motDePasse );
        return instance;
    }

    /* Méthode chargée de fournir une connexion à la base de
données */
    /* package */Connection getConnection() throws SQLException {
        return DriverManager.getConnection( url, username, password
);
    }

    /*
     * Méthodes de récupération de l'implémentation des différents DAO
     * (uniquement deux dans le cadre de ce TP)
     */
    public ClientDao getClientDao() {
        return new ClientDaoImpl( this );
    }

    public CommandeDao getCommandeDao() {
        return new CommandeDaoImpl( this );
    }
}

```

Code des interfaces DAO

L'interface **ClientDao** :

Code : Java - com.sdzee.tp.dao.ClientDao

```

package com.sdzee.tp.dao;

import java.util.List;

import com.sdzee.tp.beans.Client;

public interface ClientDao {
    void creer( Client client ) throws DAOException;

    Client trouver( long id ) throws DAOException;

    List<Client> lister() throws DAOException;

    void supprimer( Client client ) throws DAOException;
}

```

L'interface **CommandeDao** :

Code : Java - com.sdzee.tp.dao.CommandeDao

```

package com.sdzee.tp.dao;

import java.util.List;

```

```

import com.sdzee.tp.beans.Commande;

public interface CommandeDao {
    void creer( Commande commande ) throws DAOException;
    Commande trouver( long id ) throws DAOException;
    List<Commande> lister() throws DAOException;
    void supprimer( Commande commande ) throws DAOException;
}

```

Code des implémentations DAO

Code de ClientDaoImpl :

Code : Java - com.sdzee.tp.dao.ClientDaoImpl

```

package com.sdzee.tp.dao;

import static com.sdzee.tp.dao.DAOUtilitaire.fermeturesSilencieuses;
import static com.sdzee.tp.dao.DAOUtilitaire.initialisationRequetePreparee;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.List;

import com.sdzee.tp.beans.Client;

public class ClientDaoImpl implements ClientDao {

    private static final String SQL_SELECT      = "SELECT id, nom,";
    prenom, adresse, telephone, email, image FROM Client ORDER BY id";
    private static final String SQL_SELECT_PAR_ID = "SELECT id, nom,";
    prenom, adresse, telephone, email, image FROM Client WHERE id = ?";
    private static final String SQL_INSERT       = "INSERT INTO
Client (nom, prenom, adresse, telephone, email, image) VALUES (?, ?, ?, ?, ?, ?)";
    private static final String SQL_DELETE_PAR_ID = "DELETE FROM
Client WHERE id = ?";

    private DAOFactory           daoFactory;

    ClientDaoImpl( DAOFactory daoFactory ) {
        this.daoFactory = daoFactory;
    }

    /* Implémentation de la méthode définie dans l'interface
    ClientDao */
    @Override
    public Client trouver( long id ) throws DAOException {
        return trouver( SQL_SELECT_PAR_ID, id );
    }

    /* Implémentation de la méthode définie dans l'interface
    ClientDao */
    @Override
    public void creer( Client client ) throws DAOException {
        Connection connexion = null;
        PreparedStatement preparedStatement = null;

```

```

        ResultSet valeursAutoGenerees = null;

        try {
            connexion = daoFactory.getConnection();
            preparedStatement = initialisationRequetePreparee(
                connexion, SQL_INSERT, true,
                client.getNom(), client.getPrenom(),
                client.getAdresse(), client.getTelephone(),
                client.getEmail(), client.getImage() );
            int statut = preparedStatement.executeUpdate();
            if ( statut == 0 ) {
                throw new DAOException( "Échec de la création du
client, aucune ligne ajoutée dans la table." );
            }
            valeursAutoGenerees =
            preparedStatement.getGeneratedKeys();
            if ( valeursAutoGenerees.next() ) {
                client.setId( valeursAutoGenerees.getLong( 1 ) );
            } else {
                throw new DAOException( "Échec de la création du
client en base, aucun ID auto-généré retourné." );
            }
        } catch ( SQLException e ) {
            throw new DAOException( e );
        } finally {
            fermeturesSilencieuses( valeursAutoGenerees,
            preparedStatement, connexion );
        }
    }

    /* Implémentation de la méthode définie dans l'interface
ClientDao */
    @Override
    public List<Client> lister() throws DAOException {
        Connection connection = null;
        PreparedStatement preparedStatement = null;
        ResultSet resultSet = null;
        List<Client> clients = new ArrayList<Client>();

        try {
            connection = daoFactory.getConnection();
            preparedStatement = connection.prepareStatement(
                SQL_SELECT );
            resultSet = preparedStatement.executeQuery();
            while ( resultSet.next() ) {
                clients.add( map( resultSet ) );
            }
        } catch ( SQLException e ) {
            throw new DAOException( e );
        } finally {
            fermeturesSilencieuses( resultSet, preparedStatement,
            connection );
        }

        return clients;
    }

    /* Implémentation de la méthode définie dans l'interface
ClientDao */
    @Override
    public void supprimer( Client client ) throws DAOException {
        Connection connexion = null;
        PreparedStatement preparedStatement = null;

        try {
            connexion = daoFactory.getConnection();
            preparedStatement = initialisationRequetePreparee(
                connexion, SQL_DELETE_PAR_ID, true, client.getId() );
            int statut = preparedStatement.executeUpdate();
            if ( statut == 0 ) {

```

```

        throw new DAOException( "Échec de la suppression du
client, aucune ligne supprimée de la table." );
    } else {
        client.setId( null );
    }
} catch ( SQLException e ) {
    throw new DAOException( e );
} finally {
    fermeturesSilencieuses( preparedStatement, connexion );
}
}

/*
* Méthode générique utilisée pour retourner un client depuis la
base de
* données, correspondant à la requête SQL donnée prenant en
paramètres les
* objets passés en argument.
*/
private Client trouver( String sql, Object... objets ) throws
DAOException {
    Connection connexion = null;
    PreparedStatement preparedStatement = null;
    ResultSet resultSet = null;
    Client client = null;

    try {
        /* Récupération d'une connexion depuis la Factory */
        connexion = daoFactory.getConnection();
        /*
        * Préparation de la requête avec les objets passés en arguments
        * (ici, uniquement un id) et exécution.
        */
        preparedStatement = initialisationRequetePreparee(
connexion, sql, false, objets );
        resultSet = preparedStatement.executeQuery();
        /* Parcours de la ligne de données retournée dans le
ResultSet */
        if ( resultSet.next() ) {
            client = map( resultSet );
        }
    } catch ( SQLException e ) {
        throw new DAOException( e );
    } finally {
        fermeturesSilencieuses( resultSet, preparedStatement,
connexion );
    }

    return client;
}

/*
* Simple méthode utilitaire permettant de faire la correspondance
(le
* mapping) entre une ligne issue de la table des clients (un
ResultSet) et
* un bean Client.
*/
private static Client map( ResultSet resultSet ) throws
SQLException {
    Client client = new Client();
    client.setId( resultSet.getLong( "id" ) );
    client.setNom( resultSet.getString( "nom" ) );
    client.setPrenom( resultSet.getString( "prenom" ) );
    client.setAdresse( resultSet.getString( "adresse" ) );
    client.setTelephone( resultSet.getString( "telephone" ) );
    client.setEmail( resultSet.getString( "email" ) );
    client.setImage( resultSet.getString( "image" ) );
    return client;
}
}

```

```
}
```

Code de CommandeDaoImpl :

Code : Java - com.sdzee.tp.dao.CommandeDaoImpl

```
package com.sdzee.tp.dao;

import static com.sdzee.tp.dao.DAOUtilitaire.fermeturesSilencieuses;
import static com.sdzee.tp.dao.DAOUtilitaire.initialisationRequetePreparee;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Timestamp;
import java.util.ArrayList;
import java.util.List;

import org.joda.time.DateTime;

import com.sdzee.tp.beans.Commande;

public class CommandeDaoImpl implements CommandeDao {

    private static final String SQL_SELECT      = "SELECT id,
id_client, date, montant, mode_paiement, statut_paiement,
mode_livraison, statut_livraison FROM Commande ORDER BY id";
    private static final String SQL_SELECT_PAR_ID = "SELECT id,
id_client, date, montant, mode_paiement, statut_paiement,
mode_livraison, statut_livraison FROM Commande WHERE id = ?";
    private static final String SQL_INSERT       = "INSERT INTO
Commande (id_client, date, montant, mode_paiement, statut_paiement,
mode_livraison, statut_livraison) VALUES (?, ?, ?, ?, ?, ?, ?)";
    private static final String SQL_DELETE_PAR_ID = "DELETE FROM
Commande WHERE id = ?";

    private DAOFactory           daoFactory;

    CommandeDaoImpl( DAOFactory daoFactory ) {
        this.daoFactory = daoFactory;
    }

    /* Implémentation de la méthode définie dans l'interface
CommandeDao */
    @Override
    public Commande trouver( long id ) throws DAOException {
        return trouver( SQL_SELECT_PAR_ID, id );
    }

    /* Implémentation de la méthode définie dans l'interface
CommandeDao */
    @Override
    public void creer( Commande commande ) throws DAOException {
        Connection connexion = null;
        PreparedStatement preparedStatement = null;
        ResultSet valeursAutoGenerees = null;

        try {
            connexion = daoFactory.getConnection();
            preparedStatement = initialisationRequetePreparee(
connexion, SQL_INSERT, true,
                commande.getClient().getId(), new Timestamp(
commande.getDate().getMillis() ),
                commande.getMontant(), commande.getDate(),
                commande.getModePaiement(), commande.getStatutPaiement(),
                commande.getModeLivraison(), commande.getStatutLivraison());
            preparedStatement.executeUpdate();
        } catch ( SQLException e ) {
            throw new DAOException( "Problème lors de l'insertion de la
commande", e );
        } finally {
            if ( connexion != null ) {
                try {
                    connexion.close();
                } catch ( SQLException e ) {
                    e.printStackTrace();
                }
            }
            if ( preparedStatement != null ) {
                try {
                    preparedStatement.close();
                } catch ( SQLException e ) {
                    e.printStackTrace();
                }
            }
            if ( valeursAutoGenerees != null ) {
                try {
                    valeursAutoGenerees.close();
                } catch ( SQLException e ) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

```
        commande.getMontant(),
        commande.getModePaiement(),
        commande.getStatutPaiement(),
        commande.getModeLivraison(),
        commande.getStatutLivraison() );
        int statut = preparedStatement.executeUpdate();
        if ( statut == 0 ) {
            throw new DAOException( "Échec de la création de la
commande, aucune ligne ajoutée dans la table." );
        }
        valeursAutoGenerees =
preparedStatement.getGeneratedKeys();
        if ( valeursAutoGenerees.next() ) {
            commande.setId( valeursAutoGenerees.getLong( 1 ) );
        } else {
            throw new DAOException( "Échec de la création de la
commande en base, aucun ID auto-généré retourné." );
        }
    } catch ( SQLException e ) {
        throw new DAOException( e );
    } finally {
        fermeturesSilencieuses( valeursAutoGenerees,
preparedStatement, connexion );
    }
}

/* Implémentation de la méthode définie dans l'interface
ClientDao */
@Override
public List<Commande> lister() throws DAOException {
Connection connection = null;
PreparedStatement preparedStatement = null;
ResultSet resultSet = null;
List<Commande> commandes = new ArrayList<Commande>();

try {
    connection = daoFactory.getConnection();
    preparedStatement = connection.prepareStatement(
SQL_SELECT );
    resultSet = preparedStatement.executeQuery();
    while ( resultSet.next() ) {
        commandes.add( map( resultSet ) );
    }
} catch ( SQLException e ) {
    throw new DAOException( e );
} finally {
    fermeturesSilencieuses( resultSet, preparedStatement,
connection );
}

return commandes;
}

/* Implémentation de la méthode définie dans l'interface
CommandeDao */
@Override
public void supprimer( Commande commande ) throws DAOException {
Connection connexion = null;
PreparedStatement preparedStatement = null;

try {
    connexion = daoFactory.getConnection();
    preparedStatement = initialisationRequetePreparee(
connexion, SQL_DELETE_PAR_ID, true, commande.getId() );
    int statut = preparedStatement.executeUpdate();
    if ( statut == 0 ) {
        throw new DAOException( "Échec de la suppression de
la commande, aucune ligne supprimée de la table." );
    } else {
        commande.setId( null );
    }
}
```

```
        }
    } catch ( SQLException e ) {
        throw new DAOException( e );
    } finally {
        fermeturesSilencieuses( preparedStatement, connexion );
    }
}

/*
* Méthode générique utilisée pour retourner une commande depuis la
base de
* données, correspondant à la requête SQL donnée prenant en
paramètres les
* objets passés en argument.
*/
private Commande trouver( String sql, Object... objets ) throws
DAOException {
    Connection connexion = null;
    PreparedStatement preparedStatement = null;
    ResultSet resultSet = null;
    Commande commande = null;

    try {
        /* Récupération d'une connexion depuis la Factory */
        connexion = daoFactory.getConnection();
        /*
        * Préparation de la requête avec les objets passés en arguments
        * (ici, uniquement un id) et exécution.
        */
        preparedStatement = initialisationRequetePreparee(
connexion, sql, false, objets );
        resultSet = preparedStatement.executeQuery();
        /* Parcours de la ligne de données retournée dans le
ResultSet */
        if ( resultSet.next() ) {
            commande = map( resultSet );
        }
    } catch ( SQLException e ) {
        throw new DAOException( e );
    } finally {
        fermeturesSilencieuses( resultSet, preparedStatement,
connexion );
    }

    return commande;
}

/*
* Simple méthode utilitaire permettant de faire la correspondance
(le
* mapping) entre une ligne issue de la table des commandes (un
ResultSet)
* et un bean Commande.
*/
private Commande map( ResultSet resultSet ) throws SQLException
{
    Commande commande = new Commande();
    commande.setId( resultSet.getLong( "id" ) );

    /*
    * Petit changement ici : pour récupérer un client, il nous faut
faire
    * appel à la méthode trouver() du DAO Client, afin de récupérer un
bean
    * Client à partir de l'id présent dans la table Commande.
    */
    ClientDao clientDao = daoFactory.getClientDao();
    commande.setClient( clientDao.trouver( resultSet.getLong(
"id_client" ) ) );
}
```

```

        commande.setDate( new DateTime( resultSet.getTimestamp(
"date" ) ) );
        commande.setMontant( resultSet.getDouble( "montant" ) );
        commande.setModePaiement( resultSet.getString(
"mode_paiement" ) );
        commande.setStatutPaiement( resultSet.getString(
"statut_paiement" ) );
        commande.setModeLivraison( resultSet.getString(
"mode_livraison" ) );
        commande.setStatutLivraison( resultSet.getString(
"statut_livraison" ) );
        return commande;
    }

}

```

Code des beans

Client :

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

```

package com.sdzee.tp.beans;

import java.io.Serializable;

public class Client implements Serializable {

    private Long id;
    private String nom;
    private String prenom;
    private String adresse;
    private String telephone;
    private String email;
    private String image;

    public void setId( Long id ) {
        this.id = id;
    }

    public Long getId() {
        return id;
    }

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

public String getTelephone() {
    return telephone;
}

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

public String getEmail() {
    return email;
}

public void setImage( String image ) {
    this.image = image;
}

public String getImage() {
    return image;
}

}
```

Commande :

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

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

import java.io.Serializable;

import org.joda.time.DateTime;

public class Commande implements Serializable {
    private Long id;
    private Client client;
    private DateTime date;
    private Double montant;
    private String modePaiement;
    private String statutPaiement;
    private String modeLivraison;
    private String statutLivraison;

    public Long getId() {
        return id;
    }

    public void setId( Long id ) {
        this.id = id;
    }

    public Client getClient() {
        return client;
    }

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

    public DateTime getDate() {
        return date;
    }
}
```

```

public void setDate( DateTime 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;
}
}

```

Code des objets métier

Code de CreationClientForm :

Code : Java - com.sdzee.tp.forms.CreationClientForm

```

package com.sdzee.tp.forms;

import java.io.BufferedReader;
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.util.Collection;
import java.util.HashMap;
import java.util.Map;

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

```

```
import javax.servlet.http.Part;

import com.sdzee.tp.beans.Client;
import com.sdzee.tp.dao.ClientDao;
import com.sdzee.tp.dao.DAOException;

import eu.medsea.mimeutil.MimeUtil;

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 static final String CHAMP_IMAGE   = "imageClient";

    private static final int     TAILLE_TAMPON = 10240;
    // 10ko

    private String             resultat;
    private Map<String, String> erreurs
        = new
    HashMap<String, String>();
    private ClientDao           clientDao;

    public CreationClientForm( ClientDao clientDao ) {
        this.clientDao = clientDao;
    }

    public Map<String, String> getErreurs() {
        return erreurs;
    }

    public String getResultat() {
        return resultat;
    }

    public Client creerClient( HttpServletRequest request, String
chemin ) {
        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();

        traiterNom( nom, client );
        traiterPrenom( prenom, client );
        traiterAdresse( adresse, client );
        traiterTelephone( telephone, client );
        traiterEmail( email, client );
        traiterImage( client, request, chemin );

        try {
            if ( erreurs.isEmpty() ) {
                clientDao.creer( client );
                resultat = "Succès de la création du client.";
            } else {
                resultat = "Échec de la création du client.";
            }
        } catch ( DAOException e ) {
            setErreur( "imprévu", "Erreur imprévue lors de la
création." );
            resultat = "Échec de la création du client : une erreur
imprévue est survenue, merci de réessayer dans quelques instants.";
            e.printStackTrace();
        }
    }

    return client;
}
```

```
    }

    private void traiterNom( String nom, Client client ) {
        try {
            validationNom( nom );
        } catch ( FormValidationException e ) {
            setErreur( CHAMP_NOM, e.getMessage() );
        }
        client.setNom( nom );
    }

    private void traiterPrenom( String prenom, Client client ) {
        try {
            validationPrenom( prenom );
        } catch ( FormValidationException e ) {
            setErreur( CHAMP_PRENOM, e.getMessage() );
        }
        client.setPrenom( prenom );
    }

    private void traiterAdresse( String adresse, Client client ) {
        try {
            validationAdresse( adresse );
        } catch ( FormValidationException e ) {
            setErreur( CHAMP_ADRESSE, e.getMessage() );
        }
        client.setAdresse( adresse );
    }

    private void traiterTelephone( String telephone, Client client )
    {
        try {
            validationTelephone( telephone );
        } catch ( FormValidationException e ) {
            setErreur( CHAMP_TELEPHONE, e.getMessage() );
        }
        client.setTelephone( telephone );
    }

    private void traiterEmail( String email, Client client ) {
        try {
            validationEmail( email );
        } catch ( FormValidationException e ) {
            setErreur( CHAMP_EMAIL, e.getMessage() );
        }
        client.setEmail( email );
    }

    private void traiterImage( Client client, HttpServletRequest
request, String chemin ) {
        String image = null;
        try {
            image = validationImage( request, chemin );
        } catch ( FormValidationException e ) {
            setErreur( CHAMP_IMAGE, e.getMessage() );
        }
        client.setImage( image );
    }

    private void validationNom( String nom ) throws
FormValidationException {
        if ( nom != null ) {
            if ( nom.length() < 2 ) {
                throw new FormValidationException( "Le nom
d'utilisateur doit contenir au moins 2 caractères." );
            }
        } else {
            throw new FormValidationException( "Merci d'entrer un
nom d'utilisateur." );
        }
    }
```

```

    }

    private void validationPrenom( String prenom ) throws
FormValidationException {
    if ( prenom != null && prenom.length() < 2 ) {
        throw new FormValidationException( "Le prénom
d'utilisateur doit contenir au moins 2 caractères." );
    }
}

private void validationAdresse( String adresse ) throws
FormValidationException {
    if ( adresse != null ) {
        if ( adresse.length() < 10 ) {
            throw new FormValidationException( "L'adresse de
livraison doit contenir au moins 10 caractères." );
        }
    } else {
        throw new FormValidationException( "Merci d'entrer une
adresse de livraison." );
    }
}

private void validationTelephone( String telephone ) throws
FormValidationException {
    if ( telephone != null ) {
        if ( !telephone.matches( "^\\d+$" ) ) {
            throw new FormValidationException( "Le numéro de
téléphone doit uniquement contenir des chiffres." );
        } else if ( telephone.length() < 4 ) {
            throw new FormValidationException( "Le numéro de
téléphone doit contenir au moins 4 chiffres." );
        }
    } else {
        throw new FormValidationException( "Merci d'entrer un
numéro de téléphone." );
    }
}

private void validationEmail( String email ) throws
FormValidationException {
    if ( email != null && !email.matches(
"([^.@]+)(\\.[^.@]+)*@[^.@]+\\.[^.@]+([^.@]+)" ) ) {
        throw new FormValidationException( "Merci de saisir une
adresse mail valide." );
    }
}

private String validationImage( HttpServletRequest request,
String chemin ) throws FormValidationException {
/*
 * Récupération du contenu du champ image du formulaire. Il faut ici
 * utiliser la méthode getPart().
 */
String nomFichier = null;
InputStream contenuFichier = null;
try {
    Part part = request.getPart( CHAMP_IMAGE );
    nomFichier = getNomFichier( part );
}

/*
 * Si la méthode getNomFichier() a renvoyé quelque chose, il s'agit
 * donc d'un champ de type fichier (input type="file").
*/
if ( nomFichier != null && !nomFichier.isEmpty() ) {
/*
 * Antibug pour Internet Explorer, qui transmet pour une raison
 * mystique le chemin du fichier local à la machine du client...
*/
Ex : C:/dossier/sous-dossier/fichier.ext

```

```
*  
* On doit donc faire en sorte de ne sélectionner que le nom et  
* l'extension du fichier, et de se débarrasser du superflu.  
*/  
    nomFichier = nomFichier.substring(  
nomFichier.lastIndexOf( '/') + 1 )  
                                .substring( nomFichier.lastIndexOf( '\\\\' ) +  
1 );  
  
        /* Récupération du contenu du fichier */  
        contenuFichier = part.getInputStream();  
  
        /* Extraction du type MIME du fichier depuis  
l'InputStream */  
        MimeUtil.registerMimeTypeDetector(  
"eu.medsea.mimeutil.detector.MagicMimeTypeDetector" );  
        Collection<?> mimeTypes = MimeUtil.getMimeTypes(  
contenuFichier );  
  
        /*  
* Si le fichier est bien une image, alors son en-tête MIME  
* commence par la chaîne "image"  
*/  
        if ( mimeTypes.toString().startsWith( "image" ) ) {  
            /* Écriture du fichier sur le disque */  
            ecrireFichier( contenuFichier, nomFichier,  
chemin );  
        } else {  
            throw new FormValidationException( "Le fichier  
envoyé doit être une image." );  
        }  
    } catch ( IllegalStateException e ) {  
        /*  
* Exception retournée si la taille des données dépasse les limites  
* définies dans la section <multipart-config> de la déclaration de  
* notre servlet d'upload dans le fichier web.xml  
*/  
        e.printStackTrace();  
        throw new FormValidationException( "Le fichier envoyé ne  
doit pas dépasser 1Mo." );  
    } catch ( IOException e ) {  
        /*  
* Exception retournée si une erreur au niveau des répertoires de  
* stockage survient (répertoire inexistant, droits d'accès  
* insuffisants, etc.)  
*/  
        e.printStackTrace();  
        throw new FormValidationException( "Erreur de  
configuration du serveur." );  
    } catch ( ServletException e ) {  
        /*  
* Exception retournée si la requête n'est pas de type  
* multipart/form-data.  
*/  
        e.printStackTrace();  
        throw new FormValidationException(  
                "Ce type de requête n'est pas supporté, merci  
d'utiliser le formulaire prévu pour envoyer votre fichier." );  
    }  
  
    return nomFichier;  
}  
  
/*  
* 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;
    }
}

/*
 * Méthode utilitaire qui a pour unique but d'analyser l'en-tête
 * "content-disposition", et de vérifier si le paramètre "filename"
y est
 * présent. Si oui, alors le champ traité est de type File et la
méthode
 * retourne son nom, sinon il s'agit d'un champ de formulaire
classique et
 * la méthode retourne null.
*/
private static String getNomFichier( Part part ) {
    /* Boucle sur chacun des paramètres de l'en-tête "content-
disposition". */
    for ( String contentDisposition : part.getHeader( "content-
disposition" ).split( ";" ) ) {
        /* Recherche de l'éventuelle présence du paramètre
"filename". */
        if ( contentDisposition.trim().startsWith( "filename" ) )
    {
        /*
        * Si "filename" est présent, alors renvoi de sa valeur,
        * c'est-à-dire du nom de fichier sans guillemets.
        */
        return contentDisposition.substring(
contentDisposition.indexOf( '=' ) + 1 ).trim().replace( "\\"", "" );
    }
    }
    /* Et pour terminer, si rien n'a été trouvé... */
    return null;
}

/*
 * Méthode utilitaire qui a pour but d'écrire le fichier passé en
paramètre
 * sur le disque, dans le répertoire donné et avec le nom donné.
*/
private void ecrireFichier( InputStream contenuFichier, String
nomFichier, String chemin )
    throws FormValidationException {
    /* Prépare les flux. */
    BufferedInputStream entree = null;
    BufferedOutputStream sortie = null;
    try {
        /* Ouvre les flux. */
        entree = new BufferedInputStream( contenuFichier,
TAILLE_TAMPON );
        sortie = new BufferedOutputStream( new FileOutputStream(
new File( chemin + nomFichier ) ),
TAILLE_TAMPON );

        /*
        * Lit le fichier reçu et écrit son contenu dans un fichier sur le
        * disque.
        */
    }
}
```

```
 */
byte[] tampon = new byte[TAILLE_TAMPON];
int longueur = 0;
while ( ( longueur = entree.read( tampon ) ) > 0 ) {
    sortie.write( tampon, 0, longueur );
}
} catch ( Exception e ) {
    throw new FormValidationException( "Erreur lors de
l'écriture du fichier sur le disque." );
} finally {
    try {
        sortie.close();
    } catch ( IOException ignore ) {
    }
    try {
        entree.close();
    } catch ( IOException ignore ) {
    }
}
}
```

Code de CreationCommandeForm :

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 javax.servlet.http.HttpSession;

import org.joda.time.DateTime;

import com.sdzee.tp.beans.Client;
import com.sdzee.tp.beans.Commande;
import com.sdzee.tp.dao.ClientDao;
import com.sdzee.tp.dao.CommandeDao;
import com.sdzee.tp.dao.DAOException;

public final class CreationCommandeForm {
    private static final String CHAMP_CHOIX_CLIENT =
"choixNouveauClient";
    private static final String CHAMP_LISTE_CLIENTS =
"listeClients";
    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 ANCIEN_CLIENT =
"ancienClient";
    private static final String SESSION_CLIENTS =
"clients";
    private static final String FORMAT_DATE =
"dd/MM/yyyy HH:mm:ss";
```

```
private String resultat;
private Map<String, String> erreurs = new
HashMap<String, String>();
private ClientDao clientDao;
private CommandeDao commandeDao;

public CreationCommandeForm( ClientDao clientDao, CommandeDao
commandeDao ) {
    this.clientDao = clientDao;
    this.commandeDao = commandeDao;
}

public Map<String, String> getErreurs() {
    return erreurs;
}

public String getResultat() {
    return resultat;
}

public Commande creerCommande( HttpServletRequest request,
String chemin ) {
    Client client;
    /*
    * Si l'utilisateur choisit un client déjà existant, pas de
validation à
* effectuer
*/
    String choixNouveauClient = getValeurChamp( request,
CHAMP_CHOIX_CLIENT );
    if ( ANCIEN_CLIENT.equals( choixNouveauClient ) ) {
        /* Récupération de l'id du client choisi */
        String idAncienClient = getValeurChamp( request,
CHAMP_LISTE_CLIENTS );
        Long id = null;
        try {
            id = Long.parseLong( idAncienClient );
        } catch ( NumberFormatException e ) {
            setErreur( CHAMP_CHOIX_CLIENT, "Client inconnu,
merci d'utiliser le formulaire prévu à cet effet." );
            id = 0L;
        }
        /* Récupération de l'objet client correspondant dans la
session */
        HttpSession session = request.getSession();
        client = ( (Map<Long, Client>) session.getAttribute(
SESSION_CLIENTS ) ).get( id );
    } else {
        /*
        * Sinon on garde l'ancien mode, pour la validation des champs.
*
* L'objet métier pour valider la création d'un client existe déjà,
* il est donc déconseillé de dupliquer ici son contenu ! À 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(
clientDao );
        client = clientForm.creerClient( request, chemin );

        /*
        * 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 de la date dans un DateTime Joda. */
DateTime dt = new DateTime();

String montant = getValeurChamp( request, CHAMP_MONTANT );
String modePaiement = getValeurChamp( request,
CHAMP_MODE_PAITEMENT );
String statutPaiement = getValeurChamp( request,
CHAMP_STATUT_PAITEMENT );
String modeLivraison = getValeurChamp( request,
CHAMP_MODE_LIVRAISON );
String statutLivraison = getValeurChamp( request,
CHAMP_STATUT_LIVRAISON );

Commande commande = new Commande();

try {
    traiterClient( client, commande );

    commande.setDate( dt );

    traiterMontant( montant, commande );
    traiterModePaiement( modePaiement, commande );
    traiterStatutPaiement( statutPaiement, commande );
    traiterModeLivraison( modeLivraison, commande );
    traiterStatutLivraison( statutLivraison, commande );

    if ( erreurs.isEmpty() ) {
        commandeDao.creer( commande );
        resultat = "Succès de la création de la commande.";
    } else {
        resultat = "Échec de la création de la commande.";
    }
} catch ( DAOException e ) {
    setErreur( "imprévu", "Erreur imprévue lors de la
création." );
    resultat = "Échec de la création de la commande : une
erreur imprévue est survenue, merci de réessayer dans quelques
instants.";
    e.printStackTrace();
}

return commande;
}

private void traiterClient( Client client, Commande commande ) {
    if ( client == null ) {
        setErreur( CHAMP_CHOIX_CLIENT, "Client inconnu, merci
d'utiliser le formulaire prévu à cet effet." );
    }
    commande.setClient( client );
}

private void traiterMontant( String montant, Commande commande )
{
    double valeurMontant = -1;
    try {
        valeurMontant = validationMontant( montant );
    } catch ( FormValidationException e ) {
        setErreur( CHAMP_MONTANT, e.getMessage() );
    }
    commande.setMontant( valeurMontant );
}

private void traiterModePaiement( String modePaiement, Commande
commande ) {
    try {
```

```
        validationModePaiement( modePaiement );
    } catch ( FormValidationException e ) {
        setErreur( CHAMP_MODE_PAIENT, e.getMessage() );
    }
    commande.setModePaiement( modePaiement );
}

private void traiterStatutPaiement( String statutPaiement,
Commande commande ) {
try {
    validationStatutPaiement( statutPaiement );
} catch ( FormValidationException e ) {
    setErreur( CHAMP_STATUT_PAIENT, e.getMessage() );
}
commande.setStatutPaiement( statutPaiement );
}

private void traiterModeLivraison( String modeLivraison,
Commande commande ) {
try {
    validationModeLivraison( modeLivraison );
} catch ( FormValidationException e ) {
    setErreur( CHAMP_MODE_LIVRAISON, e.getMessage() );
}
commande.setModeLivraison( modeLivraison );
}

private void traiterStatutLivraison( String statutLivraison,
Commande commande ) {
try {
    validationStatutLivraison( statutLivraison );
} catch ( FormValidationException e ) {
    setErreur( CHAMP_STATUT_LIVRAISON, e.getMessage() );
}
commande.setStatutLivraison( statutLivraison );
}

private double validationMontant( String montant ) throws
FormValidationException {
double temp;
if ( montant != null ) {
try {
temp = Double.parseDouble( montant );
if ( temp < 0 ) {
throw new FormValidationException( "Le montant
doit être un nombre positif." );
}
} catch ( NumberFormatException e ) {
temp = -1;
throw new FormValidationException( "Le montant doit
être un nombre." );
}
} else {
temp = -1;
throw new FormValidationException( "Merci d'entrer un
montant." );
}
return temp;
}

private void validationModePaiement( String modePaiement )
throws FormValidationException {
if ( modePaiement != null ) {
if ( modePaiement.length() < 2 ) {
throw new FormValidationException( "Le mode de
paiement doit contenir au moins 2 caractères." );
}
} else {
throw new FormValidationException( "Merci d'entrer un
mode de paiement." );
}
```

```

    }

    private void validationStatutPaiement( String statutPaiement )
throws FormValidationException {
    if ( statutPaiement != null && statutPaiement.length() < 2 )
{
    throw new FormValidationException( "Le statut de
paiement doit contenir au moins 2 caractères." );
}

private void validationModeLivraison( String modeLivraison )
throws FormValidationException {
    if ( modeLivraison != null ) {
        if ( modeLivraison.length() < 2 ) {
            throw new FormValidationException( "Le mode de
livraison doit contenir au moins 2 caractères." );
        }
    } else {
        throw new FormValidationException( "Merci d'entrer un
mode de livraison." );
    }
}

private void validationStatutLivraison( String statutLivraison )
throws FormValidationException {
    if ( statutLivraison != null && statutLivraison.length() < 2
) {
        throw new FormValidationException( "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

Contenu du fichier web.xml :

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

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

```
<web-app>
    <filter>
        <filter-name>Set Character Encoding</filter-name>
        <filter-
class>org.apache.catalina.filters.SetCharacterEncodingFilter</filter-
class>
        <init-param>
            <param-name>encoding</param-name>
            <param-value>UTF-8</param-value>
        </init-param>
        <init-param>
            <param-name>ignore</param-name>
            <param-value>false</param-value>
        </init-param>
    </filter>
    <filter-mapping>
        <filter-name>Set Character Encoding</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

    <filter>
        <filter-name>PrechargementFilter</filter-name>
        <filter-
class>com.sdzee.tp.filters.PrechargementFilter</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>PrechargementFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

    <listener>
    <listener-
class>com.sdzee.tp.config.InitialisationDaoFactory</listener-class>
    </listener>

    <servlet>
        <servlet-name>CreationClient</servlet-name>
        <servlet-class>com.sdzee.tp.servlets.CreationClient</servlet-class>
        <init-param>
            <param-name>chemin</param-name>
            <param-value>/fichiers/images/</param-value>
        </init-param>
        <multipart-config>
            <location>c:/fichiers/images</location>
            <max-file-size>2097152</max-file-size> <!-- 2 Mo -->
            <max-request-size>10485760</max-request-size> <!-- 5 x 2Mo -->
            <file-size-threshold>1048576</file-size-threshold> <!-- 1 Mo -->
        </multipart-config>
    </servlet>
    <servlet>
        <servlet-name>ListeClients</servlet-name>
        <servlet-class>com.sdzee.tp.servlets.ListeClients</servlet-class>
    </servlet>
    <servlet>
        <servlet-name>SuppressionClient</servlet-name>
        <servlet-class>com.sdzee.tp.servlets.SuppressionClient</servlet-
class>
    </servlet>
    <servlet>
        <servlet-name>CreationCommande</servlet-name>
        <servlet-class>com.sdzee.tp.servlets.CreationCommande</servlet-
class>
        <init-param>
            <param-name>chemin</param-name>
            <param-value>/fichiers/images/</param-value>
        </init-param>
        <multipart-config>
            <location>c:/fichiers/images</location>
            <max-file-size>2097152</max-file-size> <!-- 2 Mo -->
            <max-request-size>10485760</max-request-size> <!-- 5 x 2Mo -->
        </multipart-config>
    </servlet>
```

```

<file-size-threshold>1048576</file-size-threshold> <!-- 1 Mo -->
</multipart-config>
</servlet>
<servlet>
  <servlet-name>ListeCommandes</servlet-name>
  <servlet-class>com.sdzee.tp.servlets.ListeCommandes</servlet-class>
</servlet>
<servlet>
  <servlet-name>SuppressionCommande</servlet-name>
  <servlet-class>com.sdzee.tp.servlets.SuppressionCommande</servlet-
class>
</servlet>
<servlet>
  <servlet-name>Image</servlet-name>
  <servlet-class>com.sdzee.tp.servlets.Image</servlet-class>
<init-param>
  <param-name>chemin</param-name>
  <param-value>/fichiers/images/</param-value>
</init-param>
</servlet>

<servlet-mapping>
  <servlet-name>CreationClient</servlet-name>
  <url-pattern>/creationClient</url-pattern>
</servlet-mapping>
<servlet-mapping>
  <servlet-name>ListeClients</servlet-name>
  <url-pattern>/listeClients</url-pattern>
</servlet-mapping>
<servlet-mapping>
  <servlet-name>SuppressionClient</servlet-name>
  <url-pattern>/suppressionClient</url-pattern>
</servlet-mapping>
<servlet-mapping>
  <servlet-name>CreationCommande</servlet-name>
  <url-pattern>/creationCommande</url-pattern>
</servlet-mapping>
<servlet-mapping>
  <servlet-name>ListeCommandes</servlet-name>
  <url-pattern>/listeCommandes</url-pattern>
</servlet-mapping>
<servlet-mapping>
  <servlet-name>SuppressionCommande</servlet-name>
  <url-pattern>/suppressionCommande</url-pattern>
</servlet-mapping>
<servlet-mapping>
  <servlet-name>Image</servlet-name>
  <url-pattern>/images/*</url-pattern>
</servlet-mapping>
</web-app>

```

Code de la servlet CreationClient :

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

```

package com.sdzee.tp.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;
import javax.servlet.http.HttpSession;

```

```
import com.sdzee.tp.beans.Client;
import com.sdzee.tp.dao.ClientDao;
import com.sdzee.tp.dao.DAOFactory;
import com.sdzee.tp.forms.CreationClientForm;

public class CreationClient extends HttpServlet {
    public static final String CONF_DAO_FACTORY = "daofactory";
    public static final String CHEMIN           = "chemin";
    public static final String ATT_CLIENT       = "client";
    public static final String ATT_FORM         = "form";
    public static final String SESSION_CLIENTS  = "clients";

    public static final String VUE_SUCES          = "/WEB-INF/afficherClient.jsp";
    public static final String VUE_FORM           = "/WEB-INF/creerClient.jsp";

    private ClientDao clientDao;

    public void init() throws ServletException {
        /* Récupération d'une instance de notre DAO Utilisateur */
        this.clientDao = ((DAOFactory)
getServletContext().getAttribute( CONF_DAO_FACTORY )
).getClientDao();
    }

    public void doGet( HttpServletRequest request,
HttpServletResponse response ) throws ServletException, IOException
{
    /* À la réception d'une requête GET, simple affichage du
formulaire */
    this.getServletContext().getRequestDispatcher( VUE_FORM
).forward( request, response );
}

    public void doPost( HttpServletRequest request,
HttpServletResponse response ) throws ServletException, IOException
{
    /*
     * Lecture du paramètre 'chemin' passé à la servlet via la
déclaration
     * dans le web.xml
    */
    String chemin = this.getServletConfig().getInitParameter(
CHEMIN );

    /* Préparation de l'objet formulaire */
    CreationClientForm form = new CreationClientForm( clientDao
);

    /* Traitement de la requête et récupération du bean en
résultant */
    Client client = form.creerClient( request, chemin );

    /* Ajout du bean et de l'objet métier à l'objet requête */
    request.setAttribute( ATT_CLIENT, client );
    request.setAttribute( ATT_FORM, form );

    /* Si aucune erreur */
    if ( form.getErreurs().isEmpty() ) {
        /* Alors récupération de la map des clients dans la
session */
        HttpSession session = request.getSession();
        Map<Long, Client> clients = (HashMap<Long, Client>)
session.getAttribute( SESSION_CLIENTS );
        /* Si aucune map n'existe, alors initialisation d'une
nouvelle map */
        if ( clients == null ) {
            clients = new HashMap<Long, Client>();
        }
    }
}
```

```

        }

        /* Puis ajout du client courant dans la map */
        clients.put( client.getId(), client );
        /* Et enfin (ré)enregistrement de la map en session */
        session.setAttribute( SESSION_CLIENTS, clients );

        /* 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 de la servlet CreationCommande :

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

```
package com.sdzee.tp.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;
import javax.servlet.http.HttpSession;

import com.sdzee.tp.beans.Client;
import com.sdzee.tp.beans.Commande;
import com.sdzee.tp.dao.ClientDao;
import com.sdzee.tp.dao.CommandeDao;
import com.sdzee.tp.dao.DAOFactory;
import com.sdzee.tp.forms.CreationCommandeForm;

public class CreationCommande extends HttpServlet {
    public static final String CONF_DAO_FACTORY = "daofactory";
    public static final String CHEMIN = "chemin";
    public static final String ATT_COMMANDE = "commande";
    public static final String ATT_FORM = "form";
    public static final String SESSION_CLIENTS = "clients";
    public static final String APPLICATION_CLIENTS = "clients";
    "initClients";
    public static final String SESSION_COMMANDES = "commandes";
    public static final String APPLICATION_COMMANDES = "commandes";
    "initCommandes";

    public static final String VUE_SUCCES = "/WEB-INF/afficherCommande.jsp";
    public static final String VUE_FORM = "/WEB-INF/creerCommande.jsp";

    private ClientDao clientDao;
    private CommandeDao commandeDao;

    public void init() throws ServletException {
        /* Récupération d'une instance de nos DAO Client et
        Commande */
        this.clientDao = ( (DAOFactory)
getServletContext().getAttribute( CONF_DAO_FACTORY )
).getClientDao();
```

```
        this.commandeDao = ( DAOFactory )
getServletContext().getAttribute( CONF_DAO_FACTORY )
).getCommandeDao();
    }

    public void doGet( HttpServletRequest request,
HttpServletResponse response ) throws ServletException, IOException
{
    /* À la réception d'une requête GET, simple affichage du
formulaire */
    this.getServletContext().getRequestDispatcher( VUE_FORM
).forward( request, response );
}

public void doPost( HttpServletRequest request,
HttpServletResponse response ) throws ServletException, IOException
{
    /*
    * Lecture du paramètre 'chemin' passé à la servlet via la
déclaration
    * dans le web.xml
    */
    String chemin = this.getServletConfig().getInitParameter(
CHEMIN );

    /* Préparation de l'objet formulaire */
    CreationCommandeForm form = new CreationCommandeForm(
clientDao, commandeDao );

    /* Traitement de la requête et récupération du bean en
résultant */
    Commande commande = form.creerCommande( request, chemin );

    /* Ajout du bean et de l'objet métier à l'objet requête */
request.setAttribute( ATT_COMMANDE, commande );
request.setAttribute( ATT_FORM, form );

    /* Si aucune erreur */
    if ( form.getErreurs().isEmpty() ) {
        /* Alors récupération de la map des clients dans la
session */
        HttpSession session = request.getSession();
        Map<Long, Client> clients = (HashMap<Long, Client>)
session.getAttribute( SESSION_CLIENTS );
        /* Si aucune map n'existe, alors initialisation d'une
nouvelle map */
        if ( clients == null ) {
            clients = new HashMap<Long, Client>();
        }
        /* Puis ajout du client de la commande courante dans la
map */
        clients.put( commande.getClient().getId(),
commande.getClient() );
        /* Et enfin (ré)enregistrement de la map en session */
        session.setAttribute( SESSION_CLIENTS, clients );
    }

    /* Ensuite récupération de la map des commandes dans la
session */
    Map<Long, Commande> commandes = (HashMap<Long,
Commande>) session.getAttribute( SESSION_COMMANDES );
    /* Si aucune map n'existe, alors initialisation d'une
nouvelle map */
    if ( commandes == null ) {
        commandes = new HashMap<Long, Commande>();
    }
    /* Puis ajout de la commande courante dans la map */
    commandes.put( commande.getId(), commande );
    /* Et enfin (ré)enregistrement de la map en session */
    session.setAttribute( SESSION_COMMANDES, commandes );
}
```

```
        /* 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 de la servlet SuppressionClient :

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

```
package com.sdzee.tp.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;
import javax.servlet.http.HttpSession;

import com.sdzee.tp.beans.Client;
import com.sdzee.tp.dao.ClientDao;
import com.sdzee.tp.dao.DAOException;
import com.sdzee.tp.dao.DAOFactory;

public class SuppressionClient extends HttpServlet {
    public static final String CONF_DAO_FACTORY = "daofactory";
    public static final String PARAM_ID_CLIENT = "idClient";
    public static final String SESSION_CLIENTS = "clients";

    public static final String VUE = "/listeClients";

    private ClientDao clientDao;

    public void init() throws ServletException {
        /* Récupération d'une instance de notre DAO Utilisateur */
        this.clientDao = ((DAOFactory)
getServletContext().getAttribute( CONF_DAO_FACTORY )
).getClientDao();
    }

    public void doGet( HttpServletRequest request,
HttpServletResponse response ) throws ServletException, IOException
{
        /* Récupération du paramètre */
        String idClient = getValeurParametre( request,
PARAM_ID_CLIENT );
        Long id = Long.parseLong( idClient );

        /* Récupération de la Map des clients enregistrés en
session */
        HttpSession session = request.getSession();
        Map<Long, Client> clients = (HashMap<Long, Client>)
session.getAttribute( SESSION_CLIENTS );

        /* Si l'id du client et la Map des clients ne sont pas
vides */
        if ( id != null && clients != null ) {
            try {
                Client client = clientDao.getClient( id );
                clients.remove( id );
                clientDao.supprimerClient( client );
                session.setAttribute( SESSION_CLIENTS, clients );
                response.sendRedirect( VUE );
            } catch ( DAOException e ) {
                response.sendRedirect( "erreurs/daoException.html" );
            }
        }
    }
}
```

```

        /* Alors suppression du client de la BDD */
        clientDao.supprimer( clients.get( id ) );
        /* Puis suppression du client de la Map */
        clients.remove( id );
    } catch ( DAOException e ) {
        e.printStackTrace();
    }
    /* Et remplacement de l'ancienne Map en session par la
nouvelle */
    session.setAttribute( SESSION_CLIENTS, clients );
}

/* Redirection vers la fiche récapitulative */
response.sendRedirect( request.getContextPath() + VUE );
}

/*
* Méthode utilitaire qui retourne null si un paramètre est vide, et
son
* contenu sinon.
*/
private static String getValeurParametre( HttpServletRequest
request, String nomChamp ) {
    String valeur = request.getParameter( nomChamp );
    if ( valeur == null || valeur.trim().length() == 0 ) {
        return null;
    } else {
        return valeur;
    }
}

```

Code de la servlet SuppressionCommande :

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

```
package com.sdzee.tp.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;
import javax.servlet.http.HttpSession;

import com.sdzee.tp.beans.Commande;
import com.sdzee.tp.dao.CommandeDao;
import com.sdzee.tp.dao.DAOException;
import com.sdzee.tp.dao.DAOFactory;

public class SuppressionCommande extends HttpServlet {
    public static final String CONF_DAO_FACTORY = "daofactory";
    public static final String PARAM_ID_COMMANDE = "idCommande";
    public static final String SESSION_COMMANDES = "commandes";

    public static final String VUE =
"/listeCommandes";

    private CommandeDao commandeDao;

    public void init() throws ServletException {
        /* Récupération d'une instance de notre DAO Utilisateur */
        this.commandeDao = ( (DAOFactory)
getServletContext().getAttribute( CONF_DAO_FACTORY )
```

```

) .getCommandeDao() ;
}

public void doGet( HttpServletRequest request,
HttpServletResponse response ) throws ServletException, IOException
{
    /* Récupération du paramètre */
    String idCommande = getValeurParametre( request,
PARAM_ID_COMMANDE );
    Long id = Long.parseLong( idCommande );

    /* Récupération de la Map des commandes enregistrées en
session */
    HttpSession session = request.getSession();
    Map<Long, Commande> commandes = (HashMap<Long, Commande>)
session.getAttribute( SESSION_COMMANDES );

    /* Si l'id de la commande et la Map des commandes ne sont
pas vides */
    if ( id != null && commandes != null ) {
        try {
            /* Alors suppression de la commande de la BDD */
            commandeDao.supprimer( commandes.get( id ) );
            /* Puis suppression de la commande de la Map */
            commandes.remove( id );
        } catch ( DAOException e ) {
            e.printStackTrace();
        }
        /* Et remplacement de l'ancienne Map en session par la
nouvelle */
        session.setAttribute( SESSION_COMMANDES, commandes );
    }

    /* Redirection vers la fiche récapitulative */
    response.sendRedirect( request.getContextPath() + VUE );
}

/*
 * Méthode utilitaire qui retourne null si un paramètre est vide, et
son
 * contenu sinon.
*/
private static String getValeurParametre( HttpServletRequest
request, String nomChamp ) {
    String valeur = request.getParameter( nomChamp );
    if ( valeur == null || valeur.trim().length() == 0 ) {
        return null;
    } else {
        return valeur;
    }
}
}
}

```

Code du filtre

Code du filtre de préchargement :

Code : Java - com.sdzee.tp.filters.PrechargementFilter

```

package com.sdzee.tp.filters;

import java.io.IOException;
import java.util.HashMap;
import java.util.List;

```

```
import java.util.Map;

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.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;

import com.sdzee.tp.beans.Client;
import com.sdzee.tp.beans.Commande;
import com.sdzee.tp.dao.ClientDao;
import com.sdzee.tp.dao.CommandeDao;
import com.sdzee.tp.dao.DAOFactory;

public class PrechargementFilter implements Filter {
    public static final String CONF_DAO_FACTORY = "daofactory";
    public static final String ATT_SESSION_CLIENTS = "clients";
    public static final String ATT_SESSION_COMMANDES = "commandes";

    private ClientDao clientDao;
    private CommandeDao commandeDao;

    public void init( FilterConfig config ) throws ServletException
    {
        /* Récupération d'une instance de nos DAO Client et
        Commande */
        this.clientDao = ( DAOFactory )
config.getServletContext().getAttribute( CONF_DAO_FACTORY )
).getClientDao();
        this.commandeDao = ( DAOFactory )
config.getServletContext().getAttribute( CONF_DAO_FACTORY ) )
            .getCommandeDao();
    }

    public void doFilter( ServletRequest req, ServletResponse res,
FilterChain chain ) throws IOException,
ServletException {
        /* Cast de l'objet request */
        HttpServletRequest request = (HttpServletRequest) req;

        /* Récupération de la session depuis la requête */
        HttpSession session = request.getSession();

        /*
        * Si la map des clients n'existe pas en session, alors l'utilisateur
        * se
        * connecte pour la première fois et nous devons précharger en
        * session
        * les infos contenues dans la BDD.
        */
        if ( session.getAttribute( ATT_SESSION_CLIENTS ) == null ) {
            /*
            * Récupération de la liste des clients existants, et enregistrement
            * en session
            */
            List<Client> listeClients = clientDao.lister();
            Map<Long, Client> mapClients = new HashMap<Long,
Client>();
            for ( Client client : listeClients ) {
                mapClients.put( client.getId(), client );
            }
            session.setAttribute( ATT_SESSION_CLIENTS, mapClients );
        }

        /*
        * De même pour la map des commandes
        */
    }
}
```

```

        if ( session.getAttribute( ATT_SESSION_COMMANDES ) == null )
    {
        /*
        * Récupération de la liste des commandes existantes, et
        * enregistrement en session
        */
        List<Commande> listeCommandes = commandeDao.lister();
        Map<Long, Commande> mapCommandes = new HashMap<Long,
Commande>();
        for ( Commande commande : listeCommandes ) {
            mapCommandes.put( commande.getId(), commande );
        }
        session.setAttribute( ATT_SESSION_COMMANDES,
mapCommandes );
    }

    /* Pour terminer, poursuite de la requête en cours */
chain.doFilter( request, res );
}

public void destroy() {
}
}

```

Code des JSP

Code de listerClients.jsp :

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

```

<%@ page pageEncoding="UTF-8" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Liste des clients existants</title>
        <link type="text/css" rel="stylesheet" href="" />
    </head>
    <body>
        <c:import url="/inc/menu.jsp" />
        <div id="corps">
            <c:choose>
                <%-- Si aucun client n'existe en session, affichage d'un
message par défaut. --%>
                <c:when test="${ empty sessionScope.clients }">
                    <p class="erreur">Aucun client enregistré.</p>
                </c:when>
                <%-- Sinon, affichage du tableau. --%>
                <c:otherwise>
                    <table>
                        <tr>
                            <th>Nom</th>
                            <th>Prénom</th>
                            <th>Adresse</th>
                            <th>Téléphone</th>
                            <th>Email</th>
                            <th>Image</th>
                            <th class="action">Action</th>
                        </tr>
                        <%-- Parcours de la Map des clients en session, et
utilisation de l'objet varStatus. --%>
                        <c:forEach items="${ sessionScope.clients }"
var="mapClients" varStatus="boucle">

```

```

<%-- Simple test de parité sur l'index de parcours,
pour alterner la couleur de fond de chaque ligne du tableau. --%>
<tr class="\${boucle.index % 2 == 0 ? 'pair' :
'impaire' }">
    <%-- Affichage des propriétés du bean Client,
qui est stocké en tant que valeur de l'entrée courante de la map --
-%>
    <td><c:out value="\${ mapClients.value.nom
}" /></td>
    <td><c:out value="\${ mapClients.value.prenom
}" /></td>
    <td><c:out value="\${ mapClients.value.adresse
}" /></td>
    <td><c:out value="\${ mapClients.value.telephone
}" /></td>
    <td><c:out value="\${ mapClients.value.email
}" /></td>
    <td>
        <%-- On ne construit et affiche un lien vers
l'image que si elle existe. --%>
        <c:if test="\${ !empty mapClients.value.image
}">
            <c:set var="image"><c:out value="\${ mapClients.value.image
}" /></c:set>
            <a href=<c:url value="/images/\${ image
}" />">Voir</a>
        </c:if>
    </td>
    <%-- Lien vers la servlet de suppression, avec
passage du nom du client - c'est-à-dire la clé de la Map - en
paramètre grâce à la balise <c:param/>. --%>
    <td class="action">
        <a href=<c:url
value="/suppressionClient"><c:param name="idClient" value="\${ mapClients.key
}" /></c:url>">
            <img src=<c:url
value="/inc/supprimer.png"/>" alt="Supprimer" />
        </a>
    </td>
</tr>
</c:forEach>
</table>
</c:otherwise>
</c:choose>
</div>
</body>
</html>

```

Code de listerCommandes.jsp :

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

```

<%@ page pageEncoding="UTF-8" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib prefix="joda" uri="http://www.joda.org/joda/time/tags" %>
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Liste des commandes existantes</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">

```

```

<c:choose>
    <%-- Si aucune commande n'existe en session, affichage
d'un message par défaut. --%>
    <c:when test="${ empty sessionScope.commandes }">
        <p class="erreur">Aucune commande enregistrée.</p>
    </c:when>
    <%-- Sinon, affichage du tableau. --%>
    <c:otherwise>
        <table>
            <tr>
                <th>Client</th>
                <th>Date</th>
                <th>Montant</th>
                <th>Mode de paiement</th>
                <th>Statut de paiement</th>
                <th>Mode de livraison</th>
                <th>Statut de livraison</th>
                <th class="action">Action</th>
            </tr>
            <%-- Parcours de la Map des commandes en session, et
utilisation de l'objet varStatus. --%>
            <c:forEach items="${ sessionScope.commandes }"
var="mapCommandes" varStatus="boucle">
                <%-- Simple test de parité sur l'index de parcours,
pour alterner la couleur de fond de chaque ligne du tableau. --%>
                <tr class="${boucle.index % 2 == 0 ? 'pair' :
'impair'}">
                    <%-- Affichage des propriétés du bean Commande,
qui est stocké en tant que valeur de l'entrée courante de la map -
-%>
                    <td><c:out value="${
mapCommandes.value.client.prenom } ${ mapCommandes.value.client.nom
}" /></td>
                    <td><joda:format value="${
mapCommandes.value.date }" pattern="dd/MM/yyyy HH:mm:ss"/></td>
                    <td><c:out value="${ mapCommandes.value.montant
}" /></td>
                    <td><c:out value="${
mapCommandes.value.modePaiement }"/></td>
                    <td><c:out value="${
mapCommandes.value.statutPaiement }"/></td>
                    <td><c:out value="${
mapCommandes.value.modeLivraison }"/></td>
                    <td><c:out value="${
mapCommandes.value.statutLivraison }"/></td>
                    <%-- Lien vers la servlet de suppression, avec
passage de la date de la commande - c'est-à-dire la clé de la Map -
en paramètre grâce à la balise <c:param/>. --%>
                    <td class="action">
                        <a href=<c:url
value="/suppressionCommande"><c:param name="idCommande" value="${
mapCommandes.key }" /></c:url>">
                            <img src=<c:url
value="/inc/supprimer.png"/> alt="Supprimer" />
                        </a>
                    </td>
                </tr>
            </c:forEach>
        </table>
    </c:otherwise>
</c:choose>
</div>
</body>
</html>

```

Code de creerCommande.jsp :

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

```

<%@ page pageEncoding="UTF-8" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib prefix="joda" uri="http://www.joda.org/joda/time/tags" %>
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Création d'une commande</title>
        <link type="text/css" rel="stylesheet" href="" />
    </head>
    <body>
        <c:import url="/inc/menu.jsp" />
        <div>
            <form method="post" action="">
                enctype="multipart/form-data">
                    <fieldset>
                        <legend>Informations client</legend>
                        <%-- Si et seulement si la Map des clients en session n'est pas vide, alors on propose un choix à l'utilisateur --%>
                        <c:if test="${ !empty sessionScope.clients }">
                            <label for="choixNouveauClient">Nouveau client ? <span class="requis">*</span></label>
                            <input type="radio" id="choixNouveauClient" name="choixNouveauClient" value="nouveauClient" checked /> Oui
                            <input type="radio" id="choixNouveauClient" name="choixNouveauClient" value="ancienClient" /> Non
                            <br/><br />
                        </c:if>

                        <c:set var="client" value="${ commande.client }" scope="request" />
                        <div id="nouveauClient">
                            <c:import url="/inc/inc_client_form.jsp" />
                        </div>

                        <%-- Si et seulement si la Map des clients en session n'est pas vide, alors on crée la liste déroulante --%>
                        <c:if test="${ !empty sessionScope.clients }">
                            <div id="ancienClient">
                                <select name="listeClients" id="listeClients">
                                    <option value="">Choisissez un client...</option>
                                    <%-- Boucle sur la map des clients --%>
                                    <c:forEach items="${ sessionScope.clients }" var="mapClients">
                                        <%-- L'expression EL ${mapClients.value} permet de cibler l'objet Client stocké en tant que valeur dans la Map, et on cible ensuite simplement ses propriétés nom et prenom comme on le ferait avec n'importe quel bean. --%>
                                        <option value="${ mapClients.key }">${ mapClients.value.prenom } ${ mapClients.value.nom }</option>
                                    </c:forEach>
                                </select>
                            </div>
                        </c:if>
                    </fieldset>
                    <fieldset>
                        <legend>Informations commande</legend>
                        <label for="dateCommande">Date <span class="requis">*</span></label>
                        <input type="text" id="v" name="dateCommande" value="" size="30" maxlength="30" disabled />
                        <span class="erreur">${ form.erreurs['dateCommande'] }</span>
                        <br />
                    </fieldset>
                </form>
            </div>
        </body>
    </html>

```

```

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

<%-- Inclusion de la bibliothèque jQuery. Vous trouverez des cours sur
JavaScript et jQuery aux adresses suivantes :
    - http://www.siteduzero.com/tutoriel-3-309961-dynamisez-vos-sites-web-avec-javascript.html
    - http://www.siteduzero.com/tutoriel-3-659477-un-site-web-dynamique-avec-jquery.html

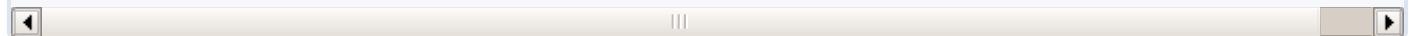
Si vous ne souhaitez pas télécharger et ajouter jQuery à votre
projet, vous pouvez utiliser la version fournie directement en ligne par Google
:
<script
src="https://ajax.googleapis.com/ajax/libs/jquery/1.8.0/jquery.min.js"></script>
--%>
<script src=">"></script>

<%-- Petite fonction jQuery permettant le remplacement de la première
partie du formulaire par la liste déroulante, au clic sur le bouton radio. --%>
<script>
```

```

        jQuery(document).ready(function(){
            /* 1 - Au lancement de la page, on cache le bloc d'éléments du
formulaire correspondant aux clients existants */
            $("div#ancienClient").hide();
            /* 2 - Au clic sur un des deux boutons radio "choixNouveauClient", on
affiche le bloc d'éléments correspondant (nouveau ou ancien client) */
            $("input[name=choixNouveauClient]:radio").click(function(){
                $("div#nouveauClient").hide();
                $("div#ancienClient").hide();
                var divId = jQuery(this).val();
                $("div#+divId").show();
            });
        });
    </script>
</body>
</html>

```



Code de afficherCommande.jsp :

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

```

<%@ page pageEncoding="UTF-8" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib prefix="joda" uri="http://www.joda.org/joda/time/tags" %>
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Affichage d'une commande</title>
        <link type="text/css" rel="stylesheet" href="" />
    </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>Image : <c:out value="${ commande.client.image
}" /></p>
            <p>Commande</p>
            <p>Date : <joda:format value="${ commande.date
}" pattern="dd/MM/yyyy HH:mm:ss"/></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>

```

Gérer un pool de connexions avec BoneCP

La solution que nous avons mise en place souffre d'une carence importante : elle manque cruellement d'ambition ! Je n'ai fait qu'effleurer la question lors de notre apprentissage de JDBC, il est maintenant temps de nous y attarder et d'y apporter une solution simple et efficace.

Contexte

Dans le second chapitre de cette partie, je vous ai avertis que l'ouverture d'une connexion avec la BDD avait un coût non négligeable en termes de performances, et que dans une application très fréquentée, il était hors de question de procéder à des ouvertures/fermetures à chaque requête effectuée sans signer l'arrêt de mort de votre serveur ! À quoi faisais-je allusion ? Penchons-nous un instant sur le contexte du problème.

Une application multi-utilisateurs

Une application web est une application centralisée à laquelle tous les utilisateurs accèdent à distance depuis leurs navigateurs. Partant de cette définition simpliste, il est aisément de deviner les limites du système. Si un nombre conséquent d'utilisateurs différents réalisent des actions sur le site dans un intervalle de temps restreint, voire de manière simultanée, le serveur va devoir traiter l'ensemble des requêtes reçues à une vitesse folle pour assurer un certain confort de navigation au visiteur d'une part, et pour ne pas s'écrouler sous la charge d'autre part.

Ceci étant dit, il ne faut pas s'alarmer pour autant : tous les composants d'un serveur d'applications ne sont pas critiques. Sans plus attendre, voyons celui qui pèche le premier...

Le coût d'une connexion à la BDD

Le maillon faible de la chaîne est sans surprise la liaison entre l'application (pour généraliser, disons le conteneur web) et la base de données. C'est ici que la dégradation des performances sera la plus importante, tant par sa rapidité que par sa lourdeur de conséquences.



De quel problème exactement sommes-nous en train de parler ?

L'établissement d'une connexion entre notre application et notre base de données MySQL a un coût en temps non négligeable, qui peut monter jusqu'à plusieurs centaines de millisecondes. Les principales raisons de cette latence sont la nécessité d'établir une connexion TCP/IP entre le conteneur et le SGBD d'une part, avec les contraintes naturelles que cela pose (lenteur du réseau, pare-feu, filtrages, etc.), et le fait que le SGBD va devoir préparer des informations relatives à l'utilisateur entrant à chaque nouvelle connexion.

Alors certes, à l'échelle d'une connexion cela reste tout à fait négligeable : quelques dixièmes de seconde de ralentissement sont presque anecdotiques... Oui, mais imaginez maintenant que cinquante utilisateurs effectuent simultanément des actions faisant intervenir une communication avec la base de données sur notre site : notre SGBD doit alors créer autant de nouvelles connexions, et nous passons ainsi de quelques dixièmes de secondes à plusieurs secondes de latence ! Et ça bien évidemment, c'est inacceptable. Sans parler de l'attente côté utilisateur, votre application va de son côté continuer à recevoir des requêtes et à tenter d'établir des connexions avec votre base, jusqu'au moment où la limite sera atteinte et votre SGBD cessera de répondre, causant tout bonnement le plantage lamentable de votre application.

En revanche, le reste des opérations effectuées sur une base de données est très rapide ! Typiquement, la plupart de celles basées sur une connexion déjà ouverte s'effectuent en quelques millisecondes seulement.

La structure actuelle de notre solution

Pour couronner le tout, la solution que nous avons mise en place fonctionne très bien pour effectuer des tests sur notre poste de développement, mais n'est pas du tout adaptée à une utilisation en production ! Eh oui, réfléchissez bien : alors que nous aurions pu nous contenter d'ouvrir une seule connexion et de la partager à l'ensemble des méthodes d'un DAO, nous procédons à l'ouverture/fermeture d'une connexion à chaque requête effectuée ! Pour vous rafraîchir la mémoire, voici mis à plat le squelette basique que nous avons appliqué :

Code : Java - Structure de base des méthodes de nos DAO

```
Connection connexion = null;
```

```

PreparedStatement preparedStatement = null;
ResultSet resultat = null;

try {
    connexion = daoFactory.getConnection();
    preparedStatement = connexion.prepareStatement( REQUETE_SQL );
    ...
    resultat = preparedStatement.executeQuery();
    ...
} finally {
    if (resultat != null) try { resultat.close(); } catch
(SQLException ignore) {}
    if (preparedStatement != null) try { preparedStatement.close(); }
} catch (SQLException ignore) {}
    if (connexion != null) try { connexion.close(); } catch
(SQLException ignore) {}
}

```

 Dans ce cas, pourquoi vous avoir fait développer une structure si peu adaptée à une utilisation en conditions réelles ?

En réalité, le tableau n'est pas aussi sombre qu'il n'y paraît. Certes, l'obtention d'une nouvelle connexion à chaque requête est une catastrophe en termes de performances. Mais le bon point, c'est que notre code est correctement organisé et découpé ! Regardez de plus près, nos DAO ne font pas directement appel à la méthode JDBC `DriverManager.getConnection()` pour établir une connexion. Ils le font par l'intermédiaire de la méthode `getConnection()` de notre **DAOFactory** :

Code : Java - DAOFactory.getConnection()

```

/* Méthode chargée de fournir une connexion à la base de données */
/* package */Connection getConnection() throws SQLException {
    return DriverManager.getConnection( url, username, password );
}

```

Ainsi notre solution est clairement mauvaise pour le moment, mais nous sentons bien qu'en modifiant la manière dont notre **DAOFactory** obtient une connexion, nous pouvons améliorer grandement la situation, et ce sans rien avoir à changer dans nos DAO ! Bref, je ne vous ai pas fait coder n'importe quoi... 😊

Principe Réutilisation des connexions

Afin d'alléger la charge que le SGBD doit actuellement supporter à chaque requête, nous allons utiliser une technique très simple : nous allons précharger un certain nombre de connexions à la base de données, et les réutiliser. L'expression employée pour nommer cette pratique est le « **connection pooling** », souvent très sauvagement francisée « **pool de connexions** ».

 Comment fonctionne ce mécanisme de réutilisation ?

Pour faire simple, le pool de connexions va pré-initialiser un nombre donné de connexions au SGBD lorsque l'application démarre. Autrement dit, il va créer plusieurs objets **Connection** et les garder ouverts et bien au chaud.

Ensuite, le pool de connexions va se charger de distribuer ses objets **Connection** aux méthodes de l'application qui en ont besoin. Concrètement, cela signifie que ces méthodes ne vont plus faire appel à `DriverManager.getConnection()`, mais plutôt à quelque chose comme `pool.getConnection()`.

Enfin, puisque l'objectif du système est de partager un nombre prédéfini de ressources, un appel à la méthode `Connection.close()` ne devra bien entendu pas provoquer la fermeture réelle d'une connexion ! En lieu et place, c'est tout simplement un renvoi dans le pool de l'objet **Connection** qui va avoir lieu. De cette manière, et seulement de cette manière, la boucle est bouclée : l'objet **Connection** inutilisé retourne à la source, et est alors prêt à être à nouveau distribué.

De manière imagée, vous pouvez voir le pool comme un système de location de véhicules. Il dispose d'une flotte de véhicules dont il est le seul à pouvoir autoriser la sortie, sortie qu'il autorise à chaque demande d'un client. Cette sortie est par nature prévue pour être de courte durée : si un ou plusieurs clients ne rendent pas le véhicule rapidement, voire ne le rendent pas du tout, alors petit à petit la flotte de véhicules disponibles va se réduire comme peau de chagrin, et le pool n'aura bientôt plus aucun véhicule disponible...



Bref, la libération des ressources dont je vous ai parlé avec insistance dans les précédents chapitres prend ici tout son intérêt : elle est la clé de voûte du système. Si une ressource inutilisée ne retourne pas au pool, les performances de l'application vont inévitablement se dégrader.

Remplacement du DriverManager par une DataSource

Maintenant que nous savons comment cela fonctionne en théorie, intéressons-nous à la pratique. Jusqu'à présent, nous avons utilisé le `DriverManager` du package `java.sql` pour obtenir une connexion à notre base de données. Or nous venons de découvrir qu'afin de mettre en place un pool de connexions, il ne faut plus passer par cet objet...



Quel est le problème avec le `DriverManager` ?

Sur une application Java classique, avec un utilisateur unique et donc une seule connexion vers une base de données, il convient très bien. Mais une application Java EE est multi-threads et multi-utilisateurs, et sous ces contraintes cet objet ne convient plus. Dans une application Java EE en conditions réelles, plusieurs connexions parallèles sont ouvertes avec la base de données et pour les raisons évoquées un peu plus tôt, il n'est pas envisageable d'ouvrir des connexions à la volée pour chaque objet ou méthode agissant sur la base.

Nous avons besoin d'une gestion des ressources plus efficace, et notre choix va se porter sur l'objet `DataSource` du nouveau package `javax.sql`. Il s'agit en réalité d'une interface qu'Oracle recommande d'utiliser en lieu de place du `DriverManager`, et ce peu importe le cas d'utilisation.



Dans ce cas, pourquoi ne pas avoir directement appris à manipuler une `DataSource` ?

Tout simplement parce qu'encore aujourd'hui, l'objet `DriverManager` est très répandu dans beaucoup de projets de faible ou moyenne envergure, et il est préférable que vous sachiez comment il fonctionne. En outre, vous allez bientôt découvrir que la manipulation d'une `DataSource` n'est pas si différente !

Choix d'une implémentation

Une `DataSource` n'est qu'une interface, il est donc nécessaire d'en écrire une implémentation. Rassurez-vous, nous n'allons pas nous occuper de cette tâche : il existe plusieurs bibliothèques, libres et gratuites, qui ont été créées par des équipes de développeurs expérimentés et validées par des années d'utilisation. Sans être exhaustif, voici une liste des solutions les plus couramment rencontrées :

- Apache DBCP
- BoneCP
- c3p0
- DBPool

Le seul petit souci, c'est que toutes ces bibliothèques ne disposent pas d'une communauté active, et toutes n'offrent pas les mêmes performances. Je vous épargne la recherche d'informations sur chacune des solutions et leurs *benchmarks* respectifs, et vous annonce que nous allons utiliser **BoneCP** !

Quoi qu'il en soit, peu importe la solution que vous choisissez dans vos projets, le principe de base ne change pas et le processus de configuration que nous allons apprendre dans le paragraphe suivant reste sensiblement le même.

Mise en place

Ajout des jar au projet

Comme pour toute bibliothèque, il va nous falloir placer l'archive jar de la solution dans notre environnement. De la même

manière que nous l'avions fait pour le driver JDBC, nous allons déposer le jar de BoneCP, téléchargeable en cliquant ici, dans le répertoire /lib de Tomcat.

En outre, la page listant les conditions requises à un bon fonctionnement de la solution nous signale qu'il faut également inclure les bibliothèques SLF4J et Google Guava à notre projet, car la solution BoneCP en dépend. Vous devez donc télécharger les deux archives en cliquant sur ce lien pour SLF4J et sur celui-ci pour Guava, et les placer sous le répertoire /WEB-INF/lib aux côtés de l'archive de BoneCP.

Prise en main de la bibliothèque

Nous devons ensuite étudier la documentation de BoneCP pour comprendre comment tout cela fonctionne. Il existe plusieurs méthodes de configuration, chacune propre à un usage particulier. Ainsi, il est possible de paramétrer l'outil à la main, via un DataSource, pour une utilisation avec le framework Spring ou encore pour une utilisation avec le duo de frameworks Spring et Hibernate.

Quoiqu'il en soit, peu importe le contexte, les propriétés accessibles sont sensiblement les mêmes et nous allons toujours retrouver :

- **jdbcUrl**, contenant l'URL de connexion JDBC ;
- **username**, contenant le nom d'utilisateur du compte à utiliser sur la BDD ;
- **password**, contenant le mot de passe du compte à utiliser sur la BDD ;
- **partitionCount**, contenant un entier de 1 à N symbolisant le nombre de partitions du pool. C'est un paramètre spécifique à BoneCP, qui n'existe pas dans des solutions comme c3p0 ou DBCP ;
- **maxConnectionsPerPartition**, contenant le nombre maximum de connexions pouvant être créées par partition. Concrètement, cela signifie que si cette valeur vaut 5 et qu'il y a 3 partitions, alors il y aura en tout 15 connexions uniques disponibles et partagées via le pool. À noter que BoneCP ne va pas initialiser l'ensemble de ces connexions d'une traite, mais va commencer par en créer autant que spécifié dans la propriété **minConnectionsPerPartition**, puis il va graduellement augmenter le nombre de connexions disponibles au fur et à mesure que la charge va monter ;
- **minConnectionsPerPartition**, contenant le nombre minimum de connexions par partition.

En ce qui nous concerne, nous allons dans le cadre de ce cours mettre en place un pool de connexions à la main. Toujours en suivant la documentation de BoneCP, nous apprenons qu'il faut passer par l'objet **BoneCPConfig** pour initialiser les différentes propriétés que nous venons de découvrir. Il suffit ensuite de créer un pool via l'objet **BoneCP**. Voici le code d'exemple :

Code : Java - Exemple de mise en place d'un pool de connexions

```
Class.forName( "com.mysql.jdbc.Driver" ); // chargement du driver JDBC (ici MySQL)
BoneCPConfig config = new BoneCPConfig(); // création d'un objet BoneCPConfig
config.setJdbcUrl( url ); // définition de l'URL JDBC
config.setUsername( nomUtilisateur ); // définition du nom d'utilisateur
config.setPassword( motDePasse ); // définition du mot de passe

config.setMinConnectionsPerPartition( 5 ); // définition du nombre min de connexions par partition
config.setMaxConnectionsPerPartition( 10 ); // définition du nombre max de connexions par partition
config.setPartitionCount( 2 ); // définition du nombre de partitions

BoneCP connectionPool = new BoneCP( config ); // création du pool à partir de l'objet BoneCPConfig

...
```

Voilà tout ce que nous avons besoin de savoir. Pour le reste, la documentation est plutôt exhaustive. 😊

Modification de la DAOFactory

Maintenant que nous savons comment procéder, nous devons modifier le code de notre **DAOFactory** pour qu'elle travaille non plus en se basant sur l'objet `DriverManager`, mais sur un pool de connexions. Je vous donne le code pour commencer, et vous explique le tout ensuite :

Code : Java - com.sdzee.dao.DAOFactory

```
package com.sdzee.dao;

import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStream;
import java.sql.Connection;
import java.sql.SQLException;
import java.util.Properties;

import com.jolbox.bonecp.BoneCP;
import com.jolbox.bonecp.BoneCPConfig;

public class DAOFactory {

    private static final String FICHIER_PROPERTIES =
"/com/sdzee/dao/dao.properties";
    private static final String PROPERTY_URL = "url";
    private static final String PROPERTY_DRIVER = "driver";
    private static final String PROPERTY_NOM_UTILISATEUR =
"nomutilisateur";
    private static final String PROPERTY_MOT_DE_PASSE =
"motdepassé";

    /* package */BoneCP connectionPool
= null;

    /* package */DAOFactory( BoneCP connectionPool ) {
        this.connectionPool = connectionPool;
    }

    /*
     * Méthode chargée de récupérer les informations de connexion à la
     * base de
     * données, charger le driver JDBC et retourner une instance de la
     * Factory
     */
    public static DAOFactory getInstance() throws
DAOConfigurationException {
        Properties properties = new Properties();
        String url;
        String driver;
        String nomUtilisateur;
        String motDePasse;
        BoneCP connectionPool = null;

        ClassLoader classLoader =
Thread.currentThread().getContextClassLoader();
        InputStream fichierProperties =
classLoader.getResourceAsStream( FICHIER_PROPERTIES );

        if ( fichierProperties == null ) {
            throw new DAOConfigurationException( "Le fichier
properties " + FICHIER_PROPERTIES + " est introuvable." );
        }

        try {
            properties.load( fichierProperties );
            url = properties.getProperty( PROPERTY_URL );
            driver = properties.getProperty( PROPERTY_DRIVER );
            nomUtilisateur = properties.getProperty(
PROPERTY_NOM_UTILISATEUR );
            motDePasse = properties.getProperty(
```

```

PROPERTY_MOT_DE_PASSE );
    } catch ( FileNotFoundException e ) {
        throw new DAOConfigurationException( "Le fichier
properties " + FICHIER_PROPERTIES + " est introuvable.", e );
    } catch ( IOException e ) {
        throw new DAOConfigurationException( "Impossible de
charger le fichier properties " + FICHIER_PROPERTIES, e );
    }

    try {
        Class.forName( driver );
    } catch ( ClassNotFoundException e ) {
        throw new DAOConfigurationException( "Le driver est
introuvable dans le classpath.", e );
    }

    try {
        /*
         * Création d'une configuration de pool de connexions via l'objet
         * BoneCPConfig et les différents setters associés.
        */
        BoneCPConfig config = new BoneCPConfig();
        /* Mise en place de l'URL, du nom et du mot de passe */
        config.setJdbcUrl( url );
        config.setUsername( nomUtilisateur );
        config.setPassword( motDePasse );
        /* Paramétrage de la taille du pool */
        config.setMinConnectionsPerPartition( 5 );
        config.setMaxConnectionsPerPartition( 10 );
        config.setPartitionCount( 2 );
        /* Création du pool à partir de la configuration, via
l'objet BoneCP */
        connectionPool = new BoneCP( config );
    } catch ( SQLException e ) {
        e.printStackTrace();
        throw new DAOConfigurationException( "Erreur de
configuration du pool de connexions.", e );
    }
    /*
     * Enregistrement du pool créé dans une variable d'instance via un
appel
     * au constructeur de DAOFactory
    */
    DAOFactory instance = new DAOFactory( connectionPool );
    return instance;
}

/* Méthode chargée de fournir une connexion à la base de
données */
/* package */ Connection getConnection() throws SQLException {
    return connectionPool.getConnection();
}

/*
 * Méthodes de récupération de l'implémentation des différents DAO
(un seul
* pour le moment)
*/
public UtilisateurDao getUtilisateurDao() {
    return new UtilisateurDaoImpl( this );
}
}

```

Comme vous pouvez le constater, le principe est très légèrement différent. Auparavant lors d'un appel à `getInstance()`, nous enregistriions les informations de connexion à la base de données dans des variables d'instance et nous les réutilisions à chaque appel à `getConnection()`. Dorénavant, lors d'un appel à `getInstance()` nous procédons directement à l'initialisation du pool de connexions, et nous enregistrons uniquement le pool ainsi obtenu dans une variable d'instance, qui est alors réutilisée à chaque appel à `getConnection()`.

Ainsi, seuls quelques éléments changent dans le code :

- le constructeur se base maintenant sur l'objet BoneCP, ici aux lignes 23 à 25 ;
- la lecture des informations depuis le fichier **Properties** ne change pas ;
- le chargement du driver ne change pas ;
- avant d'appeler le constructeur, nous procérons aux lignes 64 à 83 à la création du pool de connexions ;
- enfin, la méthode `getConnection()` se base maintenant sur le pool, et non plus sur le **DriverManager** !

Voilà tout ce qu'il est nécessaire de modifier pour mettre en place un pool de connexions dans notre application ! Simple et rapide, n'est-ce pas ? 😊

 Vous voilà devant un autre exemple de l'intérêt de bien organiser et découper le code d'une application. Dans cet exemple, la transparence est totale lors de l'utilisation des connexions, il nous suffit uniquement de modifier la méthode `getConnection()` de la classe **DAOFactory** ! Si nous n'avions pas mis en place cette *Factory* et avions procédé directement à l'ouverture d'une connexion depuis nos servlets ou objets métier, nous aurions dû reprendre l'intégralité du code concerné pour mettre en place notre pool...

À noter également l'importance dans le code existant d'avoir correctement fermé/libéré les ressources utilisées (et notamment les objets `Connection`) lors des requêtes dans nos DAO, car si les connexions utilisées n'étaient pas systématiquement renvoyées au pool, alors nous nous heurterions très rapidement à un problème d'épuisement de stock, menant inéluctablement au plantage du serveur !

Vérifications

Une fois toutes les modifications effectuées, il ne vous reste plus qu'à vérifier que votre code compile correctement et à redémarrer Tomcat ! Effectuez alors quelques tests de routine sur la page <http://localhost:8080/pro/inscription> pour vous assurer du bon fonctionnement de votre application :

- essayez de vous inscrire en oubliant de confirmer votre mot de passe ;
- essayez de vous inscrire avec une adresse mail déjà utilisée ;
- enfin, inscrivez-vous avec des informations valides.

Configuration fine du pool

Ce dernier paragraphe va peut-être vous décevoir, mais il n'existe pas de règles empiriques à appliquer pour « bien » configurer un pool de connexions. C'est un processus qui est fortement lié à la fois au nombre moyen d'utilisateurs simultanés sur le site, à la complexité des requêtes effectuées sur la base, aux capacités matérielles du serveur, au SGBD utilisé, à la qualité de service souhaitée, etc.

En somme, la configuration fine d'un pool s'effectue au cas par cas. Cela passe par une série de tests grandeur nature, durant lesquels un certain nombre de connexions et utilisations simultanées sont simulées en accord avec les spécifications du projet, afin de vérifier comment réagit la base de données, comment réagit le pool et comment sont impactées les performances globales de l'application en période de charge faible, moyenne et forte, le tout pour une configuration donnée. Si les tests sont convaincants, alors la configuration est validée. Sinon, elle est modifiée - changement du nombre de connexions minimum et maximum, et du nombre de partitions - et une autre série de tests est lancée.

Bref, aller plus loin ici n'a aucun intérêt pédagogique pour nous : nous sommes ici pour apprendre à développer une application, pas pour devenir des experts en déploiement ! 😊

En résumé

- La connexion à une base de données est une étape coûteuse en termes de temps et de performances.
- Il est nécessaire d'initialiser un nombre prédéfini de connexions, et de les partager/distribuer/réutiliser pour chaque requête entrante : c'est le principe du **pool de connexions**.
- Lorsqu'un pool de connexions est en place, un appel à la méthode `connexion.close()` ne ferme pas littéralement une connexion, mais la renvoie simplement au pool.
- La méthode `getConnection()` étant centralisée et définie dans notre Factory, il nous est très aisément de modifier son comportement.

- Un pool de connexions se base sur le principe d'une `DataSource`, objet qu'il est vivement recommandé d'utiliser en lieu et place du `DriverManager`.
 - BoneCP est une solution de pooling très efficace, aisément configurable et intégrable à n'importe quelle application Java EE.

Nous voilà enfin aptes à créer une application de A à Z ! Nous n'avons pas encore pris connaissance des autres moyens existant pour créer et gérer plus simplement une application, mais avec notre bagage nous sommes d'ores et déjà capables de produire une application complète et fonctionnelle.

Partie 6 : Aller plus loin avec JPA et JSF

Dans cette ultime partie, nous allons découvrir deux concepts très utilisés dans les projets en entreprise :

- la persistance de données, intervenant comme son nom l'indique au niveau de la gestion des données. EJB et JPA sont au programme !
- les frameworks MVC, qui imposent une organisation du code en couches bien définie. JSF 2 est celui que nous allons étudier.

Les annotations

Dans ce chapitre, nous allons découvrir les annotations, un système introduit avec Java EE 5 qui va nous permettre de nous débarrasser de notre fichier web.xml !

Présentation

Pour commencer, parce qu'il est toujours utile de s'intéresser aux raisons d'être d'une technologie, nous allons étudier le sujet de manière globale et nous attarder quelques instants sur la théorie. Si vous connaissez déjà cette fonctionnalité, qui comme nous allons le voir existe depuis quelque temps déjà sur la plate-forme Java, vous pouvez survoler cette introduction et passer directement au paragraphe qui traite spécifiquement des annotations sous Java EE.

Une annotation est tout simplement une description d'un élément. Elle peut ainsi s'appliquer à un package, une classe, une interface, un constructeur, une méthode, un champ, un argument de méthode, une variable locale ou encore à une autre annotation.



Quel est l'intérêt de décrire ces différents éléments ?

Pour répondre entièrement à cette question, nous allons aborder différents aspects.

Écrire des méta-données

Vous avez peut-être déjà entendu parler de **méta-données** ou de **méta-programmation**. Il s'agit d'un concept très simple : une méta-donnée désigne une donnée qui donne une information sur une autre donnée. Voilà pour la définition littérale. Intéressons-nous maintenant à l'intérêt de ces méta-données dans le cadre de notre projet !

Premièrement, parlons Java. En réalité, Java a toujours proposé une forme de méta-programmation que vous connaissez tous mais que vous n'avez probablement jamais appelée ainsi : il s'agit de la Javadoc ! Eh oui, la Javadoc n'est, ni plus ni moins, qu'un outil permettant de décrire certaines portions de votre code... Dans ce cas, les méta-données sont utilisées par le développeur, qui va tout simplement lire les informations contenues dans la documentation.

Deuxièmement, abordons le cas de Java EE. Là encore, même constat : la plate-forme a toujours proposé une forme de méta-programmation par le biais du fameux fichier web.xml, dans lequel nous avons saisi les descriptions de certains composants de notre application. Dans ce cas, les méta-données sont utilisées par le conteneur, qui analyse le contenu du fichier au démarrage du serveur afin d'identifier quels composants web agissent sur quelles requêtes.



Ainsi, voici la définition que nous pouvons retenir : les méta-données sont de simples informations accompagnant le code d'une application, et qui peuvent être utilisées à des fins diverses.

Pallier certaines carences

Il y a quelques années, Java ne permettait pas encore de faire de méta-programmation de manière simple. La seule solution qui s'en approchait était la Javadoc et son système de *tags*. Elle permettait uniquement de rédiger de la documentation destinée au développeur. À l'époque, certaines solutions avaient alors vu le jour afin d'en détourner l'usage, notamment XDoclet qui permettait de définir des *tags* Javadoc personnalisés et de générer du code à partir des méta-données saisies dans ce semblant de documentation.

Reconnaissant les manques de la plate-forme à ce niveau et surtout le besoin d'un système plus robuste, plus flexible et surtout homogène, l'équipe en charge de l'évolution de Java a introduit le concept des annotations avec l'avènement du JDK 1.5, en

2004. Par ailleurs, sachez que cette version a probablement été la plus riche de toutes en termes d'apport de nouveautés : les annotations comme nous venons de le voir, mais également la programmation générique, la syntaxe **for** adaptée au parcours de collections, les imports statiques, l'autoboxing, la notation varargs... Bref, les développeurs n'ont pas chômé !

C'est par ailleurs après cette mise à jour majeure du JDK que J2EE est devenu Java EE 5 en 2006, et c'est donc uniquement à partir de cette version que les annotations ont été rendues disponibles pour le développement d'applications web. Dans toutes les versions antérieures de la plate-forme (c'est-à-dire l'ensemble des versions estampillées J2EE), cette fonctionnalité n'existe pas.

Depuis Java EE 6, lancé fin 2009, le support des annotations est total et leur gestion fait partie intégrante du compilateur Java.

Simplifier le développement

La raison d'être des annotations est sans équivoque : elles permettent de simplifier considérablement le processus de développement et de maintenance d'une application. Pourquoi ? Principalement parce qu'une annotation est écrite directement dans le code, au sein d'une classe ou d'une méthode. Ainsi, l'information associée à un élément du code est accessible et visible directement. Il n'est pas nécessaire de se référer à des données écrites dans des fichiers externes. De même, lors de la phase de développement il n'est pas nécessaire de créer et maintenir ces fichiers externes.

Principe

Maintenant que nous savons de quoi il retourne, découvrons comment sont construites les annotations.

Syntaxe sans paramètres

Une annotation, sous sa forme la plus simple, est uniquement constituée d'un mot-clé précédé du signe @. Nous allons prendre pour exemple l'annotation `@Override`, que vous avez déjà pu observer dans le code de notre classe `com.sdzee.dao.UtilisateurDaoImpl` :

Code : Java - Exemple d'annotation Java

```
public class UtilisateurDaoImpl implements UtilisateurDao {  
    /* Implémentation de la méthode trouver() définie dans  
    l'interface UtilisateurDao */  
    @Override  
    public Utilisateur trouver( String email ) throws DAOException {  
        ...  
    }  
    ...  
}
```

Il s'agit là d'une annotation Java qui sert à décrire une méthode, en l'occurrence la méthode `trouver()`. Elle est constituée du caractère @ suivi du mot-clé **Override**, et elle est placée juste avant le début de la méthode.

Pour la petite histoire, sachez que cette annotation est doublement utile :

- elle permet de préciser au compilateur qu'une méthode redéfinit une méthode d'une interface. En cas d'erreur - c'est-à-dire si le nom de la méthode ne correspond à aucune méthode d'aucune interface - alors le compilateur doit prévenir le développeur et éventuellement faire échouer la compilation.
- elle rend le code plus lisible, en différenciant les méthodes redéfinies des autres. Ceci est renforcé par les comportements intuitifs introduits dans la plupart des IDE, qui marquent visuellement telles méthodes.

Je vous montre ici cette annotation en particulier pour illustrer la syntaxe utilisée, mais ne vous y trompez pas, il en existe bien d'autres. Il est d'ailleurs possible de créer ses propres annotations si besoin, mais ceci a uniquement trait au Java et sort par conséquent du cadre de ce cours. Nous, ce qui nous intéresse, c'est le Java EE !

Syntaxe avec paramètres

Il existe une seconde forme d'annotation, qui attend en argument des paramètres. Elle se construit de la manière suivante :

Code : Java - Exemple d'annotation avec paramètres

```
@WebServlet( name="TestServlet", urlPatterns = { "/test", "/ok" } )
```

Explications :

- une annotation peut attendre un ou plusieurs paramètres, séparés par une virgule et placés entre parenthèses juste après l'annotation. Dans cet exemple, les paramètres sont nommés **name** et **urlPatterns** ;
- un paramètre peut attendre une ou plusieurs valeurs :
 - si une seule valeur est attendue, celle-ci est placée entre guillemets et liée au paramètre par le symbole =. Ici, le paramètre **name** attend une unique valeur définie à "TestServlet" ;
 - si plusieurs valeurs sont attendues, celles-ci sont placées entre guillemets, séparées par une virgule et l'ensemble ainsi formé est placé entre accolades, puis lié au paramètre par le symbole = (en somme, la syntaxe d'initialisation d'un tableau). Ici, le paramètre **urlPatterns** reçoit deux valeurs définies respectivement à "/test" et "/ok".

Voilà tout ce qu'il est nécessaire de retenir concernant la syntaxe des annotations.

Avec l'API Servlet 3.0

Nous sommes enfin prêts pour aborder quelques annotations spécifiques à Java EE 6, introduites par l'API Servlet en version 3.0. Celles-ci font toutes partie du package `javax.servlet.annotation`.

WebServlet

La première à laquelle nous allons nous intéresser est celle qui permet de déclarer une servlet : `@WebServlet`. Comme vous pouvez le voir dans [sa documentation](#), elle peut accepter tous les paramètres qu'il est possible de définir dans une section `<servlet>` ou `<servlet-mapping>` du fichier web.xml.

Prenons pour exemple notre servlet **Inscription**. Pour rappel, sa déclaration dans le fichier web.xml était la suivante :

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

```
<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>
```

L'annotation qui remplace littéralement cette description est :

Code : Java - com.sdzee.servlets.Inscription

```
...
@WebServlet( name="Inscription", urlPatterns = "/inscription" )
public class Inscription extends HttpServlet {
  ...
}
```

Remarquez bien les points suivants :

- il faut placer l'annotation juste avant la déclaration de la classe dans le code de votre servlet ;
- il n'est pas nécessaire de préciser le contenu de `<servlet-class>` dans l'annotation : puisque celle-ci se trouve directement dans le code de la servlet, le compilateur sait déjà à quelle classe elle s'applique.

Ajoutez l'annotation à votre servlet, supprimez la déclaration de votre web.xml et redémarrez Tomcat. Vous pourrez alors vérifier

par vous-mêmes que votre application fonctionne exactement comme avant, en vous rendant sur la page <http://localhost:8080/pro/inscription>.

En outre, certains d'entre vous auront peut-être également remarqué que la propriété **name** de la servlet n'est plus utile. Dans le fichier **web.xml**, le champ **<servlet-name>** servait à établir un lien entre les sections **<servlet>** et **<servlet-mapping>**, mais maintenant que nous précisons directement *l'url-pattern* dans l'annotation de la servlet, son nom ne nous sert plus à rien. Par conséquent, il est possible d'écrire l'annotation sous sa forme la plus simple :

Code : Java - com.sdzee.servlets.Inscription

```
...
@.WebServlet( "/inscription" )
public class Inscription extends HttpServlet {
...
```

Ainsi, seul *l'url-pattern* sur lequel est mappée notre servlet est nécessaire à son bon fonctionnement. Là encore, vous pouvez confirmer ce comportement en modifiant l'annotation dans votre servlet **Inscription** et en vérifiant que l'application fonctionne toujours.



Par ailleurs, puisque les méta-données sont maintenant présentes directement dans le code de votre servlet, il n'est plus nécessaire de redémarrer Tomcat à chaque modification du nom de la servlet ou encore de son *url-pattern* : les modifications sont prises en compte presque instantanément !

WebFilter

Très similaire à la précédente, il existe une annotation qui permet de déclarer un filtre : **@WebFilter**. Comme pour sa cousine dédiée à la servlet, vous constaterez dans [sa documentation](#) qu'elle accepte toutes les propriétés définissables depuis une section **<filter>** ou **<filter-mapping>** du fichier **web.xml**.

Prenons pour exemple le filtre **RestrictionFilter** que nous avions mis en place pour tester la restriction d'accès sur un groupe de pages. Pour rappel, sa déclaration dans notre fichier **web.xml** était la suivante :

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>/restreint/*</url-pattern>
</filter-mapping>
```

De la même manière que pour notre servlet **Inscription**, l'annotation qui remplace littéralement cette description peut s'écrire :

Code : Java

```
...
@WebFilter( urlPatterns = "/restreint/*" )
public class RestrictionFilter implements Filter {
...
```

Dans notre application, nous avions désactivé le filtre pour ne pas être embêtés dans les autres exemples du cours. Si vous le souhaitez, vous pouvez remettre en place le filtre de restriction dans votre projet en ajoutant simplement cette annotation à la classe **RestrictionFilter**, et ainsi vérifier que le seul ajout de cette annotation implique bel et bien une activation du filtre, et ce même sans redémarrage de Tomcat ! Pour le désactiver, il vous suffira ensuite de supprimer ou de commenter l'annotation.

WebInitParam

Il existe également une annotation qui ne peut être utilisée qu'au sein d'une annotation `@WebServlet` ou `@WebFilter` : `@WebInitParam`. Comme son nom l'indique, elle est destinée à remplacer la section `<init-param>` qu'il est possible d'inclure aux déclarations d'une servlet ou d'un filtre dans le fichier `web.xml`. Sa documentation nous confirme que seuls deux attributs sont requis : un attribut `name` et un attribut `value`, remplaçant respectivement `<param-name>` et `<param-value>`.

Prenons pour exemple notre servlet **Download**. Pour rappel, sa déclaration dans le fichier `web.xml` était la suivante :

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

```
<servlet>
  <servlet-name>Download</servlet-name>
  <servlet-class>com.sdzee.servlets.Download</servlet-class>
  <init-param>
    <param-name>chemin</param-name>
    <param-value>/fichiers/</param-value>
  </init-param>
</servlet>
<servlet-mapping>
  <servlet-name>Download</servlet-name>
  <url-pattern>/fichiers/*</url-pattern>
</servlet-mapping>
```

L'annotation qui remplace cette description devient :

Code : Java - com.sdzee.servlets.Download

```
...
@WebServlet( urlPatterns = "/fichiers/*", initParams =
@WebInitParam( name = "chemin", value = "/fichiers/" ) )
public class Download extends HttpServlet {
  ...
}
```

Remarquez bien l'utilisation de l'annotation fille `@WebInitParam` au sein de l'annotation mère `@WebServlet`.

Là encore, si vous souhaitez vérifier le bon fonctionnement de l'annotation, il vous suffit de l'ajouter à votre servlet **Download**, de supprimer la déclaration de votre `web.xml`, puis de redémarrer Tomcat et de tenter de télécharger un des fichiers présents sur votre disque en vous rendant sur la page `http://localhost:8080/pro/fichiers/...` (où les ... correspondent au nom ou chemin du fichier) !

WebListener

Reposons-nous un instant avec l'annotation dédiée aux Listener : `@WebListener`. Comme vous pouvez le constater en parcourant sa documentation, elle ne requiert aucun paramètre.

D'ailleurs, vous devez vous souvenir de la courte déclaration de notre Listener **InitialisationDaoFactory** :

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

```
<listener>
  <listener-
  class>com.sdzee.config.InitialisationDaoFactory</listener-class>
</listener>
```

L'annotation qui remplace cette description est tout bonnement :

Code : Java - com.sdzee.config.InitialisationDaoFactory

```

...
@WebListener
public class InitialisationDaoFactory implements
ServletContextListener {
...

```

Comme annoncé, elle n'attend aucun paramètre et suffit à déclarer une classe en tant que Listener au sein d'une application web. Pratique, n'est-ce pas ?

MultipartConfig

Pour terminer, nous allons nous intéresser à l'annotation dédiée au traitement des requêtes de type **Multipart** : `@MultipartConfig`. Sa documentation nous confirme que, tout comme nous l'avions fait dans la section `<multipart-config>` de notre `web.xml`, nous pouvons préciser quatre paramètres : `location`, `fileSizeThreshold`, `maxFileSize` et `maxRequestSize`. Je ne reviens pas sur leur signification, nous les avons déjà découverts dans le chapitre expliquant l'envoi de fichier.

Pour rappel, voici comment nous avions déclaré notre servlet d'upload :

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

```

<servlet>
  <servlet-name>Upload</servlet-name>
  <servlet-class>com.sdzee.servlets.Upload</servlet-class>
  <init-param>
    <param-name>chemin</param-name>
    <param-value>/fichiers/</param-value>
  </init-param>
  <multipart-config>
    <location>c:/fichiers</location>
    <max-file-size>10485760</max-file-size> <!-- 10 Mo -->
    <max-request-size>52428800</max-request-size> <!-- 5 x 10Mo -->
    <file-size-threshold>1048576</file-size-threshold> <!-- 1 Mo -->
  </multipart-config>
</servlet>
<servlet-mapping>
  <servlet-name>Upload</servlet-name>
  <url-pattern>/upload</url-pattern>
</servlet-mapping>

```

C'est un cas d'étude intéressant, puisque la description contient à la fois un paramètre d'initialisation et une section Multipart. L'annotation correspondante est :

Code : Java - com.sdzee.servlets.Upload

```

...
@WebServlet( urlPatterns = "/upload", initParams = @WebInitParam(
  name = "chemin", value = "/fichiers/" ) )
@MultipartConfig( location = "c:/fichiers", maxFileSize = 10 * 1024
  * 1024, maxRequestSize = 5 * 10 * 1024 * 1024, fileSizeThreshold =
  1024 * 1024 )
public class Upload extends HttpServlet {
...

```

Eh oui, nous avons besoin de deux annotations différentes appliquées sur la même classe pour remplacer intégralement la précédente déclaration de la servlet ! En effet, alors que `@WebInitParam` est incluse dans le corps de l'annotation

@WebServlet, l'annotation @MultipartConfig est, quant à elle, indépendante et doit donc être spécifiée à part.

Côté syntaxe, vous pouvez relever deux points intéressants :

- lorsqu'un paramètre attendu est de type numérique, il ne faut pas l'entourer de guillemets comme nous le faisions jusqu'alors pour tous nos paramètres de type String ;
- lorsqu'un paramètre attendu est de type numérique, il est possible d'utiliser des opérateurs mathématiques simples dans sa valeur. Ici, j'ai utilisé l'opérateur de multiplication * afin de rendre visible au premier coup d'œil le fait que notre servlet limite les tailles respectivement à 10 Mo, 5x10 Mo et 1 Mo.

Pour tester, supprimez la déclaration du fichier **web.xml** et ajoutez ces deux annotations à votre servlet **Upload**. Redémarrez alors Tomcat, puis rendez-vous sur <http://localhost:8080/pro/upload> et tentez d'envoyer un fichier trop volumineux, un fichier sans description, et enfin un fichier correspondant aux critères de validation.

Et le web.xml dans tout ça ?

Avec ces quelques annotations simples, vous allez pouvoir mettre au régime votre fichier **web.xml** de manière drastique ! 😊

Toutefois, si vous faites vous-mêmes l'expérience et tentez de remplacer chacune des déclarations par leurs annotations équivalentes, tôt ou tard vous vous rendrez compte que tout ne va pas disparaître. En réalité, le fichier **web.xml** est encore utile pour plusieurs raisons, que nous allons rapidement découvrir.

1. Déclarer la version de l'API servlet

La première utilisation, évidente pour vous je l'espère, est la déclaration de la version de l'API Servlet requise par votre application. Autrement dit, vous avez toujours besoin de la balise <web-app> :

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

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
    version="3.0">

    ...

```

J'en profite pour vous rappeler que les annotations ne sont disponibles qu'à partir de Java EE 6, et qu'il vous faut donc impérativement déclarer l'API Servlet 3.0 pour pouvoir les utiliser dans votre code.

2. Spécifier un ordre de filtrage

Souvenez-vous du chapitre sur les filtres, je vous y avais annoncé que l'ordre de déclaration des filtres dans le fichier **web.xml** était important, car il déterminait l'ordre dans lequel vos filtres allaient être appliqués aux requêtes entrantes.

Eh bien sachez que ce niveau de configuration n'est pas atteignable avec de simples annotations. En effet, en utilisant des annotations il est impossible de préciser dans quel ordre vous souhaitez appliquer différents filtres mappés sur une même requête. Ainsi, si vous souhaitez organiser vos filtres et définir un ordre d'exécution précis, vous devrez continuer à les déclarer dans votre fichier **web.xml**.

Toutefois, si vous souhaitez limiter l'encombrement de votre **web.xml**, vous pourrez toujours utiliser des annotations pour remplacer les sections <filter>, et vous contenter d'y écrire les sections <filter-mapping>. C'est tout ce dont le conteneur a besoin pour définir l'ordre d'exécution des filtres !

Notez cependant que ce découpage, à savoir les annotations d'un côté pour remplacer les sections <filter>, et le fichier **web.xml** de l'autre pour écrire les sections <filter-mapping>, ne fonctionne avec Tomcat que depuis la



version 7.0.28 ! Auparavant, le bug 53354 empêchait le bon fonctionnement. Autrement dit, si vous utilisez une version antérieure de Tomcat et que vous souhaitez donner un ordre à vos filtres, alors vous devrez les déclarer intégralement dans votre `web.xml`, et ne pourrez pas utiliser les annotations `@WebFilter`.

3. Exécuter un filtre externe

Toujours à propos des filtres, il existe un cas particulier dans lequel vous ne pourrez pas utiliser l'annotation `@WebFilter` : lorsque le filtre que vous appliquez ne fait pas partie de votre application ! Sans accès au code source, vous n'avez pas vraiment le choix..

Par exemple, dans notre projet nous utilisons le filtre natif de Tomcat nommé **Set Character Encoding**, afin de compléter la gestion de l'UTF-8. Eh bien pour ce cas précis, nous n'avons pas d'autre choix que de déclarer le filtre à la main dans le `web.xml` de notre application.

4. Surcharger les valeurs définies dans une annotation

Pour conclure, sachez que les valeurs précisées dans toutes vos annotations sont en réalité considérées par le conteneur comme des valeurs par défaut. En d'autres termes, si vous déclarez à la fois une annotation et une section dans le fichier `web.xml`, alors les données de la section seront prises en compte quoi qu'il arrive.

Concrètement, cela signifie que par exemple si vous mappez en même temps une servlet sur une URL depuis une section dans le `web.xml`, et sur une autre URL depuis une annotation, alors la servlet sera en fin de compte mappée sur les deux URL.

Par ailleurs, dans le cas de paramètres à valeur unique comme les `@WebInitParam`, en cas de double définition c'est la valeur contenue dans le `web.xml` qui sera prise en compte, et celle contenue dans l'annotation sera ignorée.

- Une annotation est une métadonnée, une indication liée à l'élément qu'elle cible.
- Les fichiers de configuration externes sont pénibles à utiliser et difficilement maintenables.
- Il existe des annotations propres à Java EE, apparues avec la version 6 de la plate-forme :
 - `@WebServlet` permet de déclarer une servlet ;
 - `@WebFilter` permet de déclarer un filtre ;
 - `@WebInitParam` permet de préciser un paramètre d'initialisation ;
 - `@WebListener` permet de déclarer un listener ;
 - `@MultipartConfig` permet d'activer la gestion des requêtes de type **multipart** depuis une servlet.
- Le fichier `web.xml` devient presque inutile !
- La lisibilité et la compréhension d'une application sont facilitées, les informations de configuration étant maintenant directement présentes au sein du code !

La persistance des données avec JPA

Incontournables pour beaucoup, décriées par certains, les solutions de persistance ont depuis quelques années le vent en poupe, dans les applications Java SE comme dans les applications Java EE. Nous allons dans ce chapitre nous frayer un chemin dans cette jungle de nouveautés, et découvrir comment appliquer correctement une telle solution à notre projet.

Généralités

 Qu'est-ce qu'on appelle la "persistance" de données ?

Si vous ne connaissez pas ce mot, ne vous inquiétez surtout pas : au sens général, il s'agit simplement du terme utilisé pour décrire le fait de stocker des données d'une application de manière... persistante ! Autrement dit, de les sauvegarder afin qu'elles ne disparaissent pas lorsque le programme se termine. Rien de bien compliqué, en somme.

En Java, lorsqu'on parle d'une « solution de persistance des données », on évoque couramment un système permettant **la sauvegarde des données contenues dans des objets**. En réalité, vous connaissez donc déjà tous un moyen de persistance : le stockage dans une base de données relationnelle via JDBC !

 Dans ce cas, pourquoi ne pas nous contenter de notre système actuel ?

En effet, notre tandem JDBC & DAO fonctionne bien, et persiste correctement nos données dans notre base de données MySQL. C'est pourquoi **dans l'absolu**, ce que vous avez appris jusque-là est **suffisant** pour développer une application web Java EE de petite à moyenne envergure. En outre, c'est une solution légère, portable et pratique puisqu'elle ne nécessite qu'un simple conteneur de servlets pour fonctionner !

Toutefois, il faut bien vous rendre compte d'un aspect très important. Dans notre exemple - une très petite application - nous ne nous sommes pas inquiétés de devoir réécrire un DAO pour chaque table manipulée. Pourtant, c'est une belle perte de temps : dans chacune de nos classes, nous utilisons un code source très similaire pour récupérer les données des tables et les transformer en objets. Concrètement, mis à part le SQL, l'affectation des paramètres sur l'objet `PreparedStatement` et la récupération des résultats et leur affectation sur le bean, rien ne change : la structure générale du code reste toujours la même !

Maintenant, imaginez-vous devoir travailler sur un projet impliquant plusieurs centaines de tables... Vous êtes bons pour écrire autant de DAO ! Parce que, même si leur structure globale est identique, chacun d'eux contient des requêtes et attributs spécifiques. Et comme vous le savez tous, non seulement le développeur est fainéant mais, par-dessus tout, **la duplication de code dans un projet, c'est le mal incarné**.

Pour alléger cette tâche, vous pourriez bien évidemment commencer par rajouter de l'abstraction et de la généricité dans votre couche d'accès aux données (souvent raccourcie DAL, pour *Data Access Layer*) et ses interfaces, afin d'obtenir une interface unique pour tous vos DAO, et ne plus devoir écrire que les implémentations spécifiques à chaque table.

Pour alléger davantage la tâche, vous pourriez ensuite réaliser un générateur de DAO, un petit programme qui se chargerait d'écrire le code de base des implémentations pour chacune des tables à votre place, et vous n'auriez alors plus qu'à reprendre chaque méthode incomplète.

Avec un tel système en place, il deviendrait alors envisageable d'attaquer le développement d'une application de grande échelle plus sereinement.

Seulement, il resterait encore des choses très pénibles à réaliser, et surtout très chronophages. Il faudrait encore s'assurer que la correspondance entre les tables et les objets est correctement réalisée, et ce pour chaque action effectuée : lecture de données, mises à jour, insertions, suppressions... À chaque fois il faudrait écrire une requête SQL spécifique, à chaque fois il faudrait récupérer son résultat et éventuellement en extraire les données pour créer ou mettre à jour les attributs d'un bean, à chaque fois...

Vous voyez où je veux en venir ? Vous devez bien sentir que si un système global gérait pour vous ces actions récurrentes, votre travail serait bien plus agréable, n'est-ce pas ? Eh bien voilà pourquoi nous sommes en droit de ne pas nous contenter d'utiliser JDBC & DAO. Et voilà pourquoi je vous présente dans ce chapitre **la solution fournie par Java EE : JPA**.

 Qu'est-ce que JPA ?

Littéralement « Java Persistence API », il s'agit d'un standard faisant partie intégrante de la plate-forme Java EE, une

spécification qui définit un ensemble de règles permettant la gestion de la correspondance entre des objets Java et une base de données, ou autrement formulé la gestion de la persistance.

Ce mécanisme qui gère la correspondance entre des objets d'une application et les tables d'une base de données se nomme ORM, pour « Object-Relational Mapping ». Ainsi dans le sens le plus simpliste du terme, ce que nous avons réalisé dans les chapitres précédents à travers nos DAO n'est rien d'autre qu'un ORM... manuel !



Et Hibernate, TopLink, EclipseLink, OpenJPA... ?

Peut-être avez-vous déjà entendu parler de ces solutions, les bien nommés « *frameworks* ORM ». Comprenez bien que lorsqu'on parle de JPA, il s'agit uniquement d'une API, c'est-à-dire une description d'un comportement à suivre, en l'occurrence pour respecter un standard en place.

C'est le même principe que la fameuse `DataSource` que nous avions découverte dans le dernier chapitre de la partie précédente : l'interface nous expliquait comment faire, mais pour mettre en place concrètement un pool de connexions, il nous a fallu utiliser une implémentation (`BoneCP`). Eh bien là c'est exactement pareil : les **interfaces** de JPA décrivent comment respecter le **standard**, mais nous devons utiliser une **implémentation** pour en tirer parti ! Voilà donc ce que sont [Hibernate](#), [EclipseLink](#) & [consorts](#) : des implémentations du standard JPA. En d'autres termes JPA constitue la théorie, et ces *frameworks* en sont la pratique.

À ce propos, comme toujours, il y a des différences entre la théorie et la pratique. En effet, la plupart de ces solutions sortent du cadre défini par JPA, et proposent des fonctionnalités annexes qui leur sont propres. Ainsi, le développeur doit être conscient de ce qu'il souhaite réaliser : s'il utilise des fonctionnalités spécifiques à un *framework* en particulier, c'est-à-dire une des fonctionnalités qui ne sont pas décrites dans le standard JPA, alors il s'expose au risque de ne plus pouvoir faire machine arrière (choisir une autre solution de persistance) sans devoir modifier le code de son application.

Voilà donc l'avantage de disposer d'un standard bien défini : à partir du moment où l'on fait du JPA, peu importe l'implémentation utilisée, le code applicatif sera identique, et donc portable.



Pour la petite histoire, chronologiquement parlant certains de ces *frameworks* sont apparus avant JPA, c'est le cas notamment de [Hibernate](#) et [TopLink](#). Le standard a en réalité été créé à l'époque dans le but d'harmoniser les solutions existantes, afin de coller à la devise du Java : « *write once, run everywhere* ». Ainsi, vous pouvez voir ce standard comme le dénominateur commun entre toutes les solutions reconnues existantes.

Principe

Maintenant que nous savons de quoi il retourne, découvrons comment tout cela s'organise.

Attention ! Je vous préviens dès maintenant, dans ce chapitre ainsi que dans les chapitres à venir, je ne vais pas détailler mes explications aussi finement que dans les parties précédentes. La principale raison étant que pour travailler sur de tels concepts avancés, il est nécessaire que vous soyez curieux et un minimum autonomes. Des pans entiers de Java EE vont entrer en jeu, et vous allez devoir lire les documentations et liens que je vous fournis au fur et à mesure de votre découverte si vous souhaitez assimiler correctement toutes les informations que je vous transmets. En outre, une autre excellente raison qui me pousse à faire ce choix est que vous devez petit à petit vous libérer du format "tutoriel", et prendre de l'aise avec l'utilisation de ressources en tous genres (documentations, forums, extraits de code, etc.), car plus vous irez loin dans une technologie, moins vous trouverez de personnes pour vous renseigner, et plus il vous faudra compter sur vos capacités à chercher les réponses par vous-mêmes !

Des EJB dans un conteneur

Le concept mère qui se cache derrière JPA, c'est le fameux EJB ou « Enterprise JavaBean ». Voilà encore un terme dont vous avez déjà dû entendre parler à maintes reprises, et pas forcément en bien...

Si je vous dis ça, c'est parce que cette technologie a connu une évolution plutôt chaotique. À leur apparition en 1998, les objets EJB ont suscité l'enthousiasme des développeurs, enthousiasme qui a rapidement laissé place à de vives critiques de tous bords : « *Trop compliqué ! Trop lourd ! Trop de configuration ! Trop de fichiers XML !* », etc. Il aura fallu attendre jusqu'en 2006 pour que la troisième version des EJB gagne finalement ses galons et conquière les développeurs, grâce à un fonctionnement ultra simplifié.

Bien évidemment, nous n'allons pas nous intéresser aux méandres qui ont conduit à ce que sont les EJB aujourd'hui. Tout ce dont nous avons besoin, c'est de comprendre comment ils fonctionnent dans leur version actuelle.

Notez par ailleurs que la toile regorge d'informations dépassées sur ce sujet, et qu'il est parfois difficile de trouver des



informations à jour ailleurs que dans les documentations et cours officiels, bien souvent disponibles en anglais uniquement. Gardez bien en tête que les EJB aujourd'hui, ce sont les EJB 3, et que les EJB 1 et 2 c'est de l'histoire ancienne.

Principe général

Les EJB sont des objets présentant une caractéristique bien particulière : **ils sont gérés par le conteneur**. Attention, quand on parle ici de conteneur il n'est plus question du simple conteneur de servlets que nous avons utilisé jusqu'à présent ! Non, il s'agit bien ici de « **conteneur EJB** », un élément dont le travail est de gérer entièrement le cycle de vie des EJB qu'il contient.



Qu'est-ce que cela signifie concrètement ?

Eh bien tout simplement qu'une grande partie de ce travail pénible, auparavant réalisé par le développeur, est dorénavant déléguée au conteneur, laissant ainsi, au bon fainéant qu'il est, le loisir de se concentrer sur le code métier de son application. Ne vous inquiétez pas si c'est encore flou dans votre esprit, vous comprendrez lorsque nous passerons à la pratique. 😊



Par défaut, tout serveur d'applications Java EE au sens strict du terme contient un conteneur EJB. En revanche, pour ce qui est des serveurs légers comme Tomcat, ce n'est pas le cas ! Ainsi, vous ne pouvez pas manipuler d'EJB depuis Tomcat sans y ajouter un tel conteneur.

JPA, ou les EJB Entity

Il existe deux types d'EJB : les EJB Entity, et les EJB Session. Celui qui sert de pilier à JPA est le premier, et le plus simple à appréhender : l'EJB Entity. C'est lui qui définit quelles données doivent être sauvegardées, et c'est à travers lui qu'est effectuée la correspondance entre un objet et une table d'une base de données.

En apparence, c'est un objet qui ressemble beaucoup à un simple Javabean, dans lequel on ajoute simplement quelques annotations. Souvenez-vous, je vous avais expliqué dans le chapitre précédent que celles-ci n'étaient rien d'autre que des métadonnées... Eh bien en l'occurrence, ces informations sont ici utilisées pour informer le conteneur d'EJB de la manière dont l'objet devra être géré.

Un gestionnaire d'entités

Définir quelles données doivent être sauvegardées en construisant des entités est une première étape. Mais elle ne servirait à rien si n'existaient pas, derrière, un système définissant **comment** ces données doivent être sauvegardées !

Les méthodes permettant d'établir une connexion avec la base de données et de gérer la persistance se trouvent dans un objet particulier nommé **EntityManager**, ou gestionnaire d'entités. Il s'agit, là encore, d'un objet dont le cycle de vie est géré par le conteneur.

Toutefois, qui dit connexion à une base de données, dit configuration manuelle. Voilà pourquoi cet objet se base sur des informations que le développeur doit saisir dans un fichier de configuration externe, que nous allons découvrir prochainement et qui n'est rien d'autre qu'un simple fichier XML.

Pour résumer, le fonctionnement de JPA est basé sur deux briques principales :

- **des EJB « Entity »** : ce sont des objets ressemblant très fortement à des JavaBeans, dans lesquels des annotations définissent des correspondances entre les objets et leurs attributs d'un côté, et les tables relationnelles de la base de données et leurs champs de l'autre (on parle alors de **mapping relationnel/objet**) ;
- **un EntityManager** : c'est une classe qui est chargée de mettre en musique les correspondances définies dans les entités, et qui réalise donc toutes les opérations CRUD (*Create, Read, Update, Delete*) sur la base de données.

Mise en place

Le serveur d'applications GlassFish

Raisons de ce choix

Comme je vous l'ai annoncé un peu plus tôt, Tomcat ne gère pas les EJB de manière native. Et comme vous le savez maintenant, sans EJB pas de JPA. Nous pourrions certes ajouter les jar des différentes solutions dont nous allons avoir besoin pour continuer à travailler sous Tomcat, mais ce n'est pas ce que nous allons faire. Je souhaite en effet profiter de cette occasion pour vous faire travailler sur un autre serveur, un vrai serveur d'applications Java EE cette fois : GlassFish. Pas n'importe quel serveur d'ailleurs : il s'agit du **serveur de référence**, celui qui implémente à la lettre les spécifications Java EE 6. Rien d'étonnant me direz-vous, puisque c'est Oracle - la maison mère de tout l'écosystème Java - qui édite ce serveur !

Si vous vous souvenez bien, je vous en avais déjà brièvement parlé lorsque nous avions découvert la JSTL. Je vous avais alors expliqué que GlassFish, contrairement à Tomcat pour lequel il est nécessaire de fournir un jar, embarquait par défaut la bibliothèque. Eh bien je pourrais vous faire exactement la même réflexion au sujet de JPA. GlassFish embarque par défaut son **implémentation de référence**, le *framework* de persistance nommé [EclipseLink](#), alors qu'il est nécessaire de fournir un jar externe à Tomcat pour qu'il permette de travailler avec JPA.



En outre, sachez également qu'il est possible avec GlassFish de redéployer automatiquement une application après toute modification sur son code ou sa configuration, alors qu'il fallait souvent redémarrer Tomcat pour qu'il prenne en compte des modifications.

Mise en place

Vous l'aurez compris, avec GlassFish, tout est inclus par défaut. Actuellement, la dernière version en date est estampillée « GlassFish 3.1.2.2 ». Pour la récupérer, rendez-vous sur [cette page de téléchargement](#), choisissez ensuite la version intitulée *WebProfile*, et enfin la version correspondant à votre système (dans l'encadré sur la figure suivante, première colonne pour Windows, seconde pour Mac & Linux).

GlassFish Server Open Source Edition 3.1.2.2 Downloads

Confused? Not sure which version to use? Try the simple ZIP archive

Distribution	Windows [1]	Size (MB)	Linux / Unix / Mac / AIX [2] [4]	Size (MB)	Zip archive [3]	Size (MB)
GlassFish Server 3.1.2.2 Open Source Edition Full Platform *	glassfish-3.1.2.2-windows.exe (EN)	53	glassfish-3.1.2.2-unix.sh (EN) glassfish-3.1.2.2-aix.sh	53 55	glassfish-3.1.2.2.zip (EN) glassfish-3.1.2.2-aix.zip	81 90
	glassfish-3.1.2.2-windows-ml.exe (multilingual)	62	glassfish-3.1.2.2-unix-ml.sh (multilingual) glassfish-3.1.2.2-aix-ml.sh	61 65	glassfish-3.1.2.2-ml.zip (multilingual) glassfish-3.1.2.2-aix-ml.zip	93 107
GlassFish Server 3.1.2.2 Open Source Edition Web Profile *	glassfish-3.1.2.2-web-windows.exe (EN)	33	glassfish-3.1.2.2-web-unix.sh (EN)	33	glassfish-3.1.2.2-web.zip (EN)	47
	glassfish-3.1.2.2-web-windows-ml.exe (multilingual)	39	glassfish-3.1.2.2-web-aix.sh (multilingual) glassfish-3.1.2.2-web-aix-ml.sh	35 39 65	glassfish-3.1.2.2-web-aix.zip (multilingual) glassfish-3.1.2.2-web-ml.zip (multilingual) glassfish-3.1.2.2-web-aix-ml.zip	56 59 70

[1]: GUI-based installer for Windows. Can be used in silent mode.

[2]: GUI-based installer for Solaris, Linux, MacOS X, and AIX. Can be used in silent mode.

[3]: Platform-independent download file. Simply unzip and start default domain1.

[4]: The generic -unix archives will not work on AIX, so make sure you get the -aix bits.



Si une version plus récente est disponible lorsque vous lisez ce cours, vous pouvez opter pour cette dernière en lieu et place de la 3.1.2.2, mais ce n'est pas une obligation. Sachez par ailleurs que nous allons très bientôt intégrer notre serveur à Eclipse. Par conséquent, si vous tenez à utiliser une version plus récente, assurez-vous d'abord que son intégration à Eclipse est possible.

Une fois téléchargée, installez-la en suivant simplement les consignes qui vous sont données par l'assistant. Quelques conseils pour la route :

- préférez un répertoire proche de la racine de votre disque. Amis windowsiens, n'allez surtout pas installer le serveur dans les traditionnels "Program Files" ou autres "Documents And Settings"... En ce qui me concerne, je l'ai installé dans un dossier intitulé **glassfish3** directement à la racine de mon disque C:\, et c'est d'ailleurs ce que propose l'installateur par défaut ;
- n'installez pas l'outil de mise à jour automatique ("Update tool"), cela vous évitera un délai supplémentaire lors de l'installation et quelques ennuis éventuels ;
- faites attention à ne pas donner au serveur les mêmes ports que ceux utilisés par Tomcat ; si vous avez gardé par défaut le port 8080 avec Tomcat, utilisez par exemple le port 8088 pour GlassFish, afin d'éviter les conflits en cas d'utilisation simultanée. De même pour le port de la console d'administration, pour lequel vous pouvez en principe laisser la valeur par

défaut 4848, Tomcat utilisant par défaut un port différent (voir la figure suivante) ;

- si vous spécifiez un mot de passe pour la gestion du serveur, notez-le bien quelque part, pour pouvoir le retrouver en cas d'oubli.

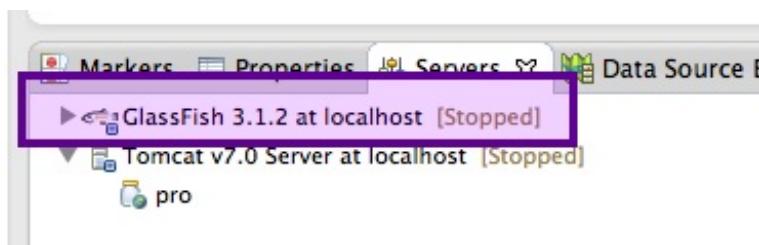
Intégration à Eclipse

L'installation du serveur terminée, ouvrez (ou redémarrez) Eclipse. Dans le volet intitulé **Servers**, en bas de votre espace de travail Eclipse, faites un clic droit dans le vide et choisissez New > Server. Dans la fenêtre qui s'ouvre alors s'affiche une liste des différents serveurs actuellement gérés par votre version d'Eclipse.

Si, dans cette liste, figure un dossier nommé **GlassFish**, alors votre Eclipse est déjà capable de prendre en charge votre serveur fraîchement installé. Si toutefois aucun dossier portant ce nom n'apparaît, vous devrez cliquer sur le lien intitulé « *Download additional servers adapters* ». Eclipse va alors scanner les différents outils disponibles, et une fois la recherche terminée vous devrez choisir l'entrée nommée « **Oracle GlassFish** » et la télécharger. Une fois la récupération terminée, vous pourrez alors poursuivre.

Une fois prêts à intégrer votre serveur à Eclipse, déroulez le dossier nommé **GlassFish** et cliquez sur l'entrée intitulée **GlassFish 3.1.2.2** (ou supérieure si une version plus récente est disponible lorsque vous lisez ce cours). Sur les écrans suivants, ne changez rien aux réglages par défaut et terminez en cliquant sur **Finish**.

Votre serveur apparaît alors dans le volet inférieur de votre espace de travail, aux côtés de votre serveur Tomcat utilisé jusqu'à présent dans le cours (voir la figure suivante).



Configuration

Commencez par vérifier que votre serveur se lance correctement en cliquant sur le bouton de démarrage intégré au volet serveur d'Eclipse. Si des erreurs surviennent, redémarrez Eclipse. Si des erreurs surviennent encore, redémarrez votre poste. Si après cela des erreurs surviennent toujours, alors vous devrez désinstaller proprement GlassFish et reprendre calmement les étapes précédentes.

Une fois le serveur lancé avec succès, vous allez pouvoir procéder à sa configuration. De quelle configuration est-il question exactement ? De celle du pool de connexions que vous allez mettre en place ! Eh oui, maintenant que vous savez mettre en place un pool, vous n'avez plus d'excuse...



Quand nous avons découvert les pools, nous n'avions rien configuré avec Tomcat ! Nous avions simplement ajouté quelques jar et codé quelques lignes dans l'initialisation de notre DAOFactory...

C'est vrai, seulement puisque nous allons travailler avec JPA, c'est dorénavant notre conteneur qui va s'occuper de manipuler la base de données pour nous ! Il va donc bien falloir que nous lui précisions quelque part comment faire. Il existe plusieurs manières de procéder. Celle que je vais vous présenter consiste à paramétrier le pool directement au niveau du serveur, et non plus au niveau de l'application.

Pour commencer, on prend les mêmes et on recommence ! Il faut récupérer les jar de BoneCP et les placer dans le répertoire **/lib** de GlassFish.



Où se trouve ce répertoire exactement ?

Vous remarquerez rapidement que l'organisation interne d'un serveur GlassFish est sensiblement différente de celle d'un serveur Tomcat. Nous n'allons pas nous amuser à parcourir chaque dossier de l'arborescence, je vais simplement vous guider lorsque cela sera nécessaire. En l'occurrence, vous devez placer vos jar dans le dossier **/glassfish3/glassfish/domains/domain1/lib/ext/**.

Une fois ceci fait, il faut ensuite procéder au paramétrage du pool et de la connexion à la base de données. Plutôt que de vous perdre dans les formulaires de configuration de la console d'administration de GlassFish (qui, pour les curieux, est accessible à l'adresse <http://localhost:4848>), je vous ai préparé un fichier XML prêt à l'emploi, contenant toutes les informations dont nous avons besoin : [cliquez ici pour le télécharger](#) (clic droit > Enregistrer sous...). Merci qui ?

Pour information, ce fichier contient la déclaration d'une connexion MySQL via JDBC classique, référencée par le nom « **jdbc/bonecp_resource** », et son association à un pool de connexions basé sur BoneCP.

Une fois le fichier récupéré, vous allez devoir l'appliquer à votre serveur. Pour ce faire, rendez-vous dans le répertoire **/glassfish3/bin**, et copiez-y le fichier. Exécutez alors le fichier nommé **asadmin** (le fichier .bat si vous travaillez sous Windows, l'autre sinon). Une console de commandes s'ouvre alors. Sous Windows, vous allez devoir y taper la commande suivante :

Code : Console

```
add-resources bonecp-datasource.xml
```

Sous Mac ou Linux, vous devrez probablement préciser le chemin complet vers le fichier XML pour que la commande localise correctement votre fichier XML :

Code : Console

```
add-resources /chemin/complet/vers/glassfish3/bin/bonecp-datasource.xml
```

Si l'opération se déroule sans accrocs, vous verrez alors s'afficher un message confirmant le succès de l'opération. Vous pourrez finalement fermer la fenêtre.



Voilà tout ce qu'il est nécessaire de configurer côté serveur. Dorénavant pour tout projet que vous développerez et déploierez sur ce serveur, vous pourrez profiter très simplement d'un pool de connexions BoneCP vers une base de données MySQL prêt à l'usage !

Création du projet

Nous sommes maintenant prêts à créer un nouveau projet sous Eclipse. Pour commencer, rendez-vous dans **New > Dynamic Web Project**, et nommez par exemple votre projet **pro_jpa**. Copiez-y ensuite depuis l'ancien projet **pro** la JSP **inscription.jsp**, le fichier CSS et son répertoire **/inc**, le bean **Utilisateur**, le DAO **UtilisateurDao**, les classes **DAOException** et **FormValidationException**, l'objet métier **InscriptionForm** et enfin la servlet **Inscription**. En clair, tout ce qui va nous être utile pour mettre en place un système d'inscription basé cette fois sur JPA !



Pour le moment, ne vous préoccuez pas des éventuelles erreurs ou avertissements affichés par Eclipse dans votre nouveau projet. Nous allons transformer cet embryon, étape par étape.

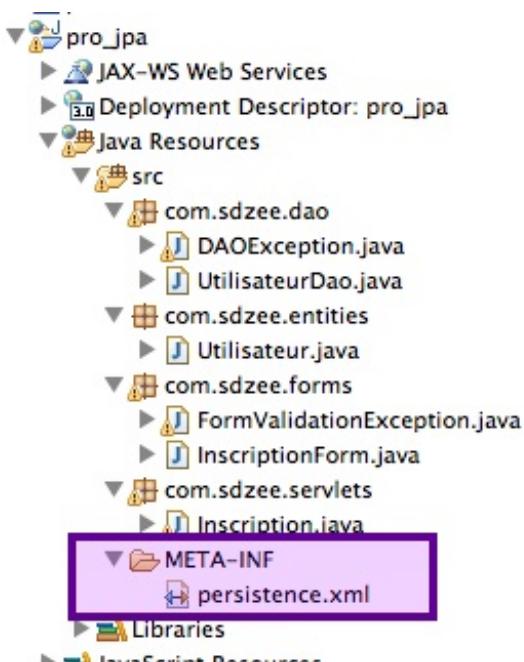
Ensuite, nous allons devoir créer un fichier nommé **glassfish-web.xml** dans le répertoire **/WEB-INF** de notre application. Sans grande surprise, il s'agit d'un fichier ressemblant fortement au fichier **web.xml**, que nous allons ici utiliser pour définir le contexte de déploiement de notre application, ainsi qu'une option qui peut toujours servir. Regardons d'abord le code, et parlons-en ensuite :

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

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE glassfish-web-app PUBLIC "-//GlassFish.org//DTD GlassFish
Application Server 3.1 Servlet 3.0//EN"
"http://glassfish.org/dtds/glassfish-web-app_3_0-1.dtd">
<glassfish-web-app>
    <context-root>/pro_jpa</context-root>
    <class-loader delegate="true"/>
    <jsp-config>
        <property name="keepgenerated" value="true">
            <description>Conserve une copie du code des servlets auto-
générées.</description>
        </property>
    </jsp-config>
</glassfish-web-app>
```

Observez la syntaxe des quelques sections ici mises en place. Concernant la propriété **keepgenerated**, il s'agit d'une option permettant de demander au serveur de garder une copie du code Java des servlets auto-générées depuis vos JSP. Nous n'allons pas nous en servir, mais cela pourra toujours vous être utile dans la suite de votre apprentissage.

Créez ensuite un dossier nommé **META-INF** dans le répertoire **src** du projet, et créez-y un fichier nommé **persistence.xml**. Voici, sur la figure suivante, l'arborescence que vous êtes alors censés obtenir.



Et voici son code de base :

Code : XML - src/META-INF/persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
  xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
  http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">

</persistence>
```

Dans ce fichier nécessaire au bon fonctionnement de JPA, nous allons définir deux choses :

- les informations de connexion à la base de données dont auront besoin les **EntityManager** ;
- une unité de persistance pour chacune de nos entités.



Qu'est-ce que c'est que cette histoire d'unité ?

Une unité de persistance est un contexte, une section qui permet de définir à quelles classes va s'appliquer un **EntityManager**, sur quelle connexion il va se baser et comment il va dialoguer avec la BDD pour persister les données.



Ne venons-nous pas de configurer la connexion directement depuis le serveur ?

Oui, tout à fait. Mais il est tout de même nécessaire de préciser à l'application quelle connexion elle doit utiliser ! Voilà d'ailleurs pourquoi la connexion porte un nom dans notre serveur : c'est pour lui permettre d'être retrouvée très simplement dans un annuaire, que l'on nomme **JNDI**. Nous n'allons pas nous attarder sur ce concept, qui fait partie de Java au sens large.

Nous allons donc mettre en place une unité que nous allons nommer "bdd_sdzee_PU" (PU pour « Persistence Unit »), qui va s'appliquer à notre entité **Utilisateur** et qui va se baser sur la connexion nommée **jdbc/bonecp_resource**. Voici le code nécessaire :

Code : XML

```
<persistence-unit name="bdd_sdzee_PU" transaction-type="JTA">
  <jta-data-source>jdbc/bonecp_resource</jta-data-source>
```

```
<class>com.sdzee.entities.Utilisateur</class>
</persistence-unit>
```

Je vous laisse observer la syntaxe à employer. Concernant le type de transaction, c'est un concept avancé que nous n'allons pas aborder dans ce cours, je vous demande de me faire aveuglément confiance ! ;-)

Pour terminer, nous avons la possibilité de définir des propriétés supplémentaires, concernant la connexion à établir par exemple. Je vous évoquais un peu plus tôt plusieurs manières de configurer une connexion, en voilà une seconde : si nous n'avions pas configuré le pool sur notre serveur, nous aurions dû préciser tout cela directement dans ce fichier. Dans notre cas, nous n'avons aucune propriété à ajouter, et voici le fichier complet :

Code : XML - src/META-INF/persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
  xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
  http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
  <persistence-unit name="bdd_sdzee_PU" transaction-type="JTA">
    <jta-data-source>jdbc/bonecp_resource</jta-data-source>
    <class>com.sdzee.entities.Utilisateur</class>
    <properties/>
  </persistence-unit>
</persistence>
```

Création d'une entité Utilisateur

Nous pouvons maintenant attaquer la transformation de notre projet. Notre première cible est le bean **Utilisateur**, dans lequel nous allons inclure quelques annotations. Attention, il n'est pas ici question des annotations Java EE que je vous ai présentées dans le chapitre précédent, mais bien d'[annotations propres à JPA](#). Gardez ce lien vers la Javadoc officielle sous le coude, car vous allez en avoir besoin pour comprendre en détail chacune des annotations que nous allons mettre en jeu.

Pour commencer, nous allons indiquer à notre serveur, plus précisément à notre conteneur, que notre bean **Utilisateur** va devenir un EJB Entity. Pour cela, nous devons l'annoter avec `@Entity` :

Code : Java - Extrait de l'entité com.sdzee.tp.beans.Utilisateur

```
@Entity
public class Utilisateur {
  ...
}
```

En théorie, il faudrait également indiquer via l'annotation `@Table(name = "Utilisateur")` que l'entité est liée à la table nommée **Utilisateur** dans notre base. Toutefois, le comportement par défaut du conteneur, en l'absence de cette précision, est de considérer que le nom de la table est identique à celui de la classe. Dans notre cas, notre bean et notre table s'appellent tous deux **Utilisateur**, et nous pouvons donc nous passer de cette annotation superflue.

Il est ensuite nécessaire de définir la correspondance entre les données de l'entité **Utilisateur** et les données qui se trouvent dans la table. Nous allons donc devoir annoter les attributs de notre entité. Là encore, par défaut le conteneur sait identifier lui-même que tel attribut correspond à tel champ, à partir du moment où les champs portent le même nom que les attributs.

Lorsque ce n'est pas le cas par contre, il est nécessaire d'ajouter une annotation sur l'attribut pour préciser avec quel champ doit être établie une correspondance. Dans notre cas, nous avons deux attributs qui portent un nom différent de leur équivalent dans la table, et nous devons donc ajouter les annotations `@Column(name = "...")` sur chacun d'eux :

Code : Java - Extrait de l'entité com.sdzee.tp.beans.Utilisateur

```

@Column( name = "mot_de_passe" )
private String motDePasse;
...
@Column( name = "date_inscription" )
private Timestamp dateInscription;

```

Enfin, pour préciser qu'un attribut est associé à la clé primaire de la table, il est nécessaire de l'annoter avec `@Id`. En outre, pour que le conteneur puisse correctement gérer l'id auto-généré lors de la création d'un utilisateur en base, il faut ajouter l'annotation `@GeneratedValue` sur cet attribut (voyez [sa documentation](#) pour plus de détails) :

Code : Java - Extrait de l'entité com.sdzee.tp.beans.Utilisateur

```

@Id
@GeneratedValue( strategy = GenerationType.IDENTITY )
private Long id;

```

Et... c'est tout ! Eh oui, voilà tout ce qu'il est nécessaire de modifier pour transformer notre simple JavaBean en un EJB Entity, prêt à être géré par notre conteneur ! Par ailleurs, puisque nous venons de transformer notre bean en entité, nous allons en profiter pour renommer le package qui la contient par `com.sdzee.tp.entities`. Voici le code final de notre nouvelle entité, les getters/setters exclus :

Code : Java - com.sdzee.tp.entities.Utilisateur

```

package com.sdzee.entities;

import java.sql.Timestamp;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Utilisateur {

    @Id
    @GeneratedValue( strategy = GenerationType.IDENTITY )
    private Long id;
    private String email;
    @Column( name = "mot_de_passe" )
    private String motDePasse;
    private String nom;
    @Column( name = "date_inscription" )
    private Timestamp dateInscription;

    ... // couples de getters/setters pour chaque attribut déclaré
}

```

Remarquez la simplicité avec laquelle nous avons opéré les changements : cinq petites annotations et le tour est joué ! Si nos attributs portaient tous le même nom que leur équivalent dans la table, seules trois annotations auraient suffi !

Création d'un EJB Session

Le moment est pour nous venu de découvrir le second type d'EJB : l'EJB Session. Vous ne savez pas encore de quoi il retourne, mais ne vous inquiétez pas, il n'y a rien de sorcier. Il s'agit simplement d'un objet qui donne accès aux services & méthodes qu'il contient. Il existe deux types d'EJB Session : ceux qui sont **Stateless**, et ceux qui sont **Stateful**. Pour vous aider à comprendre la différence entre ces deux types d'EJB Session, prenons deux exemples :

Stateless : authentification sur un site marchand

Premier exemple, un système d'authentification (connexion) à un site marchand. Dans sa version la plus simple, pour réaliser cette tâche il suffit d'un objet contenant une méthode qui compare un couple d'identifiants passés en paramètres à ceux qui sont stockés dans la table des clients en base, et qui retourne un code de succès ou d'erreur en retour. Cet objet ne présente donc pas de risques liés aux multiples *Threads* (en anglais, on parle de *thread-safety*), et une seule instance peut très bien être partagée par de multiples requêtes issues de clients différents.

De manière concise, voici les propriétés d'un EJB Stateless :

- aucune donnée n'est retenue ni enregistrée, c'est-à-dire qu'aucun état n'est retenu. On dit alors que l'objet est sans état, ou *Stateless* ;
- aucun mécanisme ne garantit que deux appels consécutifs à une méthode d'un tel EJB visent une seule et même instance ** ;
- les accès concurrents sont impossibles, mais le système est *threadsafe* tout de même puisque le conteneur envoie les requêtes simultanées vers des instances différentes du même EJB ** .

(** : nous allons y revenir lorsque nous modifierons notre servlet)

Stateful : panier sur un site marchand

Second exemple, un système de panier sur un site marchand. Pour réaliser une telle tâche, nous avons besoin d'un objet qui soit capable de retenir les commandes effectuées par un client, et qui puisse être réutilisé par ce même client pendant sa session d'utilisation, sans risque qu'un autre client puisse y accéder. Cet objet présente donc un risque lié aux multiples *Threads* : une même instance ne doit surtout pas être partagée par plusieurs requêtes issues de clients différents, sans quoi le panier perd tout son intérêt.

De manière concise, voici les propriétés d'un EJB Stateful :

- des données sont retenues dans l'objet après un appel, c'est-à-dire qu'il conserve un état. On dit alors que l'objet est à état, ou *Stateful* ;
- l'accès à une instance de l'EJB est réservé à un seul client à la fois ;
- les accès concurrents sont impossibles, le conteneur gère une liste d'attente en cas de tentatives simultanées.

Maintenant que nous connaissons le principe, réfléchissons un peu. À quoi va bien pouvoir nous servir un EJB Session dans notre application ? Je vous donne un indice : nous allons utiliser un EJB Session de type Stateless. Regardez rapidement le code de nos différents objets, et essayez de repérer lequel parmi eux correspond parfaitement à la définition d'un objet sans état.

Vous avez trouvé ? Eh bien oui, nous allons tout bonnement remplacer notre précédent DAO ! Après tout, il s'agit bien là d'une classe ne contenant que des méthodes d'interaction avec JDBC. Bien entendu cette fois, nous n'allons plus converser avec JDBC. Car cela, c'est le conteneur qui va s'en occuper pour nous grâce à JPA ! Dorénavant, nous allons simplement demander à notre `EntityManager` de donner des ordres à notre base de données, via ses méthodes `persist()`, `find()`, `remove()`, etc. N'hésitez pas à parcourir en détail la Javadoc de cet objet afin de découvrir tous les trésors qu'il recèle.

Voici donc ce que va devenir notre classe `UtilisateurDao`. Je vous donne d'abord le code, les explications viennent après :

Code : Java - com.sdzee.dao.UtilisateurDao

```
package com.sdzee.dao;

import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.NoResultException;
import javax.persistence.PersistenceContext;
import javax.persistence.Query;

import com.sdzee.entities.Utilisateur;

@Stateless
public class UtilisateurDao {
    private static final String JPQL_SELECT_PAR_EMAIL = "SELECT u
FROM Utilisateur u WHERE u.email=:email";
    private static final String PARAM_EMAIL = "email";
```

```

    // Injection du manager, qui s'occupe de la connexion avec la
BDD
    @PersistenceContext( unitName = "bdd_sdzee_PU" )
    private EntityManager           em;

    // Enregistrement d'un nouvel utilisateur
    public void creer( Utilisateur utilisateur ) throws DAOException
    {
        try {
            em.persist( utilisateur );
        } catch ( Exception e ) {
            throw new DAOException( e );
        }
    }

    // Recherche d'un utilisateur à partir de son adresse email
    public Utilisateur trouver( String email ) throws DAOException {
        Utilisateur utilisateur = null;
        Query requete = em.createQuery( JPQL_SELECT_PAR_EMAIL );
        requete.setParameter( PARAM_EMAIL, email );
        try {
            utilisateur = (Utilisateur) requete.getSingleResult();
        } catch ( NoResultException e ) {
            return null;
        } catch ( Exception e ) {
            throw new DAOException( e );
        }
        return utilisateur;
    }
}

```



Quarante lignes, imports et sauts de ligne compris. On est bien loin des 100+ lignes de notre ancien **UtilisateurDaoImpl** !

Première remarque, la structure globale de l'objet n'a pas changé : il est toujours constitué de deux méthodes, chargées respectivement de créer et de trouver un utilisateur en base. Afin de ne pas avoir à modifier les appels à ces deux méthodes depuis notre objet métier, j'ai pris soin de conserver leur nom et paramètres à l'identique : `creer(Utilisateur utilisateur)` et `trouver(String email)`.

Deuxièmement, remarquez avec quelle simplicité nous précisons à notre conteneur que l'objet est un EJB de type Stateless. Il suffit pour cela d'une annotation `@Stateless` placée avant la déclaration de la classe.

De même, remarquez avec quelle simplicité nous pouvons injecter dans notre EJB une instance d'un **EntityManager** dépendant d'une unité de persistance, via l'annotation `@PersistenceContext(unitName = "...")`. Vous reconnaîtrez ici le nom de l'unité que nous avons déclarée dans le fichier **persistence.xml** : `bdd_sdzee_PU`.



« Injecter » ?

C'est le terme utilisé pour décrire le fait que le cycle de vie de l'objet annoté est géré par le conteneur. En d'autres termes, cela signifie que nous n'avons plus besoin de nous occuper de la création ni de l'initialisation de l'objet, c'est le conteneur qui va le faire pour nous ! En effet comme vous pouvez le voir dans ce code, nous faisons appel à des méthodes de l'objet `em` mais à aucun moment nous ne créons ni n'initialisons une instance d'objet, nous nous contentons uniquement de sa déclaration et de son annotation !

Analysons ensuite la méthode de création, la plus simple des deux. Alors qu'elle occupait auparavant une vingtaine de lignes et faisait appel à des méthodes utilitaires pour la création d'une requête préparée et la libération des ressources utilisées, elle est désormais réduite à peau de chagrin grâce à une seule et unique méthode : `persist()`. C'est cette méthode de l'**EntityManager** qui va se charger de tout pour nous, nous avons uniquement besoin de lui transmettre notre entité **Utilisateur** et tout le reste s'effectue derrière les rideaux ! Et si nous n'avions pas mis en place une exception spécifique de type **DAOException**, notre méthode tiendrait sur une seule et unique ligne ! 😊

En ce qui concerne la méthode de récupération d'un utilisateur en se basant sur son adresse email, ce n'est pas aussi magique. En effet, comment notre **EntityManager** pourrait-il deviner que nous souhaitons utiliser l'argument passé en paramètre dans une

clause **WHERE** appliquée à une requête sur la table **Utilisateur** ? La seule requête de lecture qu'il est possible de faire par défaut, c'est la plus basique, à savoir la recherche d'un élément en se basant sur sa clé primaire. Ce cas mis à part, l'**EntityManager** ne peut pas deviner ce que vous souhaitez récupérer : la clause **WHERE** permet à elle seule d'effectuer de nombreuses requêtes différentes sur une seule et même table.

Nous n'y coupons pas, nous devons écrire un minimum de SQL pour parvenir à nos fins. Toutefois, il n'est plus nécessaire d'écrire du SQL directement via JDBC comme nous le faisions dans nos DAO. Dorénavant, JPA nous facilite la tâche en encadrant notre travail, et nous propose pour cela deux méthodes différentes : le langage **JPQL**, et le système **Criteria**. Le plus simple à prendre en mains est le JPQL, car il ressemble très fortement au langage SQL. Cependant, il présente une différence majeure : il n'est pas utilisé pour interagir avec la base de données, mais avec les entités de notre application. Et c'est bien là l'objectif de JPA : découpler l'application du système de stockage final.

Pour créer une telle requête, il suffit d'appeler la méthode `createQuery()` de l'**EntityManager**. Analysons brièvement la syntaxe de notre requête :

Code : JPQL

```
SELECT u FROM Utilisateur u WHERE u.email=:email
```

Tout d'abord, observez la ressemblance frappante avec le langage SQL. Voici les subtilités à noter :

1. dans la section **FROM**, nous définissons le type de l'objet ciblé par la requête, **Utilisateur**, et son alias dans la requête courante, ici **u** ;
2. dans la section **SELECT**, nous précisons simplement que nous souhaitons récupérer l'entité **Utilisateur** en notant son alias **u** ;
3. dans la clause **WHERE**, nous ciblons l'attribut **email** de l'entité **Utilisateur** en concaténant simplement l'alias et le nom du champ, ici **u.email**. Enfin, nous spécifions que nous allons fournir un paramètre à la requête et le nommons **email** via la notation **:email**.

Ainsi, après la construction de cette requête à la ligne 32, nous fournissons un paramètre nommé **email** et contenant l'adresse utilisée pour effectuer la recherche parmi les entités, grâce à la très explicite méthode `setParameter()`. Ceci devrait vous rappeler le principe des requêtes préparées que nous avions auparavant mises en place dans nos DAO.

Enfin, nous réalisons l'appel qui va déclencher la requête JPQL via la méthode `getSingleResult()`. Comme vous pouvez le lire dans sa documentation, celle-ci retourne un unique résultat. Nous aurions également pu utiliser la méthode `getResultSet()`, mais ce n'est pas nécessaire : nous savons qu'une adresse email est forcément unique dans notre base de données, et nous pouvons donc être certains qu'il n'existera jamais plus d'un résultat.

Par contre, puisque nous utilisons `getSingleResult()` nous devons faire attention au cas où aucun résultat ne serait retourné, c'est-à-dire lorsqu'une adresse n'existe pas encore en base. En effet, la documentation de la méthode nous prévient qu'elle envoie une exception de type **NoResultException** lorsque rien n'est trouvé. Voilà pourquoi nous entourons l'appel d'un bloc **try / catch**, dans lequel nous retournons **null** en cas d'absence de résultat.

Les autres exceptions possibles étant peu probables ou liées à des erreurs de configuration, nous nous contentons pour finir de les encapsuler dans notre exception spécifique **DAOException**.

 Nous en avons terminé avec notre EJB. Ne vous laissez pas méprendre par la longueur de mes explications : j'ai pris le temps de détailler chaque point pour que vous compreniez aisément comment s'utilisent les nouveaux éléments que vous avez découverts, mais si vous regardez maintenant à nouveau le code, vous vous rendrez compte qu'il est très simple et surtout très court !

Modification de la servlet

Auparavant, nous récupérions une instance de DAO lors de la création de la servlet, depuis une Factory qui était quant à elle initialisée au lancement de l'application. Avec JPA, nous allons pouvoir nous débarrasser de tout cet arsenal, et par conséquent nous n'allons plus avoir besoin de la méthode `init()` dans notre servlet !

 Dans ce cas, comment allons-nous transmettre une unique instance d'objet qui sera partagée par toutes les requêtes entrantes, comme l'était notre DAO auparavant ?

Eh bien en réalité, nous n'allons plus nous contenter d'une seule instance, mais de plusieurs instances que le conteneur va gérer pour nous. Il va littéralement créer un pool d'EJB, similaire sur le principe à notre pool de connexions. Lorsque plusieurs requêtes quasi simultanées émanant de clients différents seront redirigées vers notre servlet, le conteneur va transmettre une instance différente de notre EJB à chacune d'elles tant qu'il en restera dans son pool. Quand le pool sera vide, il générera une file d'attente et attendra le retour des instances déjà distribuées pour servir les requêtes en attente.

Sur le papier, tout cela paraît bien compliqué, n'est-ce pas ? Eh bien en pratique, figurez-vous qu'il n'y a rien de plus simple ! D'après vous, qu'est-ce qui permet de donner des informations au conteneur, et qui nécessite insolemment peu d'efforts ?... Bingo ! C'est à l'aide d'une simple mais puissante annotation que nous allons **injecter** notre EJB directement dans notre servlet : `@EJB`.



« Injecter » ?

Eh oui, là encore nous allons faire appel à un mécanisme d'injection similaire à celui que nous avons découvert un peu plus tôt dans notre EJB Stateless. Pour rappel, cela signifie que nous n'avons plus besoin de nous occuper de la création ni de l'initialisation de l'objet, c'est le conteneur qui va le faire pour nous ! Volez plutôt :

Code : Java

```
package com.sdzee.servlets;

import java.io.IOException;

import javax.ejb.EJB;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import com.sdzee.dao.UtilisateurDao;
import com.sdzee.entities.Utilisateur;
import com.sdzee.forms.InscriptionForm;

@WebServlet( urlPatterns = { "/inscription" } )
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";

    // Injection de notre EJB (Session Bean Stateless)
    @EJB
    private UtilisateurDao utilisateurDao;

    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( utilisateurDao
);

    /* Traitement 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 );
}
}
```

Comme vous pouvez le voir à la ligne 33, nous transmettons notre EJB - annoté à la ligne 23 - à notre objet métier, de la même manière que nous lui passions l'instance de DAO auparavant. Mais alors que nous devions initialiser manuellement le contenu de notre objet **utilisateurDao** dans notre version précédente, cette fois nous nous reposons entièrement sur le conteneur : à aucun moment nous ne créons ni n'initialisons une instance d'objet, nous nous contentons uniquement de sa déclaration et de son annotation !



Comprenez et retenez bien cet aspect fondamental de l'EJB : nous ne nous occupons plus de son cycle de vie, nous déléguons entièrement ce travail au conteneur.

Avant de passer à la suite, il y a quelque chose de très important que vous devez comprendre au sujet de l'injection d'EJB dans une servlet : **elle est exclusivement réservée aux EJB de type Stateless**. Souvenez-vous : il n'existe qu'une seule et unique instance de votre servlet, que le conteneur initialise au chargement de votre application. Autrement dit, les variables d'instances sont partagées entre toutes les requêtes qui sont redirigées vers votre servlet !

Ainsi, déclarer et utiliser un objet localement dans une des méthodes **doXXX()** ne pose aucun problème, car chaque requête entrante - c'est-à-dire chaque *Thread* - va créer localement sa propre instance de l'objet. Par contre, déclarer un objet en dehors des méthodes **doXXX()** est à proscrire si l'objet en question conserve un état (en d'autres termes, s'il est Stateful), car il pourra alors être partagé par plusieurs requêtes issues de clients différents. Si vous ne comprenez pas le problème, souvenez-vous de l'exemple du panier d'achats sur un site marchand...

Bref, voilà pourquoi il ne faut jamais procéder à l'injection d'objets Stateful dans une servlet : cela causerait très probablement des comportements non souhaités, que vous ne pourrez déceler qu'après des tests poussés.



Dans ce cas, pourquoi nous ne nous sommes pas posé cette question lorsque nous partagions notre DAO entre toutes nos requêtes auparavant ?

Remarque pertinente : notre DAO était effectivement déclaré en tant que variable d'instance dans notre servlet. Eh bien l'explication, c'est que notre DAO était... sans état ! Eh oui, il ne conservait aucune donnée et pouvait très bien être rendu accessible à différents *threads*. Bien entendu, je m'étais bien gardé de vous parler de tout ça si tôt, vous aviez déjà bien assez de grain à moudre avec le reste. Disons que je vous avais conduits à me faire inconsciemment et aveuglément confiance ! 😊



Dans un chapitre annexe à la fin du cours, vous en apprendrez davantage sur le cycle de vie d'une servlet et sur son caractère *multi-threads*, via des exemples simples et un déroulement pas à pas du cheminement des requêtes traitées.

Enfin, mais vous l'avez probablement déjà remarqué dans le code, vous n'oublierez pas d'insérer l'annotation Java EE **@WebServlet** que nous avons découverte dans le chapitre précédent, afin de déclarer votre servlet et ainsi ne plus avoir à maintenir un pénible fichier web.xml !

Modification de l'objet métier

Nous apercevons le bout du tunnel ! Dans notre objet métier, il nous reste une légère modification à apporter, afin de prendre en charge l'initialisation de la date d'inscription au sein du code. Eh oui, réfléchissez bien : puisque c'est maintenant un **EntityManager** qui va s'occuper pour nous de l'insertion en base, et que cette **EntityManager** fait tout derrière les rideaux de manière automatisée via la méthode **persist()**, nous n'avons plus la possibilité de préciser manuellement à MySQL d'initialiser le champ **date_inscription** via la fonction NOW() comme nous le faisions auparavant. Nous pourrions mettre en place une valeur par défaut pour le champ, en allant directement modifier la table SQL, mais nous sommes là pour faire du Java et un simple ajout suffit pour pallier ce manque :

Code : Java - Extraits de com.sdzee.forms.InscriptionForm

```

public Utilisateur inscrireUtilisateur( HttpServletRequest request )
{
    String email = getValeurChamp( request, CHAMP_EMAIL );
    String motDePasse = getValeurChamp( request, CHAMP_PASS );
    String confirmation = getValeurChamp( request, CHAMP_CONF );
    String nom = getValeurChamp( request, CHAMP_NOM );
    Timestamp date = new Timestamp( System.currentTimeMillis() );

    Utilisateur utilisateur = new Utilisateur();
    try {
        traiterEmail( email, utilisateur );
        traiterMotsDePasse( motDePasse, confirmation, utilisateur );
        traiterNom( nom, utilisateur );
        traiterDate( date, utilisateur );
        ...
    }
    ...
}

/*
 * Simple initialisation de la propriété dateInscription du bean
 * avec la
 * * date courante.
 */
private void traiterDate( Timestamp date, Utilisateur utilisateur )
{
    utilisateur.setDateInscription( date );
}

```

À la ligne 6, nous initialisons un objet **Timestamp** avec la date et l'heure courantes. Ligne 13, nous réalisons un appel à une méthode **traiterDate()**, que nous définissons ensuite aux lignes 23 à 25 et qui se contente d'appeler le *setter* de notre entité. Le reste du code est inchangé, je me suis donc permis de tronquer les blocs pour rester concis. Vous prendrez bien soin d'intégrer correctement ces légers ajouts au code existant de votre objet métier !

 Nous y voilà ! En prenant un peu de recul, vous vous apercevrez que le code gérant l'accès aux données est autrement plus léger qu'il ne l'était lorsque nous faisions tout à la main, et que le reste de l'application profite par la même occasion de ce régime. Paradoxalement, vous vous rendrez compte que cette nouvelle architecture est très proche de l'ancienne ! Eh oui, le code que je vous avais fait mettre en place était déjà plutôt bien organisé, n'est-ce pas ? ;-)

Tests et vérifications

Vérification du bon fonctionnement d'une inscription

Depuis Eclipse, déployez votre projet sur l'instance de GlassFish que vous y avez intégré précédemment, et démarrez le serveur. Rendez-vous ensuite sur la page http://localhost:8088/pro_jpa/inscription depuis votre navigateur, puis faites de simples tests : avec erreurs dans certains champs, puis sans erreur, et enfin avec une adresse déjà enregistrée en base. Si tout est correctement en place, absolument rien ne doit changer dans le comportement de votre application par rapport à notre dernier projet ! 😊

Si vous obtenez une erreur quelconque, cela signifie que vous avez oublié quelque chose en cours de route. Assurez-vous que :

- votre serveur MySQL est bien démarré ;
- votre serveur GlassFish est bien configuré (le pool, notamment) et démarré ;
- vous avez bien copié et modifié toutes les classes et tous les fichiers nécessaires ;
- votre projet est bien déployé sur votre serveur.

Dans les coulisses par contre, vous savez que beaucoup de choses ont changé, et que vous avez presque complètement abandonné MySQL et JDBC. D'ailleurs à ce sujet, si nous allions jeter un œil aux requêtes effectuées par EclipseLink - notre solution JPA - lors d'un appel aux méthodes présentes dans notre DAO, c'est-à-dire notre EJB Stateless ?

Analyse des requêtes SQL générées lors d'une inscription

Vous pouvez analyser les requêtes SQL effectuées par un **EntityManager** en activant l'écriture de logs depuis son unité de persistance dans le fichier **persistence.xml**. Vous vous souvenez de ces propriétés dont je vous ai parlé ? Eh bien le moment est venu de nous en servir :

Code : XML - src/META-INF/persistence.xml

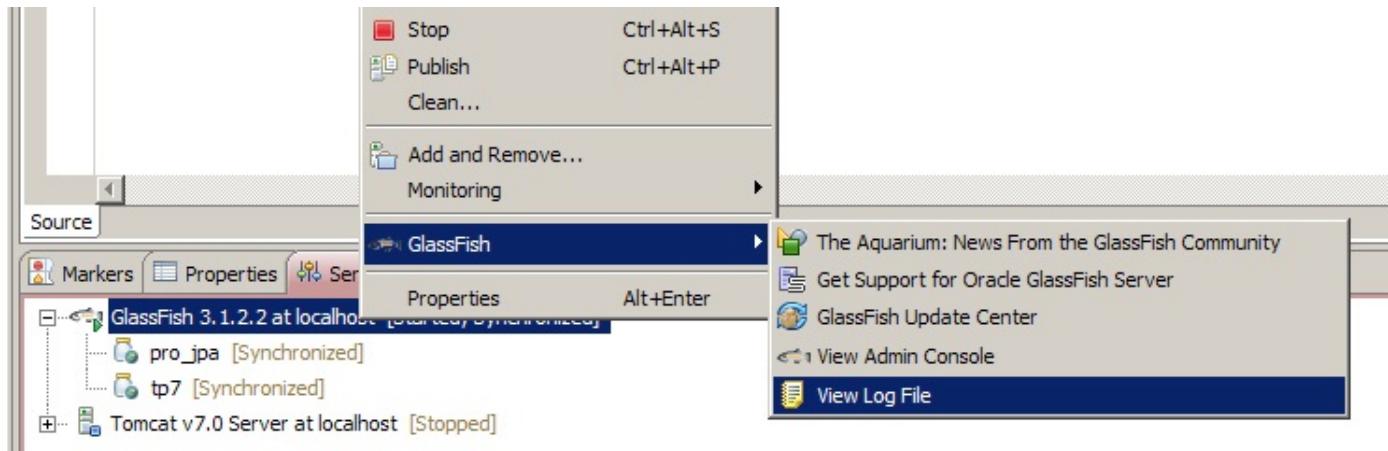
```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
  xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
  <persistence-unit name="bdd_sdzee_PU" transaction-type="JTA">
    <jta-data-source>jdbc/bonecp_resource</jta-data-source>
    <class>com.sdzee.entities.Utilisateur</class>
    <properties>
      <property name="eclipselink.logging.level.sql"
        value="FINE"/>
      <property name="eclipselink.logging.parameters"
        value="true"/>
    </properties>
  </persistence-unit>
</persistence>
```

En ajoutant cette section aux lignes 9 à 12 dans notre fichier **persistence.xml**, nous demandons explicitement à notre solution JPA - EclipseLink - d'écrire dans le fichier de logs du serveur les requêtes SQL qu'elle envoie à la base de données.



Où se trouve ce fichier de logs ?

Selon votre configuration, vous verrez peut-être s'afficher les requêtes effectuées dans l'onglet intitulé **Console** en bas de votre espace de travail Eclipse, aux côtés de l'onglet **Servers** dans lequel vous gérez le déploiement de votre application sur GlassFish. Vous avez en principe la possibilité de visualiser le contenu des logs du serveur en effectuant un clic droit sur son nom, et en suivant GlassFish > View Log file, comme indiqué sur la figure suivante.



Si cela ne fonctionne pas sur votre poste, alors il vous suffit d'aller regarder manuellement le fichier nommé **server.log** situé dans le répertoire **/glassfish3/glassfish/domains/domain1/logs**, avec votre éditeur de texte préféré. Si par exemple vous essayez de vous connecter avec une adresse email déjà enregistrée, vous devriez visualiser une requête de cette forme, qui correspond bien à la requête que nous avons écrite en JPQL dans notre EJB :

Code : Console

```
[#|2012-12-03T23:27:09.358+0800|FINE|glassfish3.1.2|org.eclipse.persistence.session
.../glassfish3/glassfish/domains/domain1/eclipseApps/pro_jpa/WEB-
INF/classes/_bdd_sdzee_PU.sql|_ThreadID=18;_ThreadName=Thread-3;
ClassName=null;MethodName=null;|SELECT ID, date inscription, EMAIL, mot de passe, N
```

```
)
    bind => [coyote@gmail.com] |#]
```

Dans cet exemple, j'ai essayé de m'inscrire avec l'adresse coyote@gmail.com alors qu'elle existe déjà en base. Ne vous préoccupez pas des informations affichées en amont : ce sont simplement d'autres marqueurs qui ne nous intéressent pas pour notre étude. Si maintenant vous vous inscrivez avec succès, vous allez pouvoir observer les requêtes suivantes :

Code : Console

```
[#| .... | INSERT INTO UTILISATEUR (date_inscription, EMAIL, mot_de_passe, NOM) VALU
    bind => [2012-12-
03 23:37:31.582, coyote@test.com, wZdEPOFZ3KpA7qCanXtfeHtb0GDe66qEmh0FvLKnp00dP0PGK
[#| .... | SELECT LAST_INSERT_ID() |#]
```

Dans cet exemple, j'ai utilisé pour m'inscrire l'adresse coyote@test.com, et j'ai renseigné mon nom avec le mot "test". Il est intéressant de remarquer deux choses :

- la méthode `persist()` de l'EntityManager - que, pour rappel, nous appelons à la ligne 20 de notre EJB Stateless - s'occupe pour nous de générer automatiquement une requête d'insertion et d'y assigner les différentes valeurs des paramètres attendus. Nul besoin de le préciser, la requête générée est bien entendu une requête préparée ;
- afin de récupérer l'id auto-généré, vous pouvez voir ici la seconde requête SQL effectuée : `SELECT LAST_INSERT_ID()`.

Je vous laisse analyser davantage par vous-mêmes les échanges ayant lieu dans les différents cas d'usage de notre petite application.

Aller plus loin

Vous devez être conscients d'une chose : je ne peux ici que guider vos premiers pas. Les *frameworks* ORM, même limités strictement au standard JPA, couvrent un panel d'applications si vaste qu'il est presque impossible d'être exhaustif sur le sujet. Par ailleurs, bien que vous soyez arrivés jusqu'ici, il va encore vous falloir acquérir de l'expérience avant de pouvoir saisir pleinement toutes les possibilités offertes par ces solutions.

 En aparté, voilà pourquoi il est extrêmement important à mes yeux que vous assimiliez les concepts dans l'ordre, en découvrant et appliquant les bonnes pratiques au fur et à mesure de votre apprentissage. Voilà donc pourquoi je vous ai fait apprendre étape par étape depuis le début de ce cours : pour que vous ayez une expérience concrète de ce qui se passe en coulisses. Et voilà pourquoi même si je n'ai fait ici que vous mettre dans la bonne direction, je vous ai donné les clés nécessaires à la bonne compréhension des principes mis en jeu.

Vous aurez très vite l'occasion d'en découvrir davantage avec la prochaine et dernière étape du fil rouge, mais il reste un nombre conséquent d'informations importantes et d'outils très pratiques que nous n'allons pas aborder ensemble. En voici quelques-uns, pour vous mettre l'eau à la bouche :

- avec un *framework* ORM, il est possible de générer automatiquement le code des entités depuis les tables d'une base de données ! Encore moins de code à écrire, et donc moins de travail pour le développeur ;
- avec un *framework* ORM, il est possible de générer automatiquement les tables d'une base de données depuis les entités d'une application ! Eh oui, la réciproque est également possible : si un développeur préfère réaliser un modèle objet plutôt qu'un modèle de données, il lui est possible de ne pas avoir à s'occuper lui-même de la création des tables !
- avec un *framework* ORM, la gestion des transactions est automatisée. C'est un aspect que nous n'avons pas abordé dans ce cours, mais qui entre très vite en jeu dans une application un petit peu plus évoluée ;
- avec un *framework* ORM, ...

ORM, ou ne pas ORM ?

Beaucoup d'interrogations et de débats fusent sur la toile concernant le pour et le contre. Beaucoup de clichés, d'informations dépassées voire erronées, et surtout un manque d'information flagrant. Tentons de mettre au clair quand utiliser un ORM, et quand s'en passer...

Génération des requêtes

Si les requêtes créées par une solution de persistance sont parfaites pour les actions de type CUD (*Create, Update, Delete*), elles sont plus discutables pour les requêtes de type R (les SELECT et toutes leurs clauses et jointures éventuelles).

Perte de contrôle sensible sur le SQL utilisé pour communiquer avec la BDD

C'est un fait indiscutable : de par sa nature, un *framework* ORM masque les communications avec la BDD. S'il est possible de paramétriser finement une solution de persistance afin qu'elle produise des requêtes SQL au plus proche de ce que l'on aurait fait à la main, le jeu n'en vaut pas toujours la chandelle. Il faut parfois passer plus de temps à paramétriser le tout qu'à le faire soi-même. Certes, cette abstraction peut être acceptable dans certains cas, mais elle peut faire réfléchir sur de gros projets où les performances sont cruciales et dans le cadre desquels il faut optimiser au maximum les requêtes effectuées.

Fonctionnalités annexes, en dehors des bases de JPA

La plupart des *frameworks* existant fournissent des fonctionnalités intéressantes n'existant pas (encore) dans le standard JPA, et permettent ainsi au développeur de s'affranchir de certaines contraintes, souvent chronophages lorsqu'elles doivent être prises en compte manuellement.

Envergure du projet

Indubitablement, sur un projet de grande taille, l'utilisation d'un *framework* ORM entraîne un gain en temps de développement important.

Contexte du projet

En règle générale, si le modèle de données d'un projet est préexistant, alors il est probablement plus sage de partir sans ORM. À l'opposé, si le modèle de données d'un projet change fréquemment durant son développement, alors partir sur un ORM offre un avantage considérable, avec la possibilité de regénérer entièrement et automatiquement la couche d'accès aux données.

Performances attendues

Dans des projets où la performance est la préoccupation principale, il est plus sage de ne pas utiliser d'ORM. Je me permets ici un gros raccourci, mais dans ce contexte il est important de noter qu'un ORM n'est pas rapide, et que son tuning est complexe.

Possibilité de mixer avec et sans ORM

Dans l'absolu, il ne faut pas perdre de vue qu'il est possible d'utiliser JPA pour tout ce qui est simple et barbant - les actions de type CUD et certaines actions de lecture - et du SQL fait main pour le reste. Il suffit pour cela d'une couche d'abstraction, un DAO par exemple, et il devient alors aisément de ne conserver que les avantages des deux solutions.

Solutions alternatives

Il ne faut pas perdre de vue que d'autres moyens existent. C'est bien souvent une fausse bonne idée, mais il arrive que certaines grandes boîtes, sur certains grands projets, créent leur propre *framework* de persistance. En outre, des solutions plus légères et transparentes comme MyBatis reprennent ce que les développeurs apprécient dans un ORM, tout en conservant une transparence au niveau du SQL recherchée par bon nombre d'entre eux.

Voilà les principales pistes que vous serez amenés à explorer si vous êtes, un jour, contraints de vous poser la question "ORM, ou pas ORM ?". Si je ne devais utiliser qu'un seul argument pour aiguiller votre choix, je vous conseillerais de vous poser la question suivante : est-ce que vous souhaitez passer une grande partie de votre temps sur la couche d'accès aux données, ou est-ce que vous préférez consacrer votre temps et votre énergie à développer une couche métier efficace ?

Enfin pour terminer et faire un aparté sur votre apprentissage, sachez que de très petits projets comme nous en créons dans le cadre de ce cours sont des opportunités en or pour apprendre calmement et efficacement comment fonctionne une solution de persistance. N'attendez surtout pas d'être parachutés sur des projets immenses ou même simplement



— déjà bien entamés, sans quoi vous allez passer un temps fou à réellement saisir tout ce qui mérite d'être assimilé.

Nous allons nous arrêter ici en ce qui concerne la persistance de données. C'est à la fois trop et pas assez : trop, parce que beaucoup de concepts intervenant dans la bonne appréhension d'un tel système sont inconnus des novices, et pas assez parce que, même débutants, je suis certain que vous souhaitez en apprendre davantage sur cet aspect du développement !

Rassurez-vous, je ne vous abandonne pas en cours de route : comme je vous l'ai déjà expliqué, avec ce que vous avez appris dans ce chapitre - que vous allez par ailleurs mettre en pratique et compléter dans un cas plus complexe dans l'étape suivante du TP Fil rouge - vous avez les notions nécessaires pour voler de vos propres ailes ! Sans aucun doute, pour être complet sur le sujet, il faudrait un cours à part entière.

- JPA est un **standard** appartenant à Java EE, définissant un système de persistance de données.
- Hibernate, EclipseLink et consorts sont des *frameworks* ORM, des solutions qui **implémentent** ce standard.
- JPA masque intégralement le moyen de stockage au développeur, en lui permettant de travailler uniquement sur un **modèle objet**. C'est le framework qui se charge dans les coulisses de dialoguer avec la BDD pour assurer la correspondance avec le modèle.
- L'objet qui représente une table de la BDD est un **EJB** de type **Entity**. C'est un JavaBean déclaré auprès du conteneur par `@Entity`, et contenant un ensemble d'annotations sur ses propriétés, permettant de définir les contraintes et relations existantes.
- L'objet qui fournit des méthodes d'interaction avec la BDD est l'**EntityManager**, et il est entièrement géré par le conteneur.
- Le fichier de configuration de JPA se nomme **persistence.xml**, et doit être placé dans le répertoire **src/META-INF**.
- C'est une bonne pratique d'utiliser un **EJB Stateless** en guise de DAO. Il suffit alors d'y injecter l'EntityManager via `@PersistenceContext`, et d'appeler ses différentes méthodes d'interaction.
- Contrairement à un **EJB Stateful**, un EJB Stateless peut être injecté sans risques dans une servlet via `@EJB`.
- Les objets et EJB « injectés » sont entièrement gérés par le conteneur, le développeur ne fait que les utiliser.
- Le détail des requêtes SQL effectuées sur la BDD par le *framework* ORM peut être enregistré dans le fichier de logs du serveur, en activant le mode **logging** dans le fichier persistence.xml de l'application.
- Un *framework* ORM n'est ni une solution magique et réponse à tout, ni une machinerie lourde et inutilisable : une analyse rapide du contexte d'un projet permet de définir s'il est intéressant ou non d'y faire intervenir une telle solution.

TP Fil rouge - Étape 7

Septième et dernière étape du fil rouge : vous allez pouvoir pratiquer JPA dans un contexte un peu plus compliqué que celui du cours, car faisant intervenir un modèle de données plus complexe à appréhender. Vous en profiterez également pour travailler avec vos nouveaux outils : GlassFish et BoneCP !

Objectifs

Fonctionnalités

Vous êtes équipés pour refondre votre TP en utilisant JPA. Vous allez devoir :

- migrer votre projet sur GlassFish ;
- mettre en place un pool de connexions pour votre projet avec BoneCP ;
- reprendre le code existant pour y mettre en place les annotations, configurations et classes nécessaires au bon fonctionnement de JPA ;
- supprimer les classes devenues obsolètes.

En apparence, tout cela paraît bien léger, mais vous allez faire face à quelques petits obstacles qui vous obligent à chercher, et à assimiler un peu mieux encore comment fonctionne JPA. Bon courage, et ne flanchez pas dans cette dernière ligne droite ! 😊

Conseils

Environnement de développement

Contexte du projet

Vous allez pouvoir réutiliser le serveur GlassFish mis en place dans le cadre du cours pour déployer votre projet. Faites une duplication de la correction du projet telle qu'elle était à l'issue de l'étape 6, afin de conserver des sources "propres". Pour migrer le projet vers votre nouveau serveur, vous devrez ensuite vous rendre dans le *build-path* du projet dupliqué, et modifier les points suivants :

- dans l'onglet "Libraries", remplacez celle qui mentionne Tomcat dans la liste affichée par celle de GlassFish en suivant Add Library > Server Runtime, puis en choisissant votre instance de GlassFish ;
- l'entrée intitulée "Targeted runtimes" dans les propriétés de votre projet ;
- et enfin l'entrée intitulée "Server" toujours dans les propriétés de votre projet.

Configuration du pool

Vous profiterez de l'occasion pour vous exercer à mettre en place un pool de connexions BoneCP sous GlassFish et à l'utiliser. Pour cela, vous pouvez essayer de modifier le fichier XML que je vous avais préparé dans le cadre du cours, afin qu'il cible cette fois, non plus la base **bdd_sdzee**, mais votre base **tp_sdzee**. Sinon, je vous ai préparé un fichier prêt à l'emploi que vous pouvez télécharger en cliquant [ici](#).

Pour le reste de la manipulation, vous pouvez vous reporter au cours si vous ne vous souvenez plus comment faire.

Configuration de l'application

Une fois le contexte en place, vous devrez ensuite créer un fichier **WEB-INF/glassfish-web.xml**, et un fichier **src/META-INF/persistence.xml**, comme nous l'avons fait dans le cours. Reportez-vous au chapitre précédent si vous avez des doutes.

Reprise du code existant

Suppression des objets devenus obsolètes

Vous allez pouvoir dès à présent vous débarrasser de la DAOFactory, du Listener l'initialisant et des utilitaires DAO.

Transformation des JavaBeans en EJB Entity

Vous allez ensuite devoir insérer des annotations JPA dans vos beans, comme nous l'avons fait dans le cours.

Toutefois, vous allez vite vous rendre compte qu'un champ est différent des autres : celui qui correspond à une clé étrangère dans la table **Commande**. Pour le gérer correctement, il y a une particularité à prendre en compte : il va falloir préciser au conteneur quel type de relation existe entre le champ et la table ciblée par la clé. Vous pouvez chercher par vous-mêmes si vous vous en sentez capables, ou suivre les indications suivantes sinon.

Il existe plusieurs types de relation entre des champs de tables relationnelles : un à un, un à plusieurs, ou plusieurs à un. Ceci relevant du design de la BDD et clairement pas du développement Java EE, je ne vous demande pas de comprendre exactement de quoi il est question ici. En l'occurrence, dans notre TP nous avons affaire à une relation de type plusieurs à un, car plusieurs entrées de la table **Commande** peuvent pointer vers un même **Client**. Il faut donc utiliser une annotation JPA prévue à cet effet, nommée `@ManyToOne`. En complément, il faut également expliciter le champ de la table visé par la clé étrangère, via l'annotation nommée `@JoinColumn(name = "...")`.

Autre petite difficulté, le fait que nous utilisions un champ de type **DateTime** dans notre entité **Commande**. Les développeurs de la bibliothèque JodaTime ont pris la peine d'écrire une classe pour convertir un tel objet vers une date utilisable par le *framework* **Hibernate**, une solution ORM très répandue. Cependant ils n'ont pas fait le travail pour **EclipseLink**, qui est pourtant le *framework* utilisé par défaut par GlassFish. Ainsi, vous allez devoir vous occuper vous-mêmes de la conversion... Pas d'inquiétude cependant, car ce petit développement n'est pas l'objectif de ce TP. Ainsi, je vous propose une classe prête à l'emploi, réalisée par [Xandacona](#) et fournie sur [cette page de son blog](#) :

Code : Java - Conversion d'un objet DateTime pour EclipseLink

```
package com.sdzee.tp.tools;

import java.sql.Timestamp;

import org.eclipse.persistence.mappings.DatabaseMapping;
import org.eclipse.persistence.mappings.converters.Converter;
import org.eclipse.persistence.sessions.Session;
import org.joda.time.DateTime;

public class JodaDateTimeConverter implements Converter {

    private static final long serialVersionUID = 1L;

    @Override
    public Object convertDataValueToObjectValue( Object dataValue,
Session session ) {
        return dataValue == null ? null : new DateTime( (Timestamp)
dataValue );
    }

    @Override
    public Object convertObjectValueToDataValue( Object objectValue,
Session session ) {
        return objectValue == null ? null : new Timestamp( (
DateTime) objectValue ).getMillis();
    }

    @Override
    public void initialize( DatabaseMapping mapping, Session session
) {
}

    @Override
    public boolean isMutable() {
        return false;
    }
}
```

Déposez cette classe nommée **JodaDateTimeConverter** dans un nouveau package intitulé `com.sdzee.tp.tools`, et vous pourrez ensuite à l'aide des annotations `@Converter` et `@Convert` préciser à votre conteneur comment effectuer la conversion de manière automatisée. Là encore, pas de panique, je vous montre comment annoter l'attribut **date** dans votre entité **Commande** :

Code : Java

```

@Column( columnDefinition = "TIMESTAMP" )
@Converter( name = "dateTimeConverter", converterClass =
JodaDateTimeConverter.class )
@Convert( "dateTimeConverter" )
private DateTime date;

```

Transformation des DAO en EJB Session Stateless

Annotations, **EntityManager** et méthodes d'accès à la BDD sont au programme. Cette fois par contre, à la différence du cours, puisqu'aucune requête de sélection complexe n'est réalisée, nous avons uniquement besoin de recherches basées sur les clés primaires de nos tables (leurs **id**). Ainsi, il ne vous sera pas nécessaire d'écrire des requêtes JPQL à la main, vous pourrez utiliser simplement la méthode `find()` de votre **EntityManager**. Vous voilà enfin débarrassés du SQL ! 😊

Modification des servlets

Dans les servlets qui manipulaient un DAO auparavant (pour la création de clients et de commandes), vous allez devoir supprimer les références aux anciens composants (DAOFactory), injecter vos EJB Stateless et les transmettre aux objets métier en lieu et place des DAO.

Annotation des servlets et des filtres

Vous allez pouvoir reprendre toutes les servlets et les annoter avec `@WebServlet`. De même pour le filtre de préchargement avec `@WebFilter`. Enfin, n'oubliez pas les servlets nécessitant une configuration multipart ou l'ajout de paramètres d'initialisation, avec les annotations `@MultipartConfig` et `@WebInitParam`.

Une fois toutes ces modifications apportées, vous pourrez vous débarrasser du fichier web.xml !

Modification des objets métier

Lorsque nous avons utilisé pour la première fois une requête de type **MultiPart**, je vous avais averti que GlassFish ne respectait pas correctement la spécification Java EE, et ne permettait pas d'appeler directement la méthode `request.getParameter()` pour obtenir un paramètre classique contenu dans une requête de type **MultiPart**. Heureusement pour nous, la dernière version du serveur - la 3.1.2.2 que je vous ai fait télécharger dans le chapitre précédent - a corrigé ce léger souci, et il est désormais possible de réaliser ce genre d'appels.

Autrement dit, vous n'avez rien à changer dans vos objets métier ! 😊

Correction

Faites attention à bien reprendre les fichiers du cours qui restent inchangés, à corriger les classes qui nécessitent des ajustements, et à supprimer celles qui ne vous servent plus à rien. Et surtout ne laissez pas les bras devant la charge de travail impliquée par l'intégration de JPA dans votre application ! Cet exercice est l'occasion parfaite pour vous familiariser avec les bases de ces concepts avancés !

Une fois n'est pas coutume, ce n'est pas la seule manière de faire, le principal est que votre solution respecte les consignes que je vous ai données !

Le code de configuration

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

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE glassfish-web-app PUBLIC "-//GlassFish.org//DTD GlassFish
Application Server 3.1 Servlet 3.0//EN"
"http://glassfish.org/dtds/glassfish-web-app_3_0-1.dtd">
<glassfish-web-app>
    <context-root>/tp7</context-root>
    <class-loader delegate="true"/>
    <jsp-config>
        <property name="keepgenerated" value="true">

```

```

<description>Keep a copy of the generated servlet class java
code.</description>
</property>
</jsp-config>
</glassfish-web-app>
```

Code : XML - src/META-INF/persistence.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
  xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
  http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
  <persistence-unit name="tp_sdzee_PU" transaction-type="JTA">
    <jta-data-source>jdbc/bonecp_resource_tp</jta-data-source>
    <class>com.sdzee.entities.Client</class>
    <class>com.sdzee.entities.Commande</class>
    <properties/>
  </persistence-unit>
</persistence>
```

Le code des EJB Entity

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

```

package com.sdzee.tp.entities;

import java.io.Serializable;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Client implements Serializable {

    @Id
    @GeneratedValue( strategy = GenerationType.IDENTITY )
    private Long id;
    private String nom;
    private String prenom;
    private String adresse;
    private String telephone;
    private String email;
    private String image;

    public void setId( Long id ) {
        this.id = id;
    }

    public Long getId() {
        return id;
    }

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

public String getTelephone() {
    return telephone;
}

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

public String getEmail() {
    return email;
}

public void setImage( String image ) {
    this.image = image;
}

public String getImage() {
    return image;
}
}
```

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

```
package com.sdzee.tp.entities;

import java.io.Serializable;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;

import org.eclipse.persistence.annotations.Convert;
import org.eclipse.persistence.annotations.Converter;
import org.joda.time.DateTime;

import com.sdzee.tp.tools.JodaDateTimeConverter;

@Entity
public class Commande implements Serializable {
```

```
public class Commande implements Serializable {
    @Id
    @GeneratedValue( strategy = GenerationType.IDENTITY )
    private Long id;
    @ManyToOne
    @JoinColumn( name = "id_client" )
    private Client client;
    @Column( columnDefinition = "TIMESTAMP" )
    @Converter( name = "dateTimeConverter", converterClass =
JodaDateTimeConverter.class )
    @Convert( "dateTimeConverter" )
    private DateTime date;
    private Double montant;
    @Column( name = "mode_paiement" )
    private String modePaiement;
    @Column( name = "statut_paiement" )
    private String statutPaiement;
    @Column( name = "mode_livraison" )
    private String modeLivraison;
    @Column( name = "statut_livraison" )
    private String statutLivraison;

    public Long getId() {
        return id;
    }

    public void setId( Long id ) {
        this.id = id;
    }

    public Client getClient() {
        return client;
    }

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

    public DateTime getDate() {
        return date;
    }

    public void setDate( DateTime 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 EJB Session

Code : Java - com.sdzee.tp.dao.ClientDao

```

package com.sdzee.tp.dao;

import java.util.List;

import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.TypedQuery;

import com.sdzee.tp.entities.Client;

@Stateless
public class ClientDao {

    // Injection du manager, qui s'occupe de la connexion avec la
    BDD
    @PersistenceContext( unitName = "tp_sdzee_PU" )
    private EntityManager em;

    public Client trouver( long id ) throws DAOException {
        try {
            return em.find( Client.class, id );
        } catch ( Exception e ) {
            throw new DAOException( e );
        }
    }

    public void creer( Client client ) throws DAOException {
        try {
            em.persist( client );
        } catch ( Exception e ) {
            throw new DAOException( e );
        }
    }

    public List<Client> lister() throws DAOException {
        try {
            TypedQuery<Client> query = em.createQuery( "SELECT c
FROM Client c ORDER BY c.id", Client.class );
            return query.getResultList();
        } catch ( Exception e ) {

```

```

        throw new DAOException( e );
    }

    public void supprimer( Client client ) throws DAOException {
        try {
            em.remove( em.merge( client ) );
        } catch ( Exception e ) {
            throw new DAOException( e );
        }
    }
}

```

Code : Java - com.sdzee.tp.dao.CommandeDao

```

package com.sdzee.tp.dao;

import java.util.List;

import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.TypedQuery;

import com.sdzee.tp.entities.Commande;

@Stateless
public class CommandeDao {

    // Injection du manager, qui s'occupe de la connexion avec la
    BDD
    @PersistenceContext( unitName = "tp_sdzee_PU" )
    private EntityManager em;

    public Commande trouver( long id ) throws DAOException {
        try {
            return em.find( Commande.class, id );
        } catch ( Exception e ) {
            throw new DAOException( e );
        }
    }

    public void creer( Commande commande ) throws DAOException {
        try {
            em.persist( commande );
        } catch ( Exception e ) {
            throw new DAOException( e );
        }
    }

    public List<Commande> lister() throws DAOException {
        try {
            TypedQuery<Commande> query = em.createQuery( "SELECT c
FROM Commande c ORDER BY c.id", Commande.class );
            return query.getResultList();
        } catch ( Exception e ) {
            throw new DAOException( e );
        }
    }

    public void supprimer( Commande commande ) throws DAOException {
        try {
            em.remove( em.merge( commande ) );
        } catch ( Exception e ) {
            throw new DAOException( e );
        }
    }
}

```

}

Le code des servlets

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

```
package com.sdzee.tp.servlets;

import java.io.IOException;
import java.util.HashMap;
import java.util.Map;

import javax.ejb.EJB;
import javax.servlet.ServletException;
import javax.servlet.annotation.MultipartConfig;
import javax.servlet.annotation.WebInitParam;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

import com.sdzee.tp.dao.ClientDao;
import com.sdzee.tp.entities.Client;
import com.sdzee.tp.forms.CreationClientForm;

@WebServlet( urlPatterns = { "/creationClient" }, initParams =
@WebInitParam( name = "chemin", value = "/fichiers/images/" ) )
@MultipartConfig( location = "/tmp", maxFileSize = 2 * 1024 * 1024,
maxRequestSize = 10 * 1024 * 1024, fileSizeThreshold = 1024 * 1024 )
public class CreationClient extends HttpServlet {
    public static final String CHEMIN              = "chemin";
    public static final String ATT_CLIENT          = "client";
    public static final String ATT_FORM            = "form";
    public static final String SESSION_CLIENTS     = "clients";

    public static final String VUE_SUCES          = "/WEB-
INF/afficherClient.jsp";
    public static final String VUE_FORM           = "/WEB-
INF/creerClient.jsp";

    @EJB
    private ClientDao clientDao;

    public void doGet( HttpServletRequest request,
HttpServletResponse response ) throws ServletException, IOException
{
    /* À la réception d'une requête GET, simple affichage du
formulaire */
    this.getServletContext().getRequestDispatcher( VUE_FORM
).forward( request, response );
}

    public void doPost( HttpServletRequest request,
HttpServletResponse response ) throws ServletException, IOException
{
    /*
    * Lecture du paramètre 'chemin' passé à la servlet via la
déclaration
    * dans le web.xml
    */
    String chemin = this.getServletConfig().getInitParameter(
```

```

CHEMIN );

    /* Préparation de l'objet formulaire */
    CreationClientForm form = new CreationClientForm( clientDao
);

    /* Traitement de la requête et récupération du bean en
résultant */
    Client client = form.creerClient( request, chemin );

    /* Ajout du bean et de l'objet métier à l'objet requête */
    request.setAttribute( ATT_CLIENT, client );
    request.setAttribute( ATT_FORM, form );

    /* Si aucune erreur */
    if ( form.getErreurs().isEmpty() ) {
        /* Alors récupération de la map des clients dans la
session */
        HttpSession session = request.getSession();
        Map<Long, Client> clients = (HashMap<Long, Client>)
session.getAttribute( SESSION_CLIENTS );
        /* Si aucune map n'existe, alors initialisation d'une
nouvelle map */
        if ( clients == null ) {
            clients = new HashMap<Long, Client>();
        }
        /* Puis ajout du client courant dans la map */
        clients.put( client.getId(), client );
        /* Et enfin (ré)enregistrement de la map en session */
        session.setAttribute( SESSION_CLIENTS, clients );

        /* 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 : Java - com.sdzee.tp.servlets.CreationCommande

```
package com.sdzee.tp.servlets;

import java.io.IOException;
import java.util.HashMap;
import java.util.Map;

import javax.ejb.EJB;
import javax.servlet.ServletException;
import javax.servlet.annotation.MultipartConfig;
import javax.servlet.annotation.WebInitParam;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

import com.sdzee.tp.dao.ClientDao;
import com.sdzee.tp.dao.CommandeDao;
import com.sdzee.tp.entities.Client;
import com.sdzee.tp.entities.Commande;
import com.sdzee.tp.forms.CreationCommandeForm;
```

```
@WebServlet( urlPatterns = { "/creationCommande" }, initParams =  
@WebInitParam( name = "chemin", value = "/fichiers/images/" ) )  
@MultipartConfig( location = "/tmp", maxFileSize = 2 * 1024 * 1024,  
maxRequestSize = 10 * 1024 * 1024, fileSizeThreshold = 1024 * 1024 )  
public class CreationCommande extends HttpServlet {  
    public static final String CHEMIN = "chemin";  
    public static final String ATT_COMMANDE = "commande";  
    public static final String ATT_FORM = "form";  
    public static final String SESSION_CLIENTS = "clients";  
    public static final String APPLICATION_CLIENTS = "clients";  
    "initClients";  
    public static final String SESSION_COMMANDES = "commandes";  
    public static final String APPLICATION_COMMANDES = "commandes";  
    "initCommandes";  
  
    public static final String VUE_SUCCES = "/WEB-INF/afficherCommande.jsp";  
    public static final String VUE_FORM = "/WEB-INF/creerCommande.jsp";  
  
    @EJB  
    private ClientDao clientDao;  
    @EJB  
    private CommandeDao commandeDao;  
  
    public void doGet( HttpServletRequest request,  
HttpServletResponse response ) throws ServletException, IOException  
{  
    /* À la réception d'une requête GET, simple affichage du  
formulaire */  
    this.getServletContext().getRequestDispatcher( VUE_FORM  
).forward( request, response );  
}  
  
    public void doPost( HttpServletRequest request,  
HttpServletResponse response ) throws ServletException, IOException  
{  
    /*  
     * Lecture du paramètre 'chemin' passé à la servlet via la  
déclaration  
     * dans le web.xml  
    */  
    String chemin = this.getServletConfig().getInitParameter(  
CHEMIN );  
  
    /* Préparation de l'objet formulaire */  
    CreationCommandeForm form = new CreationCommandeForm(  
clientDao, commandeDao );  
  
    /* Traitement de la requête et récupération du bean en  
résultant */  
    Commande commande = form.creerCommande( request, chemin );  
  
    /* Ajout du bean et de l'objet métier à l'objet requête */  
    request.setAttribute( ATT_COMMANDE, commande );  
    request.setAttribute( ATT_FORM, form );  
  
    /* Si aucune erreur */  
    if ( form.getErreurs().isEmpty() ) {  
        /* Alors récupération de la map des clients dans la  
session */  
        HttpSession session = request.getSession();  
        Map<Long, Client> clients = (HashMap<Long, Client>)  
session.getAttribute( SESSION_CLIENTS );  
        /* Si aucune map n'existe, alors initialisation d'une  
nouvelle map */  
        if ( clients == null ) {  
            clients = new HashMap<Long, Client>();  
        }  
    }  
}
```

```

    /* Puis ajout du client de la commande courante dans la
map */
    clients.put( commande.getClient().getId(),
commande.getClient() );
    /* Et enfin (ré)enregistrement de la map en session */
    session.setAttribute( SESSION_CLIENTS, clients );

    /* Ensuite récupération de la map des commandes dans la
session */
    Map<Long, Commande> commandes = (HashMap<Long,
Commande>) session.getAttribute( SESSION_COMMANDES );
    /* Si aucune map n'existe, alors initialisation d'une
nouvelle map */
    if ( commandes == null ) {
        commandes = new HashMap<Long, Commande>();
    }
    /* Puis ajout de la commande courante dans la map */
    commandes.put( commande.getId(), commande );
    /* Et enfin (ré)enregistrement de la map en session */
    session.setAttribute( SESSION_COMMANDES, commandes );

    /* 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 : Java - com.sdzee.tp.servlets.Image

```
package com.sdzee.tp.servlets;

import java.io.BufferedReader;
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.net.URLDecoder;

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

@WebServlet( urlPatterns = { "/images/*" }, initParams =
@WebServletParam( name = "chemin", value = "/fichiers/images/" ) )
public class Image extends HttpServlet {
    public static final int TAILLE_TAMPON = 10240; // 10ko

    public void doGet( HttpServletRequest request,
HttpServletResponse response ) throws ServletException, IOException
{
    /*
     * Lecture du paramètre 'chemin' passé à la servlet via la
déclaration
     * dans le web.xml
    */
    String chemin = this.getServletConfig().getInitParameter(
"chemin" );
```

```
/*
 * Récupération du chemin du fichier demandé au sein de l'URL de la
 * requête
 */
String fichierRequis = request.getPathInfo();

/* Vérifie qu'un fichier a bien été fourni */
if ( fichierRequis == null || "/" .equals( fichierRequis ) )
{
    /*
     * Si non, alors on envoie une erreur 404, qui signifie que la
     * ressource demandée n'existe pas
    */
    response.sendError( HttpServletResponse.SC_NOT_FOUND );
    return;
}

/*
 * Décode le nom de fichier récupéré, susceptible de contenir des
 * espaces et autres caractères spéciaux, et prépare l'objet File
 */
fichierRequis = URLDecoder.decode( fichierRequis, "UTF-8" );
File fichier = new File( chemin, fichierRequis );

/* Vérifie que le fichier existe bien */
if ( !fichier.exists() ) {
    /*
     * Si non, alors on envoie une erreur 404, qui signifie que la
     * ressource demandée n'existe pas
    */
    response.sendError( HttpServletResponse.SC_NOT_FOUND );
    return;
}

/* Récupère le type du fichier */
String type = getServletContext().getMimeType(
fichier.getName() );

/*
 * Si le type de fichier est inconnu, alors on initialise un type
 * par
 * défaut
 */
if ( type == null ) {
    type = "application/octet-stream";
}

/* Initialise la réponse HTTP */
response.reset();
response.setBufferSize( TAILLE_TAMPON );
response.setContentType( type );
response.setHeader( "Content-Length", String.valueOf(
fichier.length() ) );
response.setHeader( "Content-Disposition", "inline;
filename=\"" + fichier.getName() + "\"" );

/* Prépare les flux */
BufferedInputStream entree = null;
BufferedOutputStream sortie = null;
try {
    /* Ouvre les flux */
    entree = new BufferedInputStream( new FileInputStream(
fichier ), TAILLE_TAMPON );
    sortie = new BufferedOutputStream(
response.getOutputStream(), TAILLE_TAMPON );

    /* Lit le fichier et écrit son contenu dans la réponse
HTTP */
    byte[] tampon = new byte[TAILLE_TAMPON];
```

```
        int longueur;
        while ( ( longueur = entree.read( tampon ) ) > 0 ) {
            sortie.write( tampon, 0, longueur );
        }
    } finally {
        try {
            sortie.close();
        } catch ( IOException ignore ) {
        }
        try {
            entree.close();
        } catch ( IOException ignore ) {
        }
    }
}
```

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

```
package com.sdzee.tp.servlets;

import java.io.IOException;

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

@WebServlet( urlPatterns = { "/listeClients" } )
public class ListeClients extends HttpServlet {
    public static final String ATT_CLIENT = "client";
    public static final String ATT_FORM = "form";

    public static final String VUE = "/WEB-INF/listerClients.jsp";

    public void doGet( HttpServletRequest request,
HttpServletResponse response ) throws ServletException, IOException
{
        /* À la réception d'une requête GET, affichage de la liste
des clients */
        this.getServletContext().getRequestDispatcher( VUE
).forward( request, response );
    }
}
```

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

```
package com.sdzee.tp.servlets;

import java.io.IOException;

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

@WebServlet( urlPatterns = { "/listeCommandes" } )
public class ListeCommandes extends HttpServlet {
    public static final String ATT_COMMANDE = "commande";
    public static final String ATT_FORM      = "form";
```

```

    public static final String VUE          = "/WEB-
INF/listerCommandes.jsp";

    public void doGet( HttpServletRequest request,
HttpServletRequest response ) throws ServletException, IOException
{
    /* À la réception d'une requête GET, affichage de la liste
des commandes */
    this.getServletContext().getRequestDispatcher( VUE
).forward( request, response );
}
}

```

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

```

package com.sdzee.tp.servlets;

import java.io.IOException;
import java.util.HashMap;
import java.util.Map;

import javax.ejb.EJB;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

import com.sdzee.tp.dao.ClientDao;
import com.sdzee.tp.dao.DAOException;
import com.sdzee.tp.entities.Client;

@WebServlet( urlPatterns = { "/suppressionClient" } )
public class SuppressionClient extends HttpServlet {
    public static final String PARAM_ID_CLIENT = "idClient";
    public static final String SESSION_CLIENTS = "clients";

    public static final String VUE          = "/listeClients";

    @EJB
    private ClientDao           clientDao;

    public void doGet( HttpServletRequest request,
HttpServletRequest response ) throws ServletException, IOException
{
    /* Récupération du paramètre */
    String idClient = getValeurParametre( request,
PARAM_ID_CLIENT );
    Long id = Long.parseLong( idClient );

    /* Récupération de la Map des clients enregistrés en
session */
    HttpSession session = request.getSession();
    Map<Long, Client> clients = (HashMap<Long, Client>)
session.getAttribute( SESSION_CLIENTS );

    /* Si l'id du client et la Map des clients ne sont pas
vides */
    if ( id != null && clients != null ) {
        try {
            /* Alors suppression du client de la BDD */
            clientDao.supprimer( clients.get( id ) );
            /* Puis suppression du client de la Map */
            clients.remove( id );
        }
        catch ( DAOException e )
        {
            /* Gestion de l'exception */
        }
    }
}

```

```

        } catch ( DAOException e ) {
            e.printStackTrace();
        }
        /* Et remplacement de l'ancienne Map en session par la
nouvelle */
        session.setAttribute( SESSION_CLIENTS, clients );
    }

    /* Redirection vers la fiche récapitulative */
    response.sendRedirect( request.getContextPath() + VUE );
}

/*
* Méthode utilitaire qui retourne null si un paramètre est vide, et
son
* contenu sinon.
*/
private static String getValeurParametre( HttpServletRequest
request, String nomChamp ) {
    String valeur = request.getParameter( nomChamp );
    if ( valeur == null || valeur.trim().length() == 0 ) {
        return null;
    } else {
        return valeur;
    }
}

```

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

```
package com.sdzee.tp.servlets;

import java.io.IOException;
import java.util.HashMap;
import java.util.Map;

import javax.ejb.EJB;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

import com.sdzee.tp.dao.CommandeDao;
import com.sdzee.tp.dao.DAOException;
import com.sdzee.tp.entities.Commande;

@WebServlet( urlPatterns = { "/suppressionCommande" } )
public class SuppressionCommande extends HttpServlet {
    public static final String PARAM_ID_COMMANDE = "idCommande";
    public static final String SESSION_COMMANDES = "commandes";

    public static final String VUE
        =
"/listeCommandes";

    @EJB
    private CommandeDao commandeDao;

    public void doGet( HttpServletRequest request,
HttpServletResponse response ) throws ServletException, IOException
{
    /* Récupération du paramètre */
    String idCommande = getValeurParametre( request,
PARAM_ID_COMMANDE );
    Long id = Long.parseLong( idCommande );
    Commande commande = commandeDao.findById( id );
    if ( commande != null )
        commande.setSupprime( true );
    commandeDao.update( commande );
    HttpSession session = request.getSession();
    session.setAttribute( SESSION_COMMANDES, commande );
    response.sendRedirect( "/listeCommandes" );
}
}
```

```

        /* Récupération de la Map des commandes enregistrées en
session */
HttpSession session = request.getSession();
Map<Long, Commande> commandes = (HashMap<Long, Commande>)
session.getAttribute( SESSION_COMMANDES );

        /* Si l'id de la commande et la Map des commandes ne sont
pas vides */
if ( id != null && commandes != null ) {
    try {
        /* Alors suppression de la commande de la BDD */
        commandeDao.supprimer( commandes.get( id ) );
        /* Puis suppression de la commande de la Map */
        commandes.remove( id );
    } catch ( DAOException e ) {
        e.printStackTrace();
    }
    /* Et remplacement de l'ancienne Map en session par la
nouvelle */
    session.setAttribute( SESSION_COMMANDES, commandes );
}

        /* Redirection vers la fiche récapitulative */
response.sendRedirect( request.getContextPath() + VUE );
}

/*
* Méthode utilitaire qui retourne null si un paramètre est vide, et
son
* contenu sinon.
*/
private static String getValeurParametre( HttpServletRequest
request, String nomChamp ) {
    String valeur = request.getParameter( nomChamp );
    if ( valeur == null || valeur.trim().length() == 0 ) {
        return null;
    } else {
        return valeur;
    }
}
}
}

```

Le code du filtre

Code : Java - com.sdzee.tp.filters.PrechargementFilter

```

package com.sdzee.tp.filters;

import java.io.IOException;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import javax.ejb.EJB;
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.annotation.WebFilter;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;

```

```
import com.sdzee.tp.dao.ClientDao;
import com.sdzee.tp.dao.CommandeDao;
import com.sdzee.tp.entities.Client;
import com.sdzee.tp.entities.Commande;

@WebFilter( urlPatterns = { "/*" } )
public class PrechargementFilter implements Filter {
    public static final String ATT_SESSION_CLIENTS = "clients";
    public static final String ATT_SESSION_COMMANDES = "commandes";

    @EJB
    private ClientDao clientDao;
    @EJB
    private CommandeDao commandeDao;

    public void init( FilterConfig filterConfig ) throws
ServletException {
}

    public void doFilter( ServletRequest req, ServletResponse res,
FilterChain chain ) throws IOException,
ServletException {
    /* Cast de l'objet request */
    HttpServletRequest request = (HttpServletRequest) req;

    /* Récupération de la session depuis la requête */
    HttpSession session = request.getSession();

    /*
    * Si la map des clients n'existe pas en session, alors l'utilisateur
    * se
    * connecte pour la première fois et nous devons précharger en
    * session
    * les infos contenues dans la BDD.
    */
    if ( session.getAttribute( ATT_SESSION_CLIENTS ) == null ) {
        /*
        * Récupération de la liste des clients existants, et enregistrement
        * en session
        */
        List<Client> listeClients = clientDao.lister();
        Map<Long, Client> mapClients = new HashMap<Long,
Client>();
        for ( Client client : listeClients ) {
            mapClients.put( client.getId(), client );
        }
        session.setAttribute( ATT_SESSION_CLIENTS, mapClients );
    }

    /*
    * De même pour la map des commandes
    */
    if ( session.getAttribute( ATT_SESSION_COMMANDES ) == null ) {
        /*
        * Récupération de la liste des commandes existantes, et
        * enregistrement en session
        */
        List<Commande> listeCommandes = commandeDao.lister();
        Map<Long, Commande> mapCommandes = new HashMap<Long,
Commande>();
        for ( Commande commande : listeCommandes ) {
            mapCommandes.put( commande.getId(), commande );
        }
        session.setAttribute( ATT_SESSION_COMMANDES,
mapCommandes );
    }

    /* Pour terminer, poursuite de la requête en cours */
}
```

```
        chain.doFilter( request, res );  
    }  
  
    public void destroy() {  
    }  
}
```

Introduction aux frameworks MVC

Nous avons appris à limiter nos efforts sur la couche d'accès aux données avec JPA, nous allons maintenant découvrir comment alléger notre charge de travail sur les couches vue et contrôleur grâce aux **frameworks MVC** ! Au programme de ce chapitre d'introduction, des rappels sur le *pattern* MVC, des généralités sur le concept de *framework* et sur son intérêt, et le point sur le marché actuel des solutions existantes.

Généralités

Rappel concernant MVC

Revenons l'espace d'un court instant sur les objectifs du *pattern* MVC. Il s'agit, comme vous le savez tous maintenant, d'un modèle de conception, une bonne pratique qui décrit comment le code d'une application doit être organisé. Dans une application Java EE, il découpe littéralement le code en plusieurs couches, chacune étant chargée d'assurer des tâches bien définies : nous n'allons pas disserter davantage sur ce sujet, nous avons mis tout cela en pratique depuis le début du cours et vous connaissez déjà tous les composants constituant le modèle, le contrôleur et la vue.

Les principaux avantages d'un tel *pattern* sont évidents :

- la clarté introduite par un découpage clair et surtout standard des différentes sections d'une application permet une maintenance du code bien plus aisée que si le code ne respectait aucune règle préétablie. C'est cette quasi-universalité du mode de développement qui confère à MVC son intérêt le plus conséquent ;
- le découpage, et donc l'isolement des différentes tâches au sein d'une application, permet une meilleure répartition du travail entre les différents profils de développeurs. Le cas le plus souvent mis en avant est celui du designer web, qui peut ainsi ne s'occuper que de la vue sans avoir à se soucier de ce qui se passe derrière, ni même de comment cela se passe.

À ces avantages s'oppose toutefois un inconvénient majeur : puisqu'il y a nécessité de délimiter clairement les couches d'une application, il faut fatalement écrire davantage de code, et la structure ainsi définie impose des répétitions de code fréquentes. Est-il besoin de le préciser ? Qui dit plus de code, dit développement plus lent, risque d'erreurs plus grand, etc.

Qu'est-ce qu'un framework MVC ?

Comme vous le savez tous, la brique de base de la plate-forme Java EE est la servlet. Tout passe par elle, et même nos pages JSP sont transformées en servlets derrière les rideaux. Eh bien un *framework* MVC n'est rien d'autre qu'une surcouche à cette technologie de base. En masquant ces rouages les plus basiques, une telle solution facilite le découpage et la gestion du code d'une application. Elle intervient sur le cycle de vie d'une requête dans l'application, et prend en quelque sorte la main sur son cheminement. Voilà pourquoi on qualifie souvent les *frameworks* MVC de *frameworks* d'inversion de contrôle, parfois abrégé en IoC.

À quels besoins répond-il ?

Nous avons d'ores et déjà répondu à cette question en listant les avantages et inconvénients principaux du *pattern* MVC. L'objectif premier d'un *framework* MVC et de simplifier le flot d'exécution (ou *workflow*) d'une application, c'est-à-dire de :

- limiter les répétitions de code impliquées par la mise en place d'un découpage du code ;
- réaliser une partie du travail redondant à la place du développeur, en effectuant des tâches génériques derrière les rideaux, de manière transparente ;
- contraindre le développeur à respecter une organisation clairement définie. Alors qu'il reste libre d'implémenter comme bon lui semble le *pattern* MVC lorsqu'il travaille à la main, il doit se conformer à un format plus ou moins strict lorsqu'il fait usage d'un *framework* ;
- par corollaire du premier point, rendre le développement de certains aspects plus rapide, permettant ainsi au développeur de se concentrer sur le cœur de l'application.

Quand utiliser un framework MVC, et quand s'en passer ?

Avant de nous pencher sur les différents types de *frameworks* MVC existants, réfléchissons à la question suivante : qu'est-ce qui justifie l'utilisation d'un *framework* MVC dans une application ? Nous venons certes de découvrir les besoins auxquels il répond, mais ce n'est pas pour autant qu'il est toujours judicieux d'en utiliser un. Ainsi, même si l'intuition nous incite à

considérer qu'un *framework* ne peut apporter que du bon dans un projet, voici les principaux axes de réflexion à explorer :

- **l'envergure du projet** : si l'application développée est relativement petite, il n'est probablement pas nécessaire de mettre en place un *framework*, et le développement MVC "à la main" peut convenir ;
- **le temps d'apprentissage** : si vous ou votre équipe n'avez que peu de connaissances sur un *framework* MVC, alors il faut peser le pour et le contre entre le temps gagné durant le développement grâce à son utilisation, et le temps perdu en amont pour apprendre et assimiler la technologie ;
- **le contexte du projet** : si des contraintes sont formulées au niveau du serveur et des composants utilisables par un projet, et éventuellement au niveau des performances ou du niveau de scalabilité attendus, alors il faut déterminer si l'utilisation d'un *framework* rentre dans ce cadre ou non, et vérifier qu'il respecte bien les contraintes énoncées.

Framework MVC basé sur les requêtes

Définition

La première grande catégorie de *frameworks* MVC qualifie ceux qui se basent **sur les requêtes** ; on parle également de *frameworks* basés **sur les actions**.

C'est le type de *frameworks* MVC le plus simple à comprendre et à assimiler par les développeurs qui ont déjà de l'expérience avec le développement Java EE en suivant MVC, car il reprend dans les grandes lignes sensiblement les mêmes principes. Ce sont des solutions qui interviennent directement sur le cycle de vie d'une requête au sein de l'application (on rejoint ici cette histoire de "prise de contrôle").



Il serait nécessaire de nous y attarder plus longuement si nous n'avions jamais touché à MVC, mais puisque nous en avons fait le fer de lance de notre parcours, nous savons déjà très bien de quoi il est question ici !

Principe

Là encore, le principe est extrêmement simple à comprendre pour quiconque a déjà travaillé avec MVC. Il s'agit d'un *framework* web qui prend en entrée une requête issue d'un client, qui détermine ce que le serveur doit en faire et qui renvoie enfin au client une réponse en conséquence. Le flot d'exécution peut donc être qualifié de linéaire, et reprend ce que nous avons jusqu'à présent mis en place à la main dans nos exemples.

Ainsi, le développeur doit penser en termes d'actions, et associe naturellement ce que demande l'utilisateur (sa requête) à ce que l'utilisateur reçoit (la réponse). Concrètement, le développeur écrit des classes qui représentent des actions effectuées par l'utilisateur, comme par exemple "Passer une commande" ou "Voir les détails du client". Ces classes sont chargées de récupérer les données transmises via la requête HTTP, et de travailler dessus.

En somme, avec un tel *framework* le développeur travaille toujours de près ou de loin sur la paire requête/réponse, comme nous l'avons fait jusqu'à présent. Le gros changement à noter, mais nous y reviendrons plus tard, est la mise en place d'une **servlet unique** jouant le rôle d'aiguilleur géant (ou *Front Controller*). Celle-ci se charge de déléguer les actions et traitements au modèle métier en se basant sur l'URL de la requête et sur ses paramètres. Le développeur peut alors travailler directement sur les objets `HttpServletRequest` et `HttpServletResponse` bruts dans le modèle, ou bien utiliser le système de mapping fourni par le *framework*. Ce système permet au développeur de confier au *framework* les tâches de regroupement, conversion et validation des paramètres de requête, et si nécessaire de mettre à jour des valeurs du modèle de données, avant d'invoquer les actions métier. Enfin, il doit toujours écrire lui-même les pages - bien souvent des pages JSP - en charge de créer les réponses à renvoyer au client, et il jouit donc d'une liberté totale sur le rendu HTML/CSS/JS de chaque vue.

Solutions existantes

Trois grands acteurs se partagent le devant de la scène :

- **Spring**, solution très répandue dont le spectre s'étend de la simple gestion des requêtes à l'ensemble du cycle de vie de l'application, bien souvent couplée à Hibernate pour la persistance des données ;
- **Struts**, solution éditée par Apache dont la gloire appartient au passé. Le *framework* a changé du tout au tout avec la sortie d'une seconde mouture qui se nomme Struts 2, mais qui à part son titre n'a rien de similaire à Struts, premier du nom. Par ailleurs, la communauté maintenant le projet semble moins active que les concurrentes ;
- **Stripes**, un challenger intéressant et montant, très léger et réputé pour être facile à prendre en main.

Framework MVC basé sur les composants

Définition

La seconde grande catégorie de *frameworks* MVC qualifie ceux qui se basent non pas sur les requêtes, mais sur les composants.

À la différence de ceux basés sur les requêtes, les *frameworks* basés sur les composants découpent logiquement le code en "composants", masquant ainsi le chemin d'une requête au sein de l'application. Ils essaient en quelque sorte d'abstraire les concepts de requête et réponse, et de traiter une application comme une simple collection de composants qui présentent leur propre méthode de rendu et des actions pour effectuer des tâches.



Les ressources traitant de ce sujet dans la langue de Molière sont sur la toile étonnamment légères. Si vous êtes anglophones, vous trouverez cependant de quoi faire, de nombreuses discussions intéressantes et fournies étant disponibles dans la langue de Shakespeare. Pour information, sachez qu'on parle en anglais d'*action-based frameworks* et de *component-based frameworks*.

À l'origine, de telles solutions ont été développées pour faciliter aux développeurs Java, peu familiers avec le développement web et plus proches du développement d'applications de bureau traditionnelles, la création d'applications web, sans devoir connaître les rouages de l'API Servlet. En outre, de telles solutions ont pour objectif secondaire de rendre la maîtrise des technologies de présentation web habituellement requises superflues, notamment HTML, CSS et JS.

Principe

Dans le MVC basé sur les composants, une unique servlet jouant le rôle de *Front Controller* va elle-même regrouper, convertir et valider les paramètres de requête, et mettre à jour les valeurs du modèle. Le développeur n'a ainsi à se soucier que des actions métier. La façon dont le contrôleur regroupe/convertit/valide/met à jour les valeurs est définie dans un unique endroit, la vue. Puisque c'est impossible à réaliser à l'aide de simple HTML "pur", un langage similaire spécifique est requis pour y parvenir. Dans le cas de JSF, c'est un langage basé sur du XML (XHTML). Vous utilisez du XML pour définir les composants de l'interface utilisateur, qui eux contiennent des informations sur la manière dont le contrôleur doit regrouper/convertir/valider/mettre à jour les valeurs et générer lui-même le rendu HTML.

Les avantages et inconvénients doivent maintenant être clairs pour vous :

- avec un framework MVC basé sur les requêtes vous devez écrire plus de code vous-mêmes pour parvenir à vos fins. Cependant, vous avez un meilleur contrôle sur le processus, c'est-à-dire sur le cheminement d'une requête au sein de l'application, et sur le rendu HTML/CSS/JS ;
- avec un framework MVC basé sur les composants, vous n'avez pas besoin d'écrire autant de code vous-mêmes. Cependant, vous avez moins de possibilités de contrôle sur le processus et le rendu HTML/CSS/JS.

Donc, si vous souhaitez réaliser des choses qui s'écartent un peu de ce que décrit le standard, vous perdrez beaucoup plus de temps si vous utilisez un framework MVC basé sur les composants.

Solutions existantes

Une fois de plus, trois grands acteurs se partagent le devant de la scène :

- JSF, ou "Java Server Faces" : il s'agit de la solution standard intégrée à Java EE 6, et les trois prochains chapitres du cours lui sont dédiés ;
- Wicket : solution éditée par Apache, elle est très en vogue grâce à sa relative simplicité. La communauté en charge de sa maintenance est très active ;
- Tapestry : solution éditée par Apache, ayant connu une évolution assez chaotique. Durant les premières années de son existence, chaque nouvelle version du *framework* cassait la compatibilité avec les versions précédentes, forçant les développeurs qui l'utilisent à migrer et/ou réécrire le code de leurs applications pour rester à jour... A priori cette sombre époque est maintenant révolue, et la version actuelle a été construite pour être évolutive.

Les "dissidents"

Sachez pour conclure qu'il existe plusieurs solutions déjà très populaires qui ne se classent dans aucune de ces deux catégories, mais qui sont tout de même utilisées pour développer des applications web, et qui peuvent être utilisées en suivant le *pattern* MVC. On retiendra notamment :

- GWT, solution éditée par Google pour la création d'applications web de type client-serveur ;
- Play!, un framework plus global qui se place au même niveau que Java EE lui-même, et qui permet de développer avec les langages Java ou Scala.
- Un *framework* MVC est destiné à simplifier et accélérer le développement d'une application web, et à isoler clairement le travail à effectuer par les différents profils de développeurs.
- Utiliser un *framework* n'est pas toujours la solution optimale, il est parfois judicieux, voire nécessaire, de s'en passer.
- Un **framework** MVC basé sur les requêtes (ou sur les actions) reprend l'organisation et les principes de ce que nous avons développé à la main jusqu'à présent. Il simplifie l'aspect contrôleur via l'intervention d'une servlet unique jouant le rôle d'aiguilleur géant. La vue est la plupart du temps réalisée avec la technologie JSP.
- Un **framework** MVC basé sur les composants masque les échanges de requêtes et réponses derrière des composants autonomes, symbolisés par l'intermédiaire de balises dans la vue. Une servlet unique joue le rôle d'aiguilleur géant.
- Le standard adopté par Java EE 6 est un **framework** MVC basé sur les composants : la solution se nomme JSF, ou *Java Server Faces*.
- D'autres modes de développement web existent sur la plate-forme Java, et sont activement développés par de grands acteurs du web et de l'Internet.

Premiers pas avec JSF

Dans cette introduction au framework JSF, nous allons découvrir en quoi il consiste dans les grandes lignes, étudier ses composants de base, puis mettre en place un exemple très simple et enfin le comparer à son équivalent réalisé avec la technologie JSP.

Qu'est-ce que JSF ?

Présentation

Nous l'avons évoqué dans le chapitre précédent, JSF (JavaServer Faces) est un **framework MVC basé sur les composants**. Il est construit sur l'API Servlet et fournit des composants sous forme de bibliothèques de balises ressemblant très fortement à la JSTL. Celles-ci peuvent être utilisées dans des pages JSP comme dans toute autre technologie de vue basée sur le Java, car le framework JSF ne limite pas le développeur à une technologie particulière pour la vue. Cependant, il existe une technologie relativement récente baptisée la Facelet, qui fait partie du standard et qu'il est recommandé d'utiliser dès lors que l'on travaille avec JSF, celle-ci étant bien plus adaptée que les pages JSP. Ces dernières sont d'ailleurs considérées comme une technologie de présentation dépréciée pour JSF depuis la sortie de Java EE 6 en 2009.

Hormis les avantages inhérents à tout *framework* MVC basé sur les composants, que nous avons listés dans le chapitre précédent, JSF offre notamment de grandes capacités de création de *templates* (ou gabarits), telles que les composants composites.



Qu'est-ce que la création de templates ?

Il s'agit tout simplement de la possibilité de découper une page en plusieurs composants indépendants, assemblés ensuite pour former une page finale. À cet égard, la technologie JSP fournit seulement le tag `<jsp:include>`, que nous avons déjà rencontré dans notre TP pour une tâche très simple, à savoir inclure automatiquement un menu sur chacune de nos pages.

Pour rester concis, JSF permet comme la plupart des *frameworks* MVC de gérer les événements et la validation des saisies par l'utilisateur, de contrôler la navigation entre les pages, de créer des vues accessibles, de faciliter l'internationalisation des pages, etc. En outre, puisqu'il est basé sur les composants et non sur les requêtes, il permet de gérer un état pour une vue donnée. Pas de panique, nous allons revenir sur ces aspects importants en temps voulu et par la pratique.

Principe

JSF est un framework MVC et propose, en guise de contrôleur unique du cycle de vie du traitement des requêtes, la FacesServlet.



Un contrôleur unique ?

Eh oui vous avez bien lu, il n'y a qu'une seule servlet chargée d'aiguiller l'intégralité des requêtes entrantes vers les bons composants, et ce pour l'application tout entière ! Cette architecture porte un nom, il s'agit du *pattern* Front Controller, qui lui-même est une spécialisation du *pattern* Médiateur.

JSF vous évite d'avoir à écrire le code - standard, passe-partout et pénible - responsable du regroupement des saisies utilisateurs (paramètres de requêtes HTTP), de leur conversion & validation, de la mise à jour des données du modèle, de l'invocation d'actions métiers et de la génération de la réponse. Ainsi vous vous retrouvez uniquement avec une page JSP ou une page XHTML (une Facelet) en guise de vue, et un JavaBean en tant que modèle. Cela accélère le développement de manière significative ! Les composants JSF sont utilisés pour lier la vue avec le modèle, et la FacesServlet utilise l'arbre des composants JSF pour effectuer tout le travail.



Tout cela semble bien compliqué, mais en réalité si vous devez n'en retenir qu'une chose, c'est celle-ci : **avec JSF, vous n'avez plus besoin d'écrire de servlets** ! Ne soyez pas inquiets face à toutes ces nouveautés, nous allons y revenir calmement et en détail un peu plus loin dans ce chapitre.

Historique

Avant de passer à la suite, penchons-nous un instant sur le passé de la solution. Certains d'entre vous le savent peut-être déjà, JSF jouit d'une certaine réputation auprès des développeurs... une mauvaise réputation ! Pourtant, si l'on y regarde d'un peu plus

près, on se rend très vite compte que la plupart des reproches faits à JSF sont sans fondements.

Afin de casser cette mauvaise réputation, hormis lorsqu'il est question de la relative difficulté d'apprentissage quand on ne dispose pas de solides connaissances en Java EE (servlets notamment), nous allons faire un rapide état des lieux de l'évolution du *framework*, de sa naissance à aujourd'hui. Bien qu'actuellement dans sa version 2.1, il a toujours besoin de se débarrasser de l'image négative qu'il dégageait lorsqu'il n'était pas encore mature. Et c'est vrai que si l'on se penche sur son histoire, on relève des inconvénients majeurs...



Cet historique contient certaines informations et certains détails qui vous échapperont certainement à la première lecture, si vous n'avez jamais travaillé avec JSF par le passé. Ne vous y attardez pas pour le moment, tâchez simplement de saisir l'évolution générale du *framework*. Si vous rencontrez un jour un projet basé sur une ancienne version de JSF, vous trouverez ici les différences majeures avec la version actuelle et les inconvénients importants.

JSF 1.0 (mars 2004)

Ce fut la version initiale. Bien qu'estampillée 1.0, elle était pleine de bugs à la fois au niveau des performances et du noyau, vous n'imaginez pas. Les applications ne fonctionnaient pas toujours comme attendu. En tant que développeur, vous auriez fui cette solution en hurlant !

JSF 1.1 (mai 2004)

Il s'agissait ici d'une mise à jour destinée à corriger les bugs de la version initiale... Et les performances n'étaient toujours pas au rendez-vous. En outre, **énorme inconvénient : il était impossible d'insérer du HTML dans une page JSF** sans défauts ! Étrange décision dans un framework destiné à la création d'applications web que de laisser de côté les designers web... Pour faire court, l'intégralité du code HTML brut était rendue **avant** les balises JSF ! Il fallait alors entourer chaque portion de code HTML par une balise JSF (**<f:verbatim>**) pour qu'elle soit incluse correctement dans l'arbre des composants (l'enchaînement des balises JSF) et rendue proprement. Bien que cela respecte les spécifications écrites à l'époque, cela a valu au framework un flot de critiques ! Imaginez vous devoir entourer chaque balise HTML intervenant dans vos pages JSP par une autre balise, juste pour qu'elle soit correctement prise en compte... Bref, c'était une vraie horreur.

JSF 1.2 (mai 2006)

Ce fut la première version préparée par la nouvelle équipe de développement de JSF, menée par Ryan Lubke. L'équipe a fourni un travail énorme, et les spécifications ont beaucoup évolué. L'amélioration de la gestion de la vue a été le principal axe de changement. Elle n'a pas seulement rendu la vue JSF indépendante des JSP, mais elle a également permis aux développeurs d'insérer du HTML dans une page JSF sans avoir à dupliquer sans cesse ces satanées balises **<f:verbatim>** nécessaires dans les versions précédentes. Un autre centre d'intérêt de l'équipe à l'époque fut l'amélioration des performances. Presque chaque version corrective mineure était alors accompagnée d'améliorations notables des performances globales du framework.

Le seul réel inconvénient commun à toutes les versions JSF 1.x (1.2 inclus) était l'absence d'une portée se plaçant entre la requête et la session (celle qui est souvent nommée "scope de conversation"). Cela forçait les développeurs à jouer avec des champs de formulaires cachés, des requêtes sur la BDD non nécessaires et/ou à abuser des sessions à chaque fois que quelqu'un voulait retenir le modèle de données initial pour les requêtes suivantes afin de procéder aux validations/conversions/mises à jour du modèle et aux invocations d'actions dans des applications web plus complexes. En somme, le développeur devait faire comme nous lorsque nous travaillons à la main sans framework. Il n'avait aucun moyen standard pour donner un état à une vue en particulier. Ce problème pouvait à l'époque être partiellement évité en utilisant une bibliothèque tierce qui sauvegardait l'état des données nécessaires pour les requêtes suivantes, notamment la balise **<t:saveState>** de la bibliothèque **MyFaces Tomahawk**, ou encore le framework de conversation **MyFaces Orchestra**.

Un autre point très pénalisant pour les webdesigners était le fait que JSF utilisait le caractère : en tant que séparateur d'identifiant afin d'assurer l'unicité des id des éléments HTML générés lors du rendu des balises, notamment lorsqu'un composant était utilisé plus d'une fois dans la vue (création de templates, boucles sur les composants, etc.). Si vous connaissez un petit peu CSS, vous devez savoir que ce caractère n'est pas autorisé dans les identifiants CSS, et les *webdesigners* devaient alors utiliser le caractère \ pour échapper les : dans les sélecteurs CSS qu'ils mettaient en place, ce qui produisait des sélecteurs étranges et sales tels que **#formulaireId\ :champId { ... }** dans les feuilles CSS.

En outre, JSF 1.x n'était livré avec aucune fonctionnalité Ajax prête à l'emploi. Ce n'était pas vraiment un inconvénient technique, mais l'explosion du web 2.0 en a fait un inconvénient fonctionnel. Exadel introduisait alors rapidement la bibliothèque **Ajax4jsf**, qui a été activement développée durant les années suivantes puis intégrée au noyau de la bibliothèque de composants JBoss RichFaces. D'autres bibliothèques offrent depuis des composants ajaxisés, **IceFaces** étant probablement la plus connue.

Quand JSF 1.2 a atteint la moitié de sa vie, une nouvelle technologie pour la vue basée sur le XML a été introduite : les **Facelets**. Cela a offert d'énormes avantages sur les JSP, particulièrement pour la création de templates et pour les performances.

JSF 2.0 (juin 2009)

Ce fut la seconde version majeure. Il y eut énormément de changements à la fois techniques et fonctionnels. La technologie JSP a été remplacée par les Facelets en tant que technologie de vue par défaut, et aux Facelets ont été greffées des fonctionnalités permettant la création de composants purement XML (appelés composants composites). Ajax a été introduit notamment, via un composant qui montre des similarités avec Ajax4jsf. Les annotations et améliorations favorisant le concept convention-plutôt-que-configuration ont été introduites pour éliminer les volumineux et verbeux fichiers de configuration autant que possible. En outre, le caractère séparateur d'identifiant : est devenu configurable.

Dernière évolution, mais non des moindres, une nouvelle portée a été introduite : le **scope view**, se plaçant entre la requête et la session. Cela a pallié un autre inconvénient majeur de JSF 1.x, comme nous l'avons vu précédemment. Nous allons y revenir en détail dans un prochain exemple pratique.

Jusque-là, tout paraît idyllique, et cette version estampillée 2.0 semble salvatrice. Cependant, bien que la plupart des inconvénients de JSF 1.x ont disparu avec cette version, il subsiste des bugs spécifiques à JSF 2.0 qui peuvent être un facteur bloquant dans le développement d'une application. Si le cœur vous en dit, vous pouvez jeter un œil aux [bugs JSF 2.x en cours](#) pour vous faire une idée, mais ils concernent une utilisation avancée du *framework*, et la plupart de ces bugs sont de toute manière contournables.

Enfin pour terminer sur une note plus gaie, c'est avec JSF 2.0 que sont apparues de nouvelles bibliothèques de composants plus graphiques. On peut notamment citer l'impressionnant [PrimeFaces](#) et [OpenFaces](#).

MVC basé sur les composants vs MVC basé sur les requêtes

Certains avancent que le plus gros inconvénient de JSF est qu'il n'autorise que peu de contrôle sur le code HTML/CSS/JS généré. Cela ne tient en réalité pas à JSF en lui-même, mais simplement au fait que c'est un framework MVC basé sur les composants, et pas sur les requêtes (actions). Si vous êtes perdus, je vous renvoie au chapitre précédent pour les définitions.

En réalité, c'est très simple : si un haut niveau de contrôle sur le rendu HTML/CSS/JS est votre principale exigence lorsque vous choisissez un framework MVC, alors vous devez regarder du côté des frameworks basés sur les requêtes comme [Spring MVC](#).

Après ce long aparté sur l'histoire de JSF, dont vous pouvez trouver la version originale anglaise sur [cette page](#), résumons la situation en quelques lignes :

- JSF 2.0 est un *framework* web MVC qui se focalise sur la simplification de la construction d'interfaces utilisateur, autrement dit la vue. Il propose nativement plus d'une centaine de balises à cet égard, et facilite la réutilisation des composants d'une interface utilisateur à l'autre ;
- avec JSF 1, tout devait être déclaré dans un fichier de configuration, et le développeur devait donc maintenir un énième fichier XML... Avec JSF 2, c'en est terminé de ces fichiers (inter)minables : les annotations remplacent les fichiers externes, et le développement s'en retrouve donc grandement accéléré et simplifié !
- dans ce cours, je ne ferai dorénavant plus jamais la distinction entre JSF 1 et JSF 2, tout bonnement parce que JSF aujourd'hui, c'est JSF 2 et rien d'autre. Dans votre tête, cela doit être tout aussi clair : JSF 1 est de l'histoire ancienne, et je n'y ferai allusion que ponctuellement pour vous informer des différences importantes avec la version actuelle, afin que vous ne soyez pas trop surpris si jamais vous devez un jour travailler sur une application développée avec JSF 1 (ce que je ne vous souhaite bien évidemment pas !).

 Les ressources **périmées** concernant JSF sont légion sur le web ! Énormément de cours, tutos, sujets de forums et documentations traitent de JSF 1, alors que JSF 2 a changé énormément de choses. Faites bien attention aux liens que vous parcourez, et basez-vous sur les dates clés de l'histoire de JSF pour déterminer si les informations que vous lisez sont d'actualité ou non. En règle générale, vous devez donc vous méfier de tout ce qui a été publié avant 2009 !

Structure d'une application JSF

Étudions brièvement comment se construit une application basée sur JSF. Cela ne devrait pas bouleverser vos habitudes, la structure globale est très similaire à celle d'une application web Java EE MVC traditionnelle :

- la vue est généralement assurée par des pages JSP ou par des pages XHTML (on parle alors de Facelets) ;
- le modèle est assuré par des entités ou des JavaBeans ;
- le contrôleur, auparavant incarné par nos servlets, est dorénavant décomposé en deux éléments :
 - une unique servlet mère servant de point d'entrée à toute requête, la FacesServlet ;
 - un JavaBean particulier, déclaré via une annotation et désigné par le terme *managed-bean*.
- le tout est mis en musique par des fichiers de configuration : le classique web.xml, mais également un nouveau fichier nommé faces-config.xml.

Nous observons que la différence majeure qui ressort jusqu'à présent est l'absence de servlets spécifiques à chaque page, comme nous devions en écrire dans nos applications MVC faites maison.

Facelets et composants

La page JSP mise au placard ?

Comme je vous l'ai annoncé à plusieurs reprises déjà, la page JSP n'est plus la technologie de référence pour la création de vues dans une application Java EE. Avec JSF, ce sont maintenant de simples pages XHTML qui sont utilisées, pages que l'on nomme des **Facelets**. Partant de ce constat, si vous avez bien suivi, vous êtes en droit de vous poser la question suivante :



Nous avons déjà découvert que les servlets n'allait plus être nécessaires dans notre application, voilà maintenant que les JSP non plus ne vont plus nous servir... Pourquoi avoir passé des chapitres entiers à apprendre à créer et manier des servlets et des JSP, si c'est pour les laisser tomber ensuite ?

Premièrement, **ces technologies sont loin d'être à la retraite**, car comme nous l'avons déjà abordé les *frameworks* MVC ne sont pas utilisés partout. En outre, il existe dans le monde de l'entreprise énormément d'applications basées sur les servlets et les JSP, applications qui ne sont pas prêtes d'être migrées ni réécrites vers un *framework* MVC, bien souvent pour des raisons de coût. Plus important encore, il existe **énormément** d'applications basées sur des *frameworks* MVC différents de JSF, notamment le très populaire Spring MVC, qui utilisent encore massivement les pages JSP en guise de vue.

Deuxièmement, l'apprentissage des *frameworks* et de leur fonctionnement est bien plus facile lorsque vous disposez déjà des bases. Bien entendu, dans l'absolu il est possible d'attaquer directement par les solutions de haut niveau - c'est-à-dire celles qui masquent les technologies de base comme les servlets, les JSP ou encore JDBC - mais tôt ou tard vous devrez regarder sous la couverture... Et quand ce jour arrivera, vous serez alors bien plus aptes à comprendre les rouages de toute cette machinerie en connaissant les fondements que si vous découvriez le tout pour la première fois !

Enfin, dans les cas d'applications de faible envergure, d'applications nécessitant d'excellentes performances et d'applications devant être aisément déployables à très large échelle, il est encore relativement courant de ne pas utiliser de *frameworks*. Bref, assurez-vous, vous n'avez pas perdu votre temps en arrivant jusqu'ici ! 😊

Structure et syntaxe

Nous allons aborder la structure d'un Facelet en la comparant avec la structure d'une technologie que nous connaissons déjà : la page JSP.

Généralités

La différence la plus importante se situe au niveau du contenu. Alors qu'une page JSP est un objet complexe, pouvant contenir à la fois des balises JSP ou JSTL et du code Java par l'intermédiaire de scriptlets, **une Facelet est un fichier XML pur** ! Elle ne peut par conséquent contenir que des balises, et il est impossible d'y inclure des scriptlets : le Java est donc définitivement banni de la vue avec cette technologie.

Au niveau du format et du rendu, la situation n'a pas changé : tout comme les servlets et les pages JSP sont le plus souvent utilisées pour générer des pages HTML, c'est le langage XHTML qui est le plus utilisé pour créer des Facelets (XHTML étant une version de la syntaxe HTML conforme au standard XML). Toutefois, sachez que le *framework* JSF est *markup-agnostique* : cela signifie qu'il peut s'adapter à n'importe quel langage, à partir du moment où ce langage respecte la structure décrite par le standard XML. Ainsi, il est également possible de construire une Facelet avec du langage XML pur, par exemple pour créer un flux RSS, ou avec le langage XUL pour n'en citer qu'un.

Au niveau de l'aspect extérieur du fichier, une Facelet porte en général une de ces trois extensions : **.jsf**, **.xhtml** ou **.faces**. Il n'existe pas une unique extension pour la simple raison que nous venons de découvrir, à savoir le fait qu'une Facelet n'est rien

d'autre qu'un document XML. Dans l'absolu, c'est par défaut l'extension .xml qui pourrait être utilisée... Cela dit, il est utile de pouvoir reconnaître facilement une Facelet parmi d'autres fichiers du même type, pour la démarquer du reste. Voilà pourquoi ces trois extensions ont *de facto* été retenues par les développeurs. Ne vous inquiétez pas, nous découvrirons très bientôt comment le framework est capable de savoir quelles extensions de fichiers lui sont destinées.

Ainsi, voici la structure à vide d'une Facelet :

Code : HTML- exemple.jsf

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

Vous retrouvez bien le *doctype* HTML, suivi de la balise `<html>` conforme au standard XHTML. Bref, en apparence ce n'est qu'une page web toute simple !

Bibliothèques et balises

Une Facelet est donc constituée de balises, dont la syntaxe est très similaire à celles des balises JSTL utilisées dans une page JSP.

Avant de poursuivre sur la forme des balises JSF, intéressons-nous aux bibliothèques. Tout comme il existe des directives permettant d'importer des bibliothèques de balises JSTL dans une page JSP, il existe un moyen pour inclure les bibliothèques de balises JSF dans une Facelet. Toutefois, il vous faut oublier le concept même de la directive JSP : il s'agissait là littéralement d'un ordre donné au conteneur, lui précisant comment il devait gérer une page JSP et générer sa servlet Java associée. Voilà pourquoi il était non seulement possible via une directive d'inclure des bibliothèques, mais également d'importer des classes Java, d'activer ou non les sessions HTTP, les expressions EL, etc.

Dans une Facelet, une bibliothèque de balises est incluse via l'ajout d'un attribut `xmlns` à la balise `<html>` qui ouvre le corps de la page. Il s'agit là d'un *namespace* XML. Je vous invite à vous renseigner sur cette notion si vous souhaitez comprendre comment cela fonctionne en détail. En ce qui nous concerne, nous allons simplement retenir qu'un tel attribut permet de déclarer une bibliothèque dans une Facelet ! Le framework JSF intègre nativement trois bibliothèques standard : **HTML**, **Core** et **UI**. Voici comment les inclure dans une Facelet :

Code : HTML- exemple.jsf

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:ui="http://java.sun.com/jsf/facelets">
...
</html>
```

Vous reconnaisez ici le système de liens utilisé également par la JSTL. De même, vous retrouvez le système de préfixe identifiant une bibliothèque : traditionnellement, h: désigne la bibliothèque HTML, ui: la bibliothèque de *templating* et de composition, et f: la bibliothèque Core. Au passage, ne confondez pas cette dernière avec la bibliothèque Core de la JSTL, nous parlons bien ici de balises JSF !

Pour terminer et revenir sur les balises JSF à proprement parler, sur la forme elles ressemblent comme deux gouttes d'eau aux balises JSTL. Par exemple, voici une balise issue de la bibliothèque HTML :

Code : HTML- Exemple de balise JSF

```
<h:outputLabel for="confirmation">Confirmation du mot de passe <span
class="requis">*</span></h:outputLabel>
```

Vous pouvez d'ores et déjà remarquer les similitudes avec les balises JSTL : il est possible d'y préciser des attributs, une balise possède un corps, il est possible d'y inclure d'autres balises, etc.

Dans nos exemples à venir, nous allons utiliser de nombreuses balises qui vous sont encore inconnues. Si j'ai pris le temps de vous décrire les balises JSP et JSTL les plus couramment utilisées lors de nos premiers pas avec Java EE, je ne vais cette fois pas rédiger d'inventaire des balises JSF, et ce pour trois raisons :

1. vous êtes à l'aise avec le concept de balises, vous savez comment elles se construisent ;
2. vous êtes à l'aise avec la recherche d'informations, vous êtes capables de trouver et parcourir les ressources et documentations par vous-mêmes ;
3. les bibliothèques standard de JSF contiennent beaucoup de balises, certaines dédiées à des tâches bien différentes de ce que vous connaissez et avez manipulé jusqu'à présent, et en faire l'inventaire détaillé et expliqué par l'exemple serait chronophage et en fin de compte peu efficace.

Pour vous faciliter la tâche, je vous communique quelques ressources extrêmement utiles :

[la documentation officielle des balises JSF](#), par Oracle ; [une documentation similaire](#), qui date un peu mais qui est présentée de manière un peu plus conviviale que le format Javadoc, par JSFToolbox ; [la page JSF de Stackoverflow](#), contenant des généralités, des exemples basiques et une liste de liens fournie et à jour sur le sujet.

Gardez ces liens accessibles dans les favoris de votre navigateur, vous allez en avoir besoin pour découvrir toutes les balises proposées par JSF !

Expressions EL

Une Facelet peut, tout comme une page JSP, contenir des expressions EL. Toutefois, celles-ci sont un peu différentes de celles que nous avons utilisées jusqu'à présent. Voyez dans cet exemple la forme d'une expression EL avec JSF :

Code : HTML - Exemple d'expression EL avec JSF

```
#{inscrireBean.utilisateur.motDePasse}
```

Vous allez me dire, à part le symbole # qui remplace le symbole \$ auparavant utilisé, a priori le reste ne change pas : les accolades, l'accès aux beans, à ses propriétés, l'opérateur . en guise de séparateur... Eh bien en réalité, c'est un peu plus compliqué que ça. Pour mieux comprendre, faisons un rapide historique de la technologie EL :

- en juin 2002, la première version de la JSTL paraît et introduit le concept des expressions EL pour la toute première fois. Celles-ci étaient alors construites pour appeler les méthodes *getter* de JavaBeans, uniquement via la syntaxe \${...} et uniquement depuis les balises JSTL ;
- en novembre 2003, la seconde mouture de la technologie JSP fait son apparition et les expressions EL deviennent partie intégrante du standard J2EE 1.4. La JSTL en version 1.1 ne contient alors plus la technologie EL, et la syntaxe \${...} fonctionne désormais en dehors des balises JSTL, dans le corps des pages JSP ;
- en mars 2004, la première version de JSF est publiée, et introduit le concept des expressions EL dites "différées". Il s'agissait alors d'expressions sous la forme # {...} et ne fonctionnait que dans des balises JSF. La principale différence avec les expressions de la forme \${...} décrites par le standard JSP est qu'elles permettent non seulement d'appeler les méthodes *getter* des beans, mais également leurs méthodes *setter* ;
- en mai 2005, les deux technologies EL coexistantes sont combinées en une seule spécification : les expressions EL sont alors dites "unifiées", et font partie intégrante du standard Java EE 5. La syntaxe # {...} devient de fait utilisable également depuis une page JSP, mais y est toujours limitée à l'accès aux méthodes *getter*, seul l'usage depuis JSF permet d'appeler les méthodes *setter* ;
- en décembre 2009, une évolution des expressions EL est introduite avec le standard Java EE 6. Celles-ci permettent désormais, depuis la syntaxe # {...}, d'appeler n'importe quelle méthode d'un bean, et non plus seulement ses *getters/setters*. C'est par ailleurs à cette occasion que les Facelets sont devenues partie intégrante du standard Java EE 6.

De cette longue épopée, vous devez retenir deux choses importantes :

1. il est possible d'utiliser la syntaxe # {...} depuis vos pages JSP ! Je ne vous l'ai pas présentée plus tôt pour ne pas vous embrouiller, et surtout pour ne pas vous voir appeler des méthodes Java dans tous les sens sans comprendre ce que MVC implique et impose, ni comprendre comment doit être construit un bean...

2. nous allons dorénavant utiliser la syntaxe # { . . . } avec JSF, car contrairement à ce que nous faisions avec la syntaxe \$ { . . . } depuis nos pages JSP, nous allons avoir besoin d'initialiser des valeurs et non plus seulement d'accéder en lecture à des valeurs.

Vous voilà au point sur les aspects superficiels de la technologie Facelet. Il est grand temps maintenant de passer derrière les rideaux, et de découvrir comment tout cela fonctionne dans les coulisses...

Comment ça marche ?

Avant tout, et parce que vous en aurez forcément un jour besoin si vous travaillez avec JSF, voici l'outil le plus utile pour comprendre les rouages du système : [la documentation officielle du framework JSF](#) dans sa version actuelle (2.1).

Vous savez déjà comment fonctionne une page JSP, mais un petit rappel ne fait jamais de mal. La technologie JSP fournit un langage de *templating*, permettant de créer des pages qui sont - par un procédé que vous connaissez - traduites en servlets Java, puis compilées. Pour faire court, le corps d'une page JSP devient l'équivalent de la méthode `service()`, la méthode mère des méthodes `doGet()` et `doPost()`. Les balises JSP et JSTL qui y sont utilisées sont directement transformées en code Java et intégrées dans la servlet générée, vous pouvez d'ailleurs vous en rendre compte par vous-mêmes en allant regarder le code des servlets auto-générées par Tomcat dans nos précédents exemples.

Dans une Facelet par contre, qui comme nous venons de le voir n'est qu'un fichier XML, les balises JSF ne sont que des appels à des composants JSF qui sont entièrement autonomes. Autrement formulé, lorsqu'un composant JSF est appelé, il génère son propre rendu dans son état courant. Le cycle de vie des composants JSF n'a pas conséquent **aucune relation avec le cycle de vie d'une page JSP** et de sa servlet auto-générée. Une Facelet est donc une page constituée d'une suite d'appels à des composants JSF (réalisés par l'intermédiaire de balises), et ceux-ci forment ce que l'on appelle **un arbre de composants**, ou *component tree* en anglais.

Ainsi, ne vous laissez pas tromper par les apparences : même si JSF vous permet de travailler avec des balises JSF qui ressemblent comme deux gouttes d'eau à des balises JSTL, retenez bien que JSF ne fonctionne pas comme fonctionnaient nos exemples MVC basés sur des servlets et des JSP. Ces similitudes ont par contre un avantage certain : puisque vous connaissez déjà MVC avec les JSP, vous pouvez attaquer le développement avec JSF très rapidement !

En prenant un peu de recul, JSF s'apparente dans son fonctionnement à Swing ou AWT : c'est un *framework* qui fournit une collection de composants standards et réutilisables, permettant la création d'interfaces utilisateur (web, en l'occurrence). À la différence des JSP, les Facelets JSF conservent un état (on dit alors que la vue est *stateful*) : ce sont les composants autonomes appelés par l'intermédiaire des balises contenues dans les Facelets qui permettent de maintenir cet état. De la même manière que Swing et AWT, les composants JSF suivent le *pattern de l'objet composite* pour gérer un arbre de composants : en clair, cela signifie à la fois qu'un objet conteneur contient un composant, et qu'un objet conteneur est lui-même un composant. La vue lie ces composants graphiques à la page XHTML, et permet ainsi au développeur de directement lier des champs HTML d'interaction utilisateur (saisie de données, listes, etc.) à des propriétés de beans, et des boutons à leurs méthodes d'action.

Un processus en 6 étapes

Arrêtons les comparaisons avec d'autres solutions, et étudions concrètement le fonctionnement du *framework*. Avec JSF, le traitement d'une requête entrant sur le serveur est découpé en six étapes, que nous allons parcourir sommairement :

1. La restauration de la vue

La requête entrante est redirigée vers l'unique servlet jouant le rôle de super-contrôleur, la `FacesServlet`. Celle-ci examine son contenu, en extrait le nom de la page ciblée et détermine s'il existe déjà une vue associée à cette page (eh oui, rappelez-vous bien qu'avec JSF la vue conserve un état). Voilà pourquoi cette étape s'intitule "restauration de la vue" : il s'agit en réalité de restaurer les éventuels composants déjà chargés si l'utilisateur a déjà accédé à la page par le passé.

La `FacesServlet` va donc chercher les composants utilisés par la vue courante. Si la vue n'existe pas déjà, elle va la créer. Si elle existe déjà, elle la réutilise. La vue contient tous les composants de l'interface utilisateur intervenant dans la page. La vue (c'est-à-dire l'ensemble des composants qui y interviennent, et donc son état) est sauvegardée dans l'objet `FacesContext`.

2. L'application des valeurs contenues dans la requête

Arrivés à cette étape, les composants de la vue courante ont tout juste été récupérés ou créés depuis l'objet `FacesContext`. Chacun d'eux va maintenant récupérer la valeur qui lui est assignée depuis les paramètres de la requête, ou éventuellement

depuis des cookies ou headers.

Ces valeurs vont alors être converties. Ainsi, si un champ est lié à une propriété de type `Integer`, alors son contenu va être converti en `Integer`. Si cette conversion échoue, un message d'erreur va être placé dans le `FacesContext`, et sera utilisé lors du futur rendu de la réponse.

À noter qu'à cette étape peut intervenir la "prise en charge immédiate des événements" : cela veut dire que si un composant est marqué comme tel, sa valeur va directement être convertie puis validée dans la foulée. Si aucun composant n'arbore cette propriété, alors les valeurs de tous les composants sont d'abord converties, puis intervient ensuite l'étape de validation sur l'ensemble des valeurs.

3. La validation des données

Les valeurs tout juste converties vont ensuite être validées, en suivant les règles de validation définies par le développeur. Si la validation d'une valeur échoue, un message d'erreur est ajouté au `FacesContext`, et le composant concerné est marqué comme "invalid" par JSF. La prochaine étape est alors directement le rendu de la réponse, il n'y aura aucune autre étape intermédiaire.

Si les valeurs sont correctes vis-à-vis des règles de validation en place, alors la prochaine étape est la mise à jour des valeurs du modèle.

4. La mise à jour des valeurs du modèle

Les composants peuvent être directement liés, par l'intermédiaire des balises présentes dans la vue, à des propriétés de beans. Ces beans sont qualifiés de *managed-beans* ou *backing-beans*, car ils sont gérés par JSF et la vue s'appuie sur eux. Si de tels liens existent, alors les propriétés de ces beans sont mises à jour avec les nouvelles valeurs des composants correspondants, fraîchement validées. Puisque la validation a eu lieu en premier lieu, le développeur est certain que les données enregistrées dans le modèle sont valides, au sens format du champ du formulaire. Il n'est pas exclu que les données ne soient pas valides d'un point de vue de ce qu'attend le code métier de l'application, mais c'est tout à fait normal, puisque cette étape est la suivante dans le processus...

5. L'appel aux actions, le code métier de l'application

Les actions associées à la soumission du formulaire sont alors appelées par JSF. Il s'agit enfin de l'entrée en jeu du code métier : maintenant que les données ont été converties, validées et enregistrées dans le modèle, elles peuvent être utilisées par l'application.

La fin de cette étape se concrétise par la redirection vers la vue correspondante, qui peut dépendre ou non du résultat produit par le code métier. Il s'agit donc de définir la navigation au sein des vues existantes, ce qui est réalisé directement depuis le bouton de validation dans la page, ou depuis un fichier de configuration XML externe nommé `faces-config.xml`.

6. Le rendu de la réponse

La dernière et ultime étape est le rendu de la réponse. La vue définie dans la navigation est finalement affichée à l'utilisateur : tous les composants qui la composent effectuent alors leur propre rendu, dans leur état courant. La page HTML ainsi générée est finalement envoyée au client, mais ça, vous vous en doutiez !

Voilà comment se déroule le traitement d'une requête avec JSF. Comme vous pouvez le voir, ça change du mode de traitement linéaire que nous avions adopté dans nos exemples MVC faits maison, notamment au niveau de la possibilité de prise en charge immédiate d'un événement qui permet le court-circuitage du processus global pour un composant en particulier ! Si tout ceci est encore très flou dans votre tête, c'est normal : beaucoup de choses vous semblent encore bien abstraites. Ne vous découragez surtout pas, car comprendre le fonctionnement de JSF est l'effort le plus intense qu'il vous faudra fournir. Dès lors que vous aurez assimilé comment se goupille toute cette mécanique, vous aurez fait plus de la moitié du chemin vers l'adoption de JSF !



Pour vous aider à bien comprendre, je vous propose de découvrir ce processus dans un exemple pratique très simple. Mais avant cela, je vous propose de vous détendre un peu en découvrant une petite astuce sous Eclipse, permettant de préparer rapidement votre espace de travail au développement de Facelets, les fameuses pages que nous allons créer à la place de nos pages JSP.

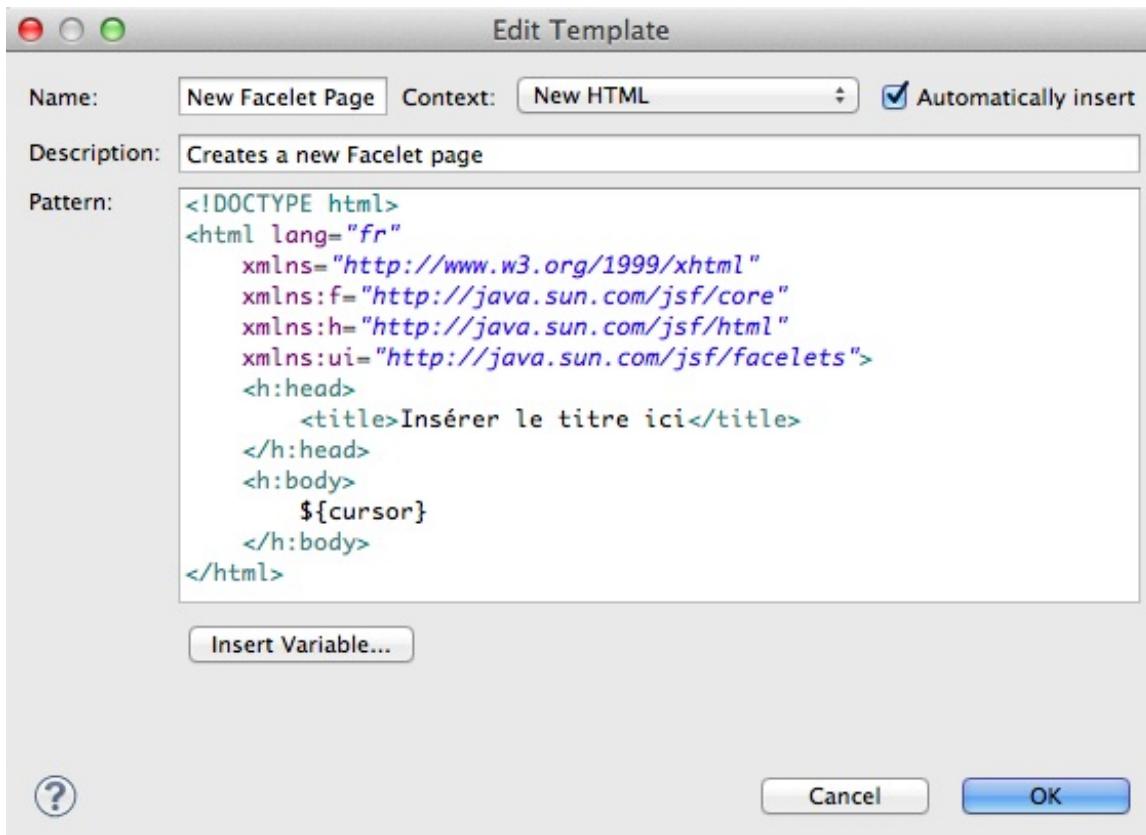
Créer un template de Facelet par défaut avec Eclipse

Avec Eclipse, il est possible de définir quel sera le contenu généré par défaut lors de la création d'un nouveau fichier. Pour préparer facilement nos vues JSF, il nous suffit donc de créer un nouveau type de fichier nommé "Facelet" et de personnaliser son contenu par défaut. Pour ce faire, rendez-vous dans les préférences d'Eclipse, puis suivez Web > HTML Files > Editor > Templates et cliquez enfin sur New.

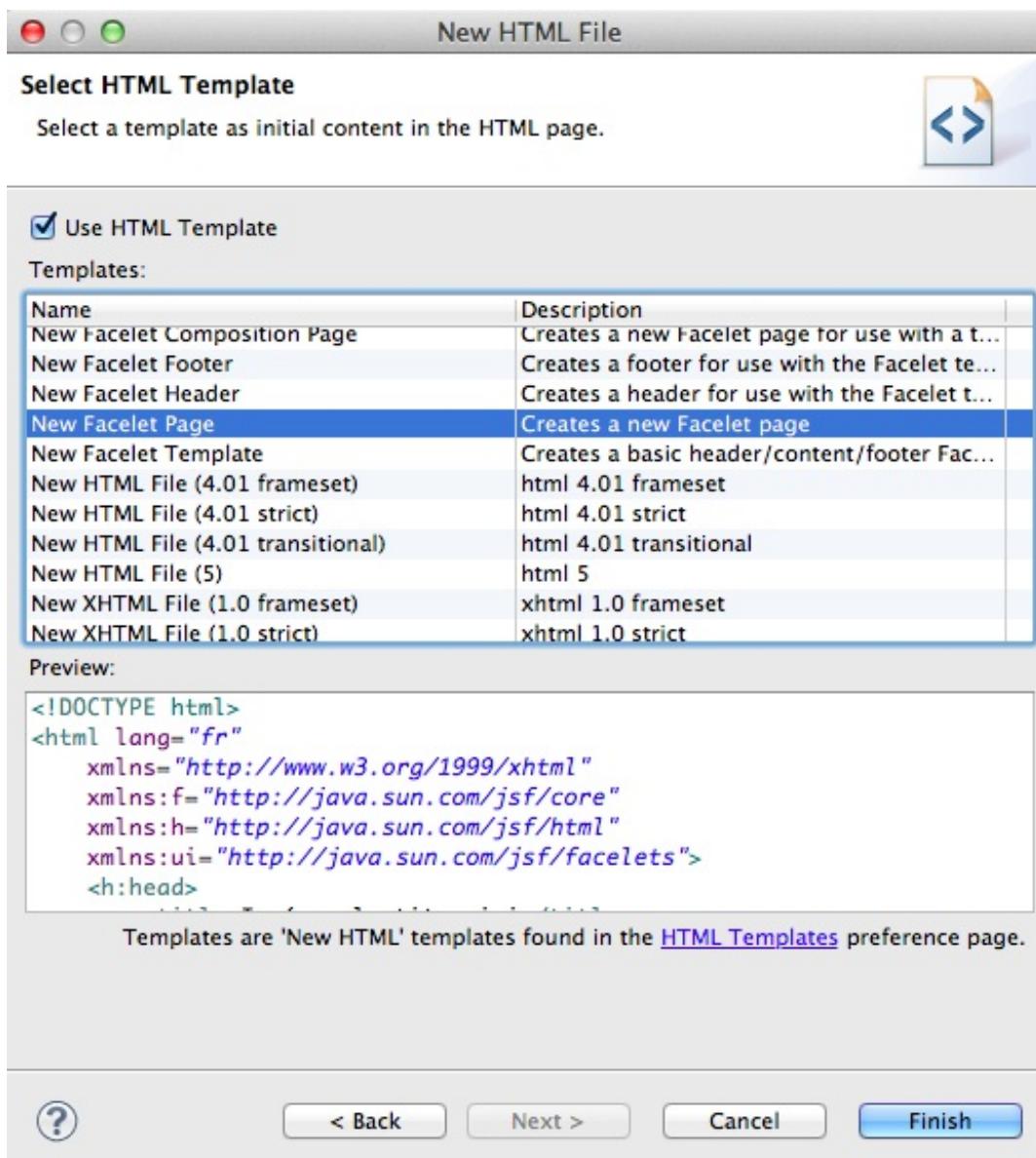
Entrez alors comme nom "New Facelet Page", puis sélectionnez le contexte "New HTML", entrez comme description "Creates a new Facelet page", puis copiez le code suivant dans le champ pattern et validez enfin en cliquant sur OK (voir la figure suivante).

Code : HTML - Modèle de nouvelle Facelet

```
<!DOCTYPE html>
<html lang="fr"
      xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:ui="http://java.sun.com/jsf/facelets">
    <h:head>
        <title>Insérer le titre ici</title>
    </h:head>
    <h:body>
        ${cursor}
    </h:body>
</html>
```



Une fois cette configuration en place, vous pourrez simplement créer une nouvelle Facelet prête à être codée ! Il vous suffira de faire un clic droit dans un projet, puis de suivre New > HTML File. Puis de donner un nom à votre fichier et de cliquer sur Next dans la fenêtre qui s'affiche alors, avant de choisir votre template fraîchement créé dans la liste qui s'affiche enfin et de valider en cliquant sur Finish, comme indiqué à la figure suivante.



Le nouveau fichier que vous venez de créer contient alors automatiquement le code de base que vous avez défini plus tôt dans les options d'Eclipse.

Premier projet De quoi avons-nous besoin ?

Puisque nous travaillons avec GlassFish, nous n'avons en apparence besoin de rien : tout y est déjà inclus ! En réalité, c'est encore une fois le même refrain : JSF n'est qu'une spécification, et pour utiliser JSF il faut donc disposer d'une implémentation. Souvenez-vous de JPA, c'était exactement pareil : l'implémentation de référence de JPA utilisée par défaut par GlassFish était EclipseLink, mais il existait d'autres implémentations très utilisées comme Hibernate notamment. En ce qui concerne JSF, il existe deux principales implémentations :

- [Oracle Mojarra](#), l'implémentation de référence, utilisée par défaut par GlassFish ;
- [Apache MyFaces](#), l'implémentation éditée par Apache.



Quelle implémentation choisir ?

Les différences entre ces deux solutions sont minimes aux premiers abords. Seule une utilisation très poussée d'une solution ou de l'autre vous fera prendre conscience des écarts existant entre ces deux implémentations, et de l'intérêt de préférer l'une à l'autre. Ainsi, il n'y a pas de réponse empirique à la question : selon le contexte de votre projet, vous serez **peut-être** amenés à changer d'une implémentation vers l'autre, en raison d'un comportement qui pose problème chez l'une mais pas chez l'autre.

Qui plus est, puisque ces deux implémentations respectent la spécification JSF, elles sont interchangeables très simplement.

L'éventuel besoin de changer d'implémentation en cours de route dans un projet n'est donc pas un réel problème.

Bref, en ce qui nous concerne, nous n'en sommes qu'aux balbutiements et allons partir par défaut sur l'implémentation Mojarra, puisque c'est celle que GlassFish embarque nativement.



Et si nous n'utilisions pas GlassFish ?

Si le serveur utilisé est un serveur d'applications Java EE au sens strict du terme, alors une implémentation JSF doit être fournie par défaut. Si par contre c'est un serveur léger comme Tomcat qui est utilisé, alors il est nécessaire d'ajouter au projet les jar de l'implémentation JSF pour pouvoir l'utiliser dans votre application. Ces archives jar sont bien entendu disponibles au téléchargement sur les sites respectifs des deux solutions.



Et si nous souhaitions utiliser MyFaces sur un serveur GlassFish ?

Si nous voulions changer d'implémentation JSF et utiliser MyFaces au lieu de Mojarra, il nous suffirait alors d'importer les archives jar de l'implémentation dans notre projet, de la même manière que nous devons les importer sur un serveur léger.



Au final, retenez bien que GlassFish est livré par défaut avec une implémentation de JSF, et donc que nous pouvons travailler avec JSF sans aucun ajout.

Création du projet

Nous pouvons maintenant attaquer la création de notre premier exemple : nous allons très modestement créer une page qui demande à l'utilisateur de saisir son nom dans un champ de formulaire, et une seconde page qui se chargera d'afficher le nom saisi à l'utilisateur. Rien de transcendant je vous l'accorde, mais c'est déjà assez pour que vous puissiez découvrir en douceur le fonctionnement de JSF.

Avant tout, nous devons mettre en place un projet web sous Eclipse. La démarche est la même que pour nos précédents travaux :

- créez un projet web dynamique ;
- nommez-le pour cet exemple **test_jsf** ;
- validez, et le projet est prêt !

Création du bean

Pour commencer, nous allons créer un simple bean pour stocker le nom saisi par l'utilisateur. Je vous donne le code, et vous explique les quelques nouveautés ensuite :

Code : Java - com.sdzee.exemple.BonjourBean

```
package com.sdzee.exemple;

import java.io.Serializable;

import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;

@ManagedBean
@RequestScoped
public class BonjourBean implements Serializable {

    private static final long serialVersionUID = 1L;

    private String nom;

    public String getNom() {
        return nom;
    }
}
```

```

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

```

Vous pouvez observer deux choses qui diffèrent des simples beans que nous utilisions dans nos exemples MVC traditionnels :

- le bean implémente l'interface **Serializable**. En réalité, je vous avais déjà prévenu que nos beans pouvaient tirer parti de cette interface lorsque nous avons découvert le JavaBean pour la première fois, mais je ne vous avais alors pas expliqué à quoi cela pouvait bien servir. En réalité c'est très simple : en rendant un bean sérialisable, vous lui donnez la capacité de survivre à un redémarrage du serveur. Cela ne va pas plus loin que cela, c'est un détail qui n'a aucune importance dans notre exemple, et j'ai simplement fait intervenir cette interface pour vous expliquer son rôle. Au passage, votre IDE Eclipse vous fera remarquer la nécessité de préciser un attribut **serialVersionUID**, qu'il peut générer automatiquement pour vous.
- le bean contient deux annotations spécifiques à JSF :
 - **@ManagedBean** : permet de préciser au serveur que ce bean est dorénavant géré par JSF. Cela signifie simplement que JSF va utiliser ce bean en tant que modèle associé à une ou plusieurs vues. Par défaut, le nom du bean correspond au nom de la classe, la majuscule en moins : en l'occurrence le nom de notre bean est donc **bonjourBean**. Si nous voulions désigner ce bean par un autre nom, par exemple **direBonjour**, alors il nous faudrait annoter le bean en précisant le nom souhaité, via `@ManagedBean(name="direBonjour")` ;
 - **@RequestScoped** : permet de préciser au serveur que ce bean a pour portée la requête. Il s'agit en l'occurrence de la portée utilisée par défaut en cas d'absence d'annotation. Ainsi, si vous omettez de l'écrire, le bean sera de toute manière placé dans la portée requête. C'est toutefois une bonne pratique de toujours écrire cette annotation, afin de clarifier le code. Il existe autant d'annotations que de portées disponibles dans JSF : **@NoneScoped**, **@RequestScoped**, **@ViewScoped**, **@SessionScoped**, **@ApplicationScoped**, et **@CustomScope**. Ne vous inquiétez pas, nous y reviendrons en détail très prochainement.



Pour information, avec JSF 1.x ces annotations n'existaient pas, il fallait déclarer chaque bean dans un fichier de configuration externe nommé **faces-config.xml**. Grâce aux annotations, ce n'est désormais plus nécessaire avec JSF 2.x !

Vous savez maintenant ce qu'est le fameux *managed-bean* ou *backing-bean* dont je vous avais parlé un peu plus tôt : un simple bean annoté pour le déclarer comme tel auprès de JSF. Un pas de plus vers la compréhension de JSF... 😊

Création des facelets

La seconde étape consiste à créer les vues de notre petite application. Nous allons donc créer deux Facelets, qui pour rappel ne sont rien d'autre que des pages XHTML et contenant des balises propres à JSF, chargées respectivement d'afficher un champ de formulaire à l'utilisateur, et de lui afficher les données saisies.

Nous allons nommer la page contenant le formulaire **bonjour.xhtml**, et la page chargée de l'affichage **bienvenue.xhtml**. Toutes deux doivent être placées directement à la racine de votre application, symbolisée par le dossier **/WebContent** dans Eclipse :

Code : HTML - /bonjour.xhtml

```

<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
    <head>
        <title>Premier exemple JSF 2.0</title>
    </head>
    <body>
        <h1>Premier exemple JSF 2.0 - bonjour.xhtml</h1>
        <h:form>
            <h:inputText value="#{bonjourBean.nom}" />
            <h:commandButton value="Souhaiter la bienvenue"
action="bienvenue" />
        </h:form>
    </body>
</html>

```

Cette page génère :

- un champ de texte JSF, et le lie avec la propriété **nom** du bean **bonjourBean** (notre *managed-bean*, géré par JSF) ;
- un bouton de formulaire, chargé d'afficher la page **bienvenue.xhtml** lorsqu'il est cliqué.

Observons la constitution de cette page. En premier lieu, vous retrouvez logiquement l'en-tête html particulier dont je vous ai parlé lorsque nous avons abordé la constitution d'une Facelet, qui contient le type de la page ainsi que les éventuelles déclarations de bibliothèques de balises JSF. En l'occurrence, nous avons ici simplement déclaré la bibliothèque HTML à la ligne 3, puisque nous n'utilisons dans la page que des balises de la bibliothèque HTML.

Vous observez ensuite des balises JSF, préfixées par h: comme déclaré dans l'en-tête html. Étudions-les dans leur ordre d'apparition :

Le header

La section header est créée par la balise **<h:head>**. Il s'agit d'un composant JSF qui permet notamment d'inclure des ressources JS ou CSS dans le contenu généré entre les balises HTML **<head>** et **</head>**, de manière automatisée depuis votre code Java.

Dans notre exemple, nous n'avons rien de particulier à y inclure, nous nous contentons d'y définir un titre pour notre page via la balise HTML **<title>**. Vous pouvez également y déclarer un charset via la balise HTML **<meta>**, etc., comme vous le faisiez dans vos pages JSP.

Le formulaire

Le formulaire est créé par la balise **<h:form>**. Il s'agit du composant JSF permettant de générer un formulaire, contenant d'éventuels champs de saisies. La méthode HTTP utilisée pour l'envoi des données est toujours **POST**. Si vous souhaitez utiliser la méthode **GET**, alors le plus simple est de ne pas utiliser le composant JSF et d'écrire directement votre formulaire en HTML brut en y spécifiant l'attribut **<form... method="get">**.

Les données sont par défaut renvoyées à la page contenant le formulaire.

Le champ de saisie

Le champ de type texte est créé par la balise **<h:inputText>**. Son attribut **value** permet de définir deux choses :

- la valeur contenue dans cet attribut sera par défaut affichée dans le champ texte, il s'agit ici du même principe que pour un champ de texte HTML classique **<input type="text" value="..." />** ;
- la valeur contenue dans cet attribut sera utilisée pour initialiser la propriété **nom** du bean **BonjourBean**, par l'intermédiaire de l'expression EL `# {bonjourBean.nom}`. En l'occurrence, JSF va évaluer l'expression lorsque le formulaire sera validé, c'est-à-dire lorsque le bouton sera cliqué, et va alors chercher l'objet nommé **bonjourBean**, puis utiliser sa méthode **setter setNom()** pour enregistrer dans la propriété **nom** la valeur contenue dans le champ texte.

Le bouton de validation

Le bouton d'envoi des données du formulaire est créé par la balise **<h:commandButton>** :

- son attribut **value** contient la valeur affichée à l'utilisateur en guise de texte du bouton, tout comme un bouton HTML classique ;
- son attribut **action** permet quant à lui de définir où rediriger l'utilisateur : en l'occurrence, nous allons le rediriger vers la page qui affiche le texte saisi, c'est-à-dire **bienvenue.xhtml**. Il nous suffit pour cela d'écrire **bienvenue** dans le champ **action** de la balise, et le composant se chargera automatiquement derrière les rideaux d'appeler la page nommée **bienvenue.xhtml**.

Sous JSF 1.x, il était nécessaire de déclarer une *navigation-rule* dans le fichier **faces-config.xml**, afin de définir quelle page serait affichée une fois le bouton cliqué. Avec JSF 2.x, vous constatez qu'il est dorénavant possible de directement placer le nom de la page dans l'attribut **action** du bouton. Pour une navigation simple, c'est très pratique et plus que suffisant ! Toutefois, sachez que pour mettre en place une navigation un peu plus complexe, il nous faudra toujours utiliser une section *navigation-rule* dans le **faces-config.xml**.



Voilà ensuite le code de la page de bienvenue, extrêmement simple :

Code : HTML - /bienvenue.xhtml

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
  <h:head>
    <title>Premier exemple JSF 2.0</title>
  </h:head>
  <h:body>
    <h1>Premier exemple JSF 2.0 - bienvenue.xhtml</h1>
    <p>Bienvenue #{bonjourBean.nom} !</p>
  </h:body>
</html>
```

Lors de l'affichage de la page, JSF va évaluer l'expression EL `#{bonjourBean.nom}` située à la ligne 9, et chercher l'objet intitulé **bonjourBean** pour afficher sa propriété **nom**, récupérée automatiquement via la méthode `getter getNom()`.

Configuration de l'application

Afin de mettre ce petit monde en musique, il nous faut pour finir écrire un peu de configuration dans le fichier **web.xml** :

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

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
  <!-- Changer cette valeur à "Production" lors du déploiement final
  de l'application -->
  <context-param>
    <param-name>javax.faces.PROJECT_STAGE</param-name>
    <param-value>Development</param-value>
  </context-param>

  <!-- Déclaration du contrôleur central de JSF : la FacesServlet --
  >
  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <!-- Mapping : association des requêtes dont le fichier porte
  l'extension .xhtml à la FacesServlet -->
  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>*.xhtml</url-pattern>
  </servlet-mapping>
</web-app>
```

Aux lignes 10 à 14, nous déclarons tout simplement la servlet mère de JSF, celle qui joue le rôle du *Front Controller* : la **FacesServlet**. C'est exactement le même principe que lorsque nous définissions nos propres servlets à la main auparavant. Il suffit de préciser un nom, en l'occurrence je l'ai nommée "**Faces Servlet**", et sa localisation dans l'application, en l'occurrence `javax.faces.webapp.FacesServlet`.

Aux lignes 17 à 20, nous procédons ensuite au mapping d'un *pattern* d'URL sur cette seule et unique **FacesServlet**. L'objectif est de rediriger **toutes les requêtes entrantes** vers elle. Dans notre exemple, nous nous contentons d'associer les vues portant l'extension `.xhtml` à la FacesServlet, puisque toutes nos vues sont ainsi constituées.

À ce sujet, sachez que tous les développeurs n'utilisent pas l'extension `.xhtml` pour leurs vues, et il est courant dans les projets

JSF existants de rencontrer quatre types d'URL différentes : /faces/*, *.jsf, *.xhtml et *.faces. Si vous devez un jour manipuler des vues portant une de ces extensions, il vous faudra simplement ajouter des mappings dans votre web.xml :

Code : XML - Mappings pour les différents types d'URL

```
<!-- Mapping des différents patterns d'URL devant être associés à
la FacesServlet -->
<servlet-mapping>
  <servlet-name>Faces Servlet</servlet-name>
  <url-pattern>/faces/*</url-pattern>
</servlet-mapping>
<servlet-mapping>
  <servlet-name>Faces Servlet</servlet-name>
  <url-pattern>*.jsf</url-pattern>
</servlet-mapping>
<servlet-mapping>
  <servlet-name>Faces Servlet</servlet-name>
  <url-pattern>*.faces</url-pattern>
</servlet-mapping>
<servlet-mapping>
  <servlet-name>Faces Servlet</servlet-name>
  <url-pattern>*.xhtml</url-pattern>
</servlet-mapping>
```

Ainsi avec une telle configuration, dans notre projet les quatre URLs suivantes pointeront toutes vers la même page **/bonjour.xhtml** :

- http://localhost:8088/test_jsf/bonjour.jsf
- http://localhost:8088/test_jsf/bonjour.faces
- http://localhost:8088/test_jsf/bonjour.xhtml
- http://localhost:8088/test_jsf/faces/bonjour.jsf

Je me répète, mais en ce qui nous concerne le seul mapping des vues portant l'extension **.xhtml** est suffisant, nous n'avons pas besoin de mettre en place tous ces autres mappings.

Enfin, vous remarquerez la déclaration d'un paramètre particulier aux lignes 4 à 7. Il s'agit d'une fonctionnalité utile proposée par JSF. Lors du développement d'une application, il est recommandé d'initialiser le paramètre nommé `javax.faces.PROJECT_STAGE` avec la valeur "**Development**". Ceci va rendre disponibles de nombreuses informations de debugging et les afficher directement au sein de vos pages en cas de problème, permettant ainsi de tracer les erreurs rapidement. Lors du déploiement final, une fois l'application achevée, il suffit alors de changer la valeur du paramètre à "**Production**", et toutes ces informations non destinées au public ne seront alors plus affichées.



Le fichier web.xml que je vous donne ici en exemple est donc générique, il ne contient rien de spécifique à ce projet en particulier et se contente d'établir les propriétés nécessaires au bon fonctionnement de JSF. En d'autres termes, vous pourrez réutiliser ce fichier tel quel pour tous vos projets JSF !

Tests & observations

Notre projet est maintenant prêt pour utilisation : seuls ces deux Facelets et ce bean suffisent ! Vérifions pour commencer le bon fonctionnement de l'application en nous rendant sur l'URL http://localhost:8088/test_jsf/bonjour.xhtml depuis notre navigateur. Vous devez observer un champ de formulaire et un bouton de validation, comme indiqué sur la figure suivante.



Premier exemple JSF 2.0 - bonjour.xhtml

Coyote | Souhaiter la bienvenue

Après un clic sur le bouton, vous devez alors être redirigés vers la page de bienvenue affichant ce que vous avez tapé dans le champ texte (voir la figure suivante).



Premier exemple JSF 2.0 - bienvenue.xhtml

Bienvenue Coyote !

Essayons maintenant de changer la cible précisée dans l'attribut **action** du bouton par une page qui n'existe pas dans notre application. Modifiez par exemple la ligne 12 de la page **bonjour.xhtml** par :

Code : HTML - Modification de l'attribut action

```
<h:commandButton value="Souhaiter la bienvenue" action="connexion"
/>
```

Rendez-vous alors à nouveau sur la page http://localhost:8088/test_jsf/bonjour.xhtml, cliquez sur le bouton et observez l'erreur affichée sur la figure suivante.



Premier exemple JSF 2.0 - bonjour.xhtml

Coyote | Souhaiter la bienvenue

- Unable to find matching navigation case with from-view-id '/bonjour.xhtml' for action 'connexion' with outcome 'connexion'

Vous voilà face au type d'informations dont je vous ai parlé lorsque je vous ai expliqué l'intérêt du paramètre **PROJECT_STAGE**, que nous avons mis en place dans notre **web.xml**. Le contrôleur JSF - la FacesServlet - est incapable de trouver une Facelet nommée **connexion.xhtml**, et JSF vous affiche donc automatiquement l'erreur, directement au sein de votre page ! Pratique pour identifier ce qui pose problème au premier coup d'œil, n'est-ce pas ?

Faisons maintenant le même essai, mais cette fois en passant le paramètre `javax.faces.PROJECT_STAGE` à "Production". Effectuez la modification dans le fichier **web.xml**, puis rendez-vous une nouvelle fois sur la page depuis votre navigateur. Dorénavant lorsque vous cliquez sur le bouton, aucune erreur ne s'affiche ! Et c'est normal, puisque le mode de production est

destiné à une utilisation publique de l'application.



Vous comprenez maintenant mieux l'intérêt de ce paramètre : il permet de définir le mode de fonctionnement de l'application, et donc de définir si les messages de debug seront affichés ou non sur les pages affichées par l'utilisateur.

Nous allons maintenant annuler nos précédentes modifications : remettons à "Development" le paramètre dans le fichier web.xml, et remettons "bienvenue" dans le champ **action** de notre formulaire. Nous allons ensuite modifier l'expression EL contenue dans notre champ texte, en y précisant un bean qui n'existe pas dans notre application. Modifiez par exemple la ligne 11 de la page **bonjour.xhtml** par :

Code : HTML - Modification de l'expression EL

```
<h:inputText value="#{inscriptionBean.nom}" />
```

Rendez-vous alors à nouveau sur la page http://localhost:8088/test_jsf/bonjour.xhtml, cliquez sur le bouton et observez l'erreur affichée sur la figure suivante.

The screenshot shows a browser window with the URL localhost:8088/test_jsf/bonjour.xhtml. The main content area displays the error message: **/bonjour.xhtml @11,55 value="#{inscriptionBean.nom}": Target Unreachable, identifier 'inscriptionBean' resolved to null**. Below the error message, there are three expandable sections: **+ Stack Trace**, **+ Component Tree**, and **+ Scoped Variables**. At the bottom of the error page, a timestamp indicates it was generated on Dec 30, 2012 at 2:33:10 PM.

Lors de l'évaluation de l'expression EL, JSF ne trouve aucun bean nommé **inscriptionBean** et affiche donc une page d'erreur détaillant le problème rencontré.

Pour terminer, annulons cette dernière modification en mettant à nouveau le bean **bonjourBean** dans l'expression EL, puis essayons de changer le nom de la méthode *setter* dans le bean. Modifiez par exemple les lignes 20 à 22 de la classe **BonjourBean** par :

Code : Java - Modification de la méthode setter

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

Rendez-vous alors à nouveau sur la page http://localhost:8088/test_jsf/bonjour.xhtml, cliquez sur le bouton et observez l'erreur affichée sur la figure suivante.

An Error Occurred:

```
javax.el.PropertyNotWritableException: /bonjour.xhtml @11,51 value="#{bonjourBean.nom}":  
The class 'com.sdzee.exemple.BonjourBean' does not have a writable property 'nom'.
```

+ Stack Trace

+ Component Tree

+ Scoped Variables

Dec 30, 2012 2:43:55 PM - Generated by Mojarra/Facelets

Lors de l'évaluation de l'expression EL, JSF ne trouve aucune méthode *setter* pour la propriété **nom** du bean **bonjourBean**. En effet, la règle pour un JavaBean impose que la méthode soit correctement nommée, et puisque nous avons changé sa dénomination, JSF considère alors qu'il n'existe pas de méthode *setter*. Il nous prévient donc logiquement qu'une exception `javax.el.PropertyNotWritableException` est levée, et que la propriété *nom* est considérée comme étant en lecture seule.

Nous y voilà, nous avons observé le fonctionnement correct de notre application et fait le tour des erreurs les plus courantes. Avant de passer aux bonnes pratiques, analysons brièvement ce qui a changé par rapport à notre ancienne méthode, sans JSF :

- auparavant, nous aurions dû écrire deux pages JSP, en lieu de place de nos deux simples Facelets ;
- auparavant, nous aurions dû écrire une servlet, chargée :
 - d'afficher la page bonjour.jsp lors de la réception d'une requête GET ;
 - de récupérer le contenu du champ texte du formulaire lors de la réception d'une requête POST, puis de s'en servir pour initialiser la propriété du bean BonjourBean, que nous aurions d'ailleurs dû créer, avant de transmettre le tout sous forme d'attributs à la page bienvenue.jsp pour affichage.

Ça paraît peu a priori, mais rendez-vous bien compte de ce dont nous n'avons plus à nous soucier avec JSF :

1. **la manipulation des objets requête et réponse** : avec JSF, nous n'avons pas conscience de ces objets, leur existence nous est masquée par le *framework*.
2. **l'extraction des paramètres contenus dans une requête HTTP** : avec JSF, il nous suffit d'écrire une EL au sein d'une Facelet, et le tour est joué.
3. **l'initialisation manuelle des beans** : avec JSF, le cycle de vie des beans annotés est entièrement géré par le *framework* ! À aucun moment nous n'avons initialisé un bean, et à vrai dire, à aucun moment nous n'avons créé une classe Java susceptible de pouvoir procéder à cette initialisation ! Tout ce que nous avons mis en place, ce sont deux Facelets et un bean...
4. **la mise en place d'attributs dans un objet HttpServletRequest pour transmission à une page JSP** : avec JSF, il suffit d'écrire une EL dans un composant de notre Facelet, et le *framework* s'occupe du reste.
5. **la redirection manuelle vers une page JSP pour affichage** : avec JSF, il suffit de préciser la page ciblée dans l'attribut `action` du composant `<h:commandButton>`.



Ainsi, vous pouvez déjà vous rendre compte de la simplification opérée par JSF : rien que sur ce petit exemple extrêmement simple, le code à produire est bien plus léger que si nous avions fait du MVC à la main !

Enfin, je vous invite à examiner le code source HTML de la page **bonjour.xhtml** que vous visualisez depuis votre navigateur. Vous y trouverez le code HTML produit par le rendu des différents composants de votre Facelet, c'est-à-dire le rendu des balises `<h:form>`, `<h:inputText>`, etc. Vous comprendrez alors mieux ce que je vous annonçais dans le chapitre précédent : avec un *framework* basé sur les composants, vous n'êtes plus aussi maîtres du HTML final envoyé à l'utilisateur que vous l'étiez lorsque vous créiez vos vues entièrement à la main par l'intermédiaire de pages JSP.

Les ressources

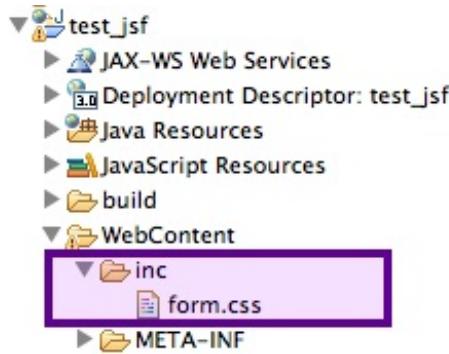
Pour terminer ce chapitre difficile et relativement abstrait sur une note plus légère, nous allons découvrir le système des ressources mis en place dans les projets JSF.

Vous avez dû vous en rendre compte, nous n'avons pas utilisé de styles CSS dans notre précédent exemple. Je vous rassure tout de suite, nous aurions pu faire comme nous en avions l'habitude, et déclarer une feuille CSS dans la section header de notre Facelet **bonjour.xhtml** :

Code : HTML- Déclaration d'une feuille CSS

```
<h:head>
    <title>Premier exemple JSF 2.0</title>
    <link type="text/css" rel="stylesheet" href="inc/form.css" />
</h:head>
```

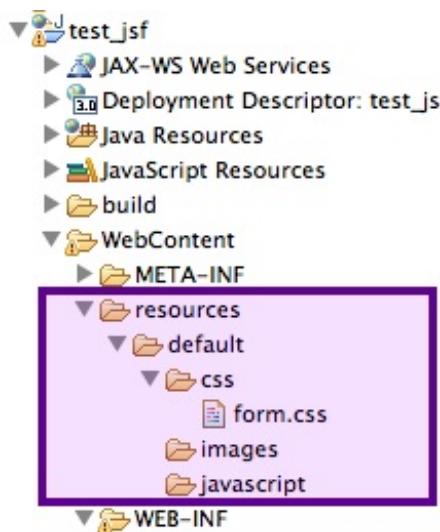
Nous aurions alors bien évidemment dû prendre soin de recopier la feuille **form.css** et le dossier **/inc** que nous utilisions dans nos précédents exemples, et de les placer à la racine du projet, comme indiqué sur la figure suivante.



Faites ces modifications, rendez-vous une nouvelle fois sur la page http://localhost:8088/test_jsf/bonjour.xhtml depuis votre navigateur, et constatez les légers changements intervenant dans l'affichage des éléments HTML.

Tout est donc normal, cependant **nous n'allons pas procéder de cette manière dans la suite du cours**. Avec JSF 2.x, une bonne pratique est de placer toutes les ressources web telles que les fichiers CSS, les images ou les fichiers JavaScript, dans un répertoire intitulé **resources** et placé directement à la racine de votre application, c'est-à-dire au même niveau que le dossier **/WEB-INF**.

Cette bonne pratique veut que nous mettions en place ce dossier **resources**, et que nous y créions des sous-dossiers pour délimiter les différentes ressources web que nous souhaitons utiliser. Si par exemple nous avions des images, des feuilles CSS et des scripts JS, pour respecter la bonne pratique nous pourrions par exemple créer l'arborescence indiquée sur la figure suivante et y placer nos différentes ressources.



Pour le moment, nous n'avons que le fichier **form.css** à y mettre.

Une fois ceci en place, nous pouvons maintenant utiliser un composant dédié à l'inclusion de ressources depuis nos Facelets. Par exemple, pour utiliser la feuille CSS **form.css** placée dans le dossier **/resources/default/css**, il suffit de remplacer l'inclusion traditionnelle via `<link type="text/css" ...>` par :

Code : HTML

```
<h:outputStylesheet library="default" name="css/form.css" />
```

Ce composant JSF permet de cibler directement le sous-dossier du répertoire **resources** via le contenu de son attribut **library**, que nous avons donc renseigné par "default" afin d'accéder au sous-dossier que nous avons mis en place. Il cible enfin le fichier souhaité via le contenu de son attribut **name**.

Effectuez les modifications (création du dossier **/resources/default/css**, déplacement du fichier **form.css** et suppression de l'ancien dossier **/inc**, puis modification de l'appel au fichier CSS depuis votre Facelet) et rendez-vous à nouveau sur la page depuis votre navigateur. Vous n'observerez aucun changement, preuve que le composant a bien fonctionné et a correctement inclus votre feuille CSS !



Le composant JSF ici utilisé est dédié à l'inclusion de feuilles CSS. Il en existe d'autres similaires, pour l'inclusion des fichiers JavaScript et des images. Nous y reviendrons le moment venu dans la suite du cours.

JSF, ou ne pas JSF ?



Faut-il opter pour JSF, ou du MVC avec des servlets/JSP faites maison ?

Dans le cas d'une application web destinée à votre apprentissage personnel, créer votre propre *framework* n'est pas une mauvaise idée, c'est un très bon exercice. Cela dit, si le long terme est envisagé pour votre application, ce n'est clairement pas judicieux. La plupart des *frameworks* MVC existants sont bien pensés, et la majorité des imprévus y sont pris en compte. En outre, l'API d'un *framework* public reconnu est bien documentée et maintenue par une communauté tierce.

Dans un autre registre, si jamais votre application devient populaire et que vous devez intégrer des développeurs supplémentaires à votre projet, afin par exemple de satisfaire aux divers besoins du client, il est bien plus aisé de trouver quelqu'un qui est déjà familier avec un *framework* existant. Avec un *framework* fait maison et probablement buggé, vous trouverez peu de développeurs prêts à se former sur une telle technologie et à prendre en charge la maintenance sachant pertinemment qu'ils ne réutiliseront probablement jamais cette technologie dans leurs projets futurs...

Enfin, sachez que tout cela ne s'applique pas uniquement à JSF, mais également à tous les autres *frameworks* populaires existant, comme Spring MVC par exemple.



Quelle est la différence entre JSF et Servlets/JSP/HTML/CSS/JS ?

Pour faire une analogie avec une technologie d'actualité dans le domaine du web, comparer JSF au pur combo Servlets/JSP/HTML/CSS/JS revient à comparer jQuery à du pur JavaScript : faire plus avec moins de code. Pour prendre PrimeFaces comme exemple, explorez sa [vitrine](#) et découvrez des exemples de codes complets. RichFaces propose lui aussi une [vitrine](#) avec des exemples de codes complets. Si vous étudiez avec attention ces exemples, vous comprendrez alors que vous n'avez bien souvent pas à vous soucier de la qualité du rendu HTML/CSS/JS d'une part, et d'autre part que vous n'avez besoin que d'un simple bean pour réaliser le modèle et d'une simple page XHTML pour la vue.

Remarquez toutefois que vous ne devez pas voir JSF comme une alternative à HTML/CSS/JS uniquement, mais bien prendre en compte également la partie serveur (typiquement JSP/Servlets). JSF permet d'éviter tout le code passe-partout en charge du regroupement des paramètres de requêtes HTTP, de leur conversion/validation, de la mise à jour des données du modèle et de l'exécution de la bonne méthode Java pour réaliser les traitements métiers. Avec JSF, vous n'avez plus qu'une page XHTML en guise de vue, et un JavaBean en tant que modèle. Cela accélère le développement de manière significative !

Bien entendu, comme c'est le cas avec tous les *frameworks* MVC basés sur les composants, vous disposez avec JSF d'une faible marge de contrôle sur le rendu HTML/CSS/JS. Ajouter du code JS personnalisé n'est par exemple pas chose aisée. Si c'est un obstacle pour votre projet, regardez plutôt du côté des *frameworks* MVC basés sur les actions comme [Spring MVC](#). Vous devez toutefois savoir que vous allez devoir écrire tout ce code HTML/CSS/JS vous-mêmes et par le biais de pages JSP, là où les Facelets de JSF vous offriraient des templates avancés.

- JSF est un *framework* MVC basé sur les composants. C'est un **standard**, car il fait partie intégrante de la plate-forme Java EE 6.
- Avec JSF, une unique servlet joue entre autres le rôle d'aiguilleur géant : la **FacesServlet**.
- Avec JSF, il n'est plus nécessaire d'écrire de servlets : seuls des **Facelets** et des **backing-beans** sont nécessaires.
- Dans sa version actuelle, JSF tire partie des annotations Java pour simplifier grandement le développement et la configuration.
- Une Facelet est une simple page XHTML, contenant des balises qui lient littéralement la vue aux composants JSF, et leur associe d'éventuelles valeurs via des expressions EL.
- L'ensemble des balises d'une vue, qui peut être constituée d'une ou plusieurs Facelets, constitue l'arbre des composants associé à cette vue.
- Avec JSF, le processus de traitement d'une requête rassemble les étapes de récupération, conversion, validation et sauvegarde des données.
- Lors du développement d'une application JSF, il est possible d'activer un mode de debug pour faciliter le pistage des erreurs rencontrées.

La gestion d'un formulaire avec JSF

Maintenant que nous sommes familiers avec JSF, nous allons mettre en pratique et apprendre à gérer proprement un formulaire, en ajoutant de la complexité au fur et à mesure de notre progression.

Une fois n'est pas coutume, nous allons réécrire notre fonction d'inscription d'utilisateur. Comme d'habitude, nous allons ajouter un petit plus à notre système : après la création initiale du système en suivant MVC avec des JSP et des servlets, nous y avions ajouté une base de données pour commencer, puis nous y avions intégré JPA. Eh bien cette fois nous allons bien évidemment employer JSF, mais également rendre la validation du formulaire... ajaxisée ! Appétissant, n'est-ce pas ? 😊

Une inscription classique

Préparation du projet

Nous allons partir sur une base propre. Pour ce faire, créez un nouveau projet web dynamique sous Eclipse, et nommez-le **pro_jsf**. Créez-y alors un fichier de configuration **/WEB-INF/glassfish-web.xml** :

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

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE glassfish-web-app PUBLIC "-//GlassFish.org//DTD GlassFish
Application Server 3.1 Servlet 3.0//EN"
"http://glassfish.org/dtds/glassfish-web-app_3_0-1.dtd">
<glassfish-web-app>
    <context-root>/pro_jsf</context-root>
    <class-loader delegate="true"/>
    <jsp-config>
        <property name="keepgenerated" value="true">
            <description>Conserve une copie du code des servlets auto-
générées.</description>
        </property>
    </jsp-config>
</glassfish-web-app>
```

Copiez-y ensuite le fichier **web.xml** que nous avions mis en place dans le projet **test_jsf**, fichier que nous pouvons, comme je vous l'avais expliqué, réutiliser tel quel.

Avant de poursuivre, je vous conseille de mettre en place une petite configuration particulière, afin d'éviter de futurs ennuis. Par défaut, le contenu d'un champ laissé vide dans vos formulaires sera considéré comme une chaîne vide. Et vous devez le savoir, **les chaînes vides sont l'ennemi du développeur** ! Heureusement, il est possible de forcer le conteneur à considérer un tel contenu comme une valeur nulle plutôt que comme une chaîne vide, en ajoutant cette section dans votre fichier **web.xml** :

Code : XML - Configuration de la gestion des chaînes vides dans /WEB-INF/web.xml

```
...
<context-param>
    <param-
    name>javax.faces.INTERPRET_EMPTY_STRING_SUBMITTED_VALUES_AS_NULL</param-
    name>
        <param-value>true</param-value>
</context-param>
...
```

Voilà tout pour le moment, nous reviendrons sur l'intérêt pratique de cette manipulation plus loin dans ce chapitre.

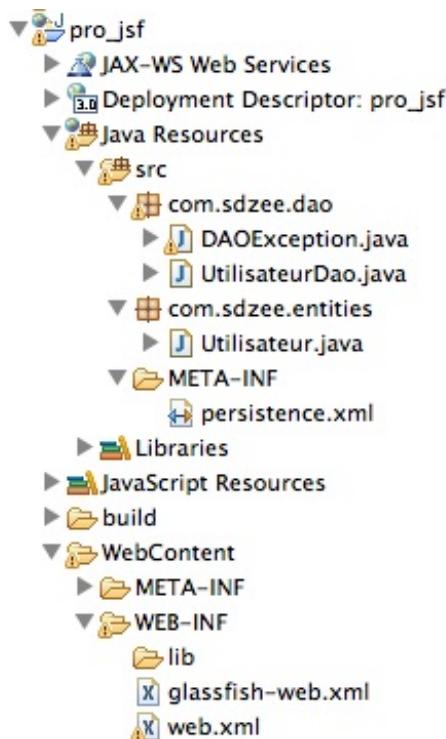
Création de la couche d'accès aux données

La première étape du développement, si c'en est une, est la "création" du modèle. En réalité, nous n'avons rien à faire ici : nous allons simplement réutiliser notre entité **Utilisateur** telle que nous l'avions développée dans notre projet **pro_jpa**, ainsi que notre

EJB Stateless ! Copiez donc simplement dans votre projet :

- la classe `com.sdzee.entities.Utilisateur`, en conservant le même package ;
- la classe `com.sdzee.dao.DAOException`, en conservant le même package ;
- la classe `com.sdzee.dao.UtilisateurDao`, en conservant le même package ;
- le fichier de configuration de JPA `src/META-INF/persistence.xml`, en conservant le même répertoire.

Voici sur la figure suivante l'arborescence que vous devez obtenir une fois arrivés à cette étape.



Création du backing bean

Nous l'avons découvert dans le chapitre précédent, un nouvel objet propre à JSF fait son apparition : le **Backing Bean**. Il s'agit en réalité d'une sorte de mini-contrôleur MVC, une sorte de glue qui relie la vue (la page JSF) au modèle de données (l'entité). Cet objet est littéralement lié à la vue, et **tous les attributs de l'entité sont exposés à la vue à travers lui**.

 Pour faire l'analogie avec ce que nous avions développé dans nos précédents exemples, cet objet va remplacer notre ancien **InscriptionForm**. Toutefois, nous n'allons pour le moment pas nous encombrer avec les méthodes de validation des différents champs du formulaire, et nous nous contenterons de mettre en place une inscription sans vérifications. Nous compléterons ensuite notre système, lorsque nous aurons construit une base fonctionnelle.

Sur la forme, ce **Backing-bean** se présente comme un bean classique, aux annotations JSF près. Puisqu'il est associé à une action, il est courant de le nommer par un verbe représentant l'action effectuée. Nous allons donc logiquement dans le cadre de notre exemple créer un bean intitulé **InscrireBean** :

Code : Java - `com.sdzee.beans.InscrireBean`

```

package com.sdzee.beans;

import java.io.Serializable;
import java.sql.Timestamp;

import javax.ejb.EJB;
import javax.faces.application.FacesMessage;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.RequestScoped;
import javax.faces.context.FacesContext;

```

```

import com.sdzee.dao.UtilisateurDao;
import com.sdzee.entities.Utilisateur;

@ManagedBean
@RequestScoped
public class IncrireBean implements Serializable {
    private static final long serialVersionUID = 1L;

    private Utilisateur utilisateur;

    // Injection de notre EJB (Session Bean Stateless)
    @EJB
    private UtilisateurDao utilisateurDao;

    // Initialisation de l'entité utilisateur
    public IncrireBean() {
        utilisateur = new Utilisateur();
    }

    // Méthode d'action appelée lors du clic sur le bouton du
    formulaire
    // d'inscription
    public void inscrire() {
        initialiserDateInscription();
        utilisateurDao.creer( utilisateur );
        FacesMessage message = new FacesMessage( "Succès de
l'inscription !" );
        FacesContext.getCurrentInstance().addMessage( null, message
);
    }

    public Utilisateur getUtilisateur() {
        return utilisateur;
    }

    private void initialiserDateInscription() {
        Timestamp date = new Timestamp( System.currentTimeMillis()
);
        utilisateur.setDateInscription( date );
    }
}

```

Notre objet contient tout d'abord une référence à notre entité **Utilisateur** à la ligne 20, à laquelle est associée une méthode *getter* lignes 40 à 42. Cette entité **Utilisateur** est initialisée depuis un simple constructeur public et sans argument aux lignes 27 à 29.

Notre DAO Utilisateur, qui pour rappel est depuis l'introduction de JPA dans notre projet un simple EJB Stateless, est injecté automatiquement via l'annotation `@EJB` à la ligne 24, exactement comme nous l'avions fait depuis notre servlet dans le projet `pro_jpa`.

Enfin, une méthode d'action nommée **inscrire()** est chargée :

- d'initialiser la propriété **dateInscription** de l'entité **Utilisateur** avec la date courante, via la méthode `initialiserDateInscription()` que nous avons créée aux lignes 44 à 47 pour l'occasion ;
- d'enregistrer l'utilisateur en base, via un appel à la méthode `creer()` du DAO Utilisateur ;
- d'initialiser un message de succès de la validation.

Dans cette dernière étape, deux nouveaux objets apparaissent :

- `FacesMessage` : cet objet permet simplement de définir un message de validation, que nous précisons ici en dur directement dans son constructeur. Il existe d'autres constructeurs, notamment un qui permet d'associer à un message un niveau de criticité, en précisant une catégorie qui est définie par `FacesMessage.Severity`. Les niveaux existants sont représentés par des constantes que vous pouvez retrouver sur la documentation de l'objet `FacesMessage`. En ce qui nous concerne, nous ne spécifions qu'un message dans le constructeur, et c'est par conséquent la criticité `Severity.INFO` qui est appliquée par défaut à notre message par JSF ;
- `FacesContext` : vous retrouvez là l'objet dont je vous ai annoncé l'existence dans le chapitre précédent, celui qui contient l'arbre des composants d'une vue ainsi que les éventuels messages d'erreur qui leur sont associés. Eh bien ici,

nous nous en servons pour mettre en place un **FacesMessage** dans le contexte courant via la méthode `addMessage()`, pour que la réponse puisse ensuite l'afficher. Je vous laisse parcourir sa documentation, et nous reviendrons ensemble sur l'intérêt de passer **null** en tant que premier argument de cette méthode lorsque nous développerons la vue.

Par défaut, nous avons annoté notre bean avec `@RequestScoped` pour le placer dans la portée requête : en effet, notre objet ne va intervenir qu'à chaque demande d'inscription et n'a donc pas vocation à être stocké plus longtemps que le temps d'une requête.



Au passage, vous remarquez ici pourquoi il est très important d'avoir découvert comment fonctionne une application Java EE MVC sans *framework*. Si vous n'aviez pas conscience de ce qui se passe derrière les rideaux, notamment des différentes portées existantes dans une application et des allers-retours de paires requête/réponse qui ont lieu à chaque intervention de l'utilisateur depuis son navigateur, vous auriez beaucoup plus de mal à saisir comment manipuler vos objets avec JSF !

Si vous avez bien observé le code de ce **backing-bean**, et si vous vous souvenez de celui de notre ancien objet métier **InscriptionForm**, vous devez instantanément vous poser la question suivante :



Où sont les méthodes de récupération et conversion des valeurs envoyées depuis le formulaire ?

Eh oui, dans notre bean nous nous contentons simplement d'initialiser la date d'inscription dans notre entité **Utilisateur**, car ce n'est pas une information saisie par l'utilisateur. Mais en ce qui concerne toutes les autres propriétés de notre entité, nous ne faisons strictement rien. Vous retrouvez ici ce que je vous ai expliqué dans la description du processus du traitement d'une requête avec JSF, le fameux parcours en six étapes. Les étapes de récupération, conversion, validation et enregistrement dans le modèle sont entièrement automatisées ! Et c'est depuis la vue que nous allons directement effectuer les associations entre les champs du formulaire et les propriétés de notre entité.

Création de la vue

Nous devons ensuite créer la Facelet générant le formulaire d'inscription. Pour rappel, une Facelet n'est qu'une simple page XHTML contenant des balises JSF. Je vous donne dès maintenant le code de la vue dans son intégralité, prenez le temps de bien regarder les composants qui interviennent et nous en reparlons en détail ensuite.

Code : HTML- /inscription.xhtml

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
    <h:head>
        <meta charset="utf-8" />
        <title>Inscription</title>
        <h:outputStylesheet library="default" name="css/form.css" />
    </h:head>
    <h:body>
        <h:form>
            <fieldset>
                <legend>Inscription</legend>
                <h:outputLabel for="email">Adresse email <span
class="requis">*</span></h:outputLabel>
                <h:inputText id="email"
value="#{inscrireBean.utilisateur.email}" required="true" size="20"
maxlength="60" />
                <h:message id="emailMessage" for="email"
errorClass="erreur" />
                <br />

                <h:outputLabel for="motdepasse">Mot de passe <span
class="requis">*</span></h:outputLabel>
                <h:inputSecret id="motdepasse"
value="#{inscrireBean.utilisateur.motDePasse}" required="true"
size="20" maxlength="20" />
            </fieldset>
        </h:form>
    </h:body>
</html>
```

```

<h:message id="motDePasseMessage" for="motdepasse"
errorClass="erreur" />
<br />

<h:outputLabel for="confirmation">Confirmation du mot de
passe <span class="requis">*</span></h:outputLabel>
<h:inputSecret id="confirmation"
value="#{inscrireBean.utilisateur.motDePasse}" required="true"
size="20" maxlength="20" />
<h:message id="confirmationMessage" for="confirmation"
errorClass="erreur" />
<br />

<h:outputLabel for="nom">Nom d'utilisateur <span
class="requis">*</span></h:outputLabel>
<h:inputText id="nom"
value="#{inscrireBean.utilisateur.nom}" required="true" size="20"
maxlength="20" />
<h:message id="nomMessage" for="nom" errorClass="erreur" />
<br />

<h:messages globalOnly="true" infoClass="info" />

<h:commandButton value="Inscription"
action="#{inscrireBean.inscrire}" styleClass="sansLabel" />
<br />
</fieldset>
</h:form>
</h:body>
</html>

```

Vous devez reconnaître la structure globale de la page, identique aux premiers exemples de Facelets que nous avons mis en place dans le chapitre précédent :

- un en-tête HTML particulier contenant la déclaration de la bibliothèque de composants, préfixés par h: ;
- une section header contenant l'inclusion de la feuille de style CSS **form.css**, que vous prendrez soin de recopier depuis le projet **test_jsf** en recréant l'arborescence **/resources/default/css** dans votre projet ;
- puis le corps de la page contenant un formulaire généré par le composant **<h:form>**.

Nous allons maintenant détailler comment nous procédons à la création des différents éléments du formulaire, en analysant les composants qui interviennent. Comme vous pourrez le constater en parcourant la documentation de chacune des balises, elles supportent toutes un nombre conséquent d'attributs. Nous allons par conséquent limiter notre analyse à ceux qui nous sont ici utiles. Si vous souhaitez connaître en détail toutes les possibilités offertes par chaque balise, je vous invite à parcourir leurs documentations en intégralité et à faire vos propres tests pour vérifier que vous avez bien compris.

Les labels

Pour générer un label associé à un champ de formulaire, concrétisé par la balise HTML **<label>**, il faut utiliser le composant **<h:outputLabel>**. Le comportement de l'attribut **for** est identique à celui de la balise HTML, il suffit d'y préciser l'**id** du champ de saisie auquel le label fait référence.

Les champs de saisie

Pour générer un champ de saisie de type texte, concrétisé par la balise HTML **<input type="text" ...>**, il faut utiliser le composant **<h:inputText>**. Tout comme la balise HTML, il accepte les attributs **id**, **value**, **size** et **maxlength**.

Pour générer un champ de saisie de type mot de passe, concrétisé par la balise HTML **<input type="password" ...>**, il faut utiliser le composant **<h:inputSecret>**. Il accepte lui aussi les attributs **id**, **value**, **size** et **maxlength**.

Petite nouveauté, nous utilisons un attribut nommé **required**, qui peut prendre comme valeur **true** ou **false**, et qui va déterminer si l'utilisateur doit obligatoirement saisir des données dans ce champ ou non. En réalité, il s'agit là d'un marqueur qui va être appliqué au composant, et qui va permettre de générer une erreur lors de la validation de la valeur du champ associé. Si un champ est marqué comme requis et qu'aucune valeur n'est entrée par l'utilisateur, alors un message d'erreur sera

automatiquement placé dans le FacesContext par JSF, et nous pourrons ensuite l'afficher à l'utilisateur dans la réponse.

Ce qu'il faut bien observer ici, c'est l'emploi d'expressions EL pour cibler la propriété **utilisateur** de notre **backing-bean**, par exemple `# { inscrireBean.utilisateur.email}` à la ligne 13. Comme je vous l'ai déjà dit, **tous les attributs de l'entité sont exposés à la vue à travers le backing-bean**. Voilà pourquoi nous devons d'abord cibler l'entité **utilisateur**, qui est elle-même une propriété du backing-bean **inscrireBean**, puis cibler la propriété de l'entité désirée. Vous comprenez mieux maintenant pourquoi je vous ai dit que ce bean était la glue qui reliait la vue au modèle !

En outre, comprenez bien que la liaison créée par l'intermédiaire de cette expression EL placée dans l'attribut **value** d'un composant de saisie est bidirectionnelle :

- lors du rendu de la page HTML, la valeur contenue dans le modèle qui est retournée par cette expression EL sera affichée dans le champ de saisie ;
- lors de l'envoi des données du formulaire par l'utilisateur, la valeur contenue dans le champ de saisie sera utilisée pour mettre à jour la valeur contenue dans le modèle.

Les messages

Depuis le temps que je vous parle de ces fameux messages d'erreurs, nous y voilà. La différence la plus marquante entre la page JSP que nous avions utilisée jusqu'à présent et notre Facelet fraîchement créée est l'utilisation d'un composant JSF pour générer un message associé à un champ de saisie en particulier. Il s'agit du **composant <h:message>**. Entre autres, celui-ci accepte deux attributs qui nous sont ici utiles :

- **for**, pour cibler l'id du champ concerné ;
- **errorClass**, pour permettre de donner une classe CSS particulière lors de l'affichage du message généré s'il s'agit d'une erreur.

Ce composant va donc afficher, lors du rendu de la réponse, l'éventuel message associé au champ ciblé par l'attribut **for**. S'il s'agit d'un message d'erreur, que JSF sait différencier des messages d'information qui peuvent éventuellement être placés dans le FacesContext grâce au niveau de criticité associé à un message dont je vous ai parlé un peu plus tôt, alors le style **erreur** défini dans notre feuille CSS sera appliqué.

Nous utilisons en fin de page un autre élément responsable de l'affichage de messages à l'utilisateur : **le composant <h:messages>**. Par défaut, celui-ci provoque l'affichage de tous les messages disponibles dans la vue, y compris ceux qui sont déjà affichés via un **<h:message>** ailleurs dans la page. Toutefois, il est possible de n'afficher que les messages qui ne sont attachés à aucun composant défini, c'est-à-dire les messages dont l'**id** est **null**, en utilisant l'attribut optionnel **globalOnly="true"** :

Code : HTML

```
<h:messages globalOnly="true" />
```

Vous comprenez maintenant pourquoi dans la méthode **inscrire()** de notre **backing-bean**, nous avons passé **null** en paramètre de la méthode **FacesContext.addMessage()** : c'est pour pouvoir distinguer notre message à caractère général (nous nous en servons pour stocker le résultat final de l'inscription) des messages liés aux composants de la vue. Comprenez donc bien que le code suivant dans notre **backing-bean** attacherait le message donné au composant **<h:message for="clientId">**, et que nous passons **null** pour n'attacher notre message à aucun composant existant.

Code : Java

```
facesContext.addMessage("clientId", facesMessage);
```

Notre message a ainsi un caractère **global**. Voilà d'ailleurs pourquoi l'attribut de la balise **<h:messages>** permettant de cibler uniquement ce type de messages s'intitule... **globalOnly** !

Enfin, nous utilisons l'attribut **infoClass** pour donner à notre message global le style **info** qui est défini dans notre feuille CSS. Nous pourrions utiliser également l'attribut **styleClass**, mais puisque JSF permet de différencier les messages selon leur gravité, autant en profiter !

Le bouton d'envoi

Pour générer un bouton de soumission de formulaire, concrétisé par la balise HTML `<input type="submit" ...>`, il faut utiliser le composant `<h:commandButton>`. Nous l'avons déjà étudié dans notre premier exemple : le contenu de son attribut `value` est affiché en tant que texte du bouton HTML généré, et son attribut `action` permet de définir la navigation. À la différence de notre premier exemple cependant, où nous redirigeons l'utilisateur vers une autre page, nous utilisons ici une expression EL pour appeler une méthode de notre **backing-bean**, en l'occurrence notre méthode d'action `inscrire()`. Vous retrouvez ici ce que je vous ai expliqué en vous présentant la technologie EL : les expressions se basant sur la syntaxe `#{...}` permettent d'appeler n'importe quelle méthode d'un bean, et pas seulement une méthode *getter* comme c'était le cas avec `$ {...}`.

Nous utilisons enfin l'attribut `styleClass`, pour appliquer au bouton HTML généré la classe `sansLabel` définie dans notre feuille CSS.



Nous avons fait le tour de tout ce qu'il faut savoir sur notre léger système d'inscription. Pour le moment, aucun contrôle de validation n'est effectué hormis les simples `required="true"` sur les champs du formulaire. De même, aucune information n'est affichée hormis un message d'erreur sur chaque champ laissé vide, et notre message de succès lorsque l'inscription fonctionne.

Tests & observations

Notre projet est maintenant prêt pour utilisation. En fin de compte, seuls une Facelet et un backing bean sont suffisants, le reste étant récupéré depuis notre projet JPA. Vérifions pour commencer le bon fonctionnement de l'application en nous rendant sur l'URL `http://localhost:8088/pro_jsf/bonjour.xhtml` depuis notre navigateur. Vous devez observer le formulaire d'inscription tel qu'il existait dans nos précédents exemples.

Si ce n'est pas le cas, c'est que vous avez oublié quelque chose en cours de route. Vérifiez bien que vous avez :

- copié les classes **Utilisateur**, **UtilisateurDao** et **DAOException** en conservant leurs packages respectifs, depuis le projet `pro_jpa` ;
- copié le fichier **META-INF/persistence.xml** depuis le projet `pro_jpa` ;
- copié le fichier **form.css** depuis le projet `test_jsf`, en le plaçant dans l'arborescence `/resources/default/css/` ;
- copié le fichier **web.xml** depuis le projet `test_jsf` ;
- démarré votre serveur MySQL ;
- démarré votre serveur GlassFish ;
- déployé votre projet `pro_jsf` sur le serveur GlassFish.

Le formulaire s'affichant correctement, nous pouvons alors tester une inscription. Pour ce premier cas, nous allons essayer avec des informations valides, et qui n'existent pas déjà en base. Par exemple, avec une adresse jamais utilisée auparavant, deux mots de passe corrects et identiques et un nom d'utilisateur suffisamment long, comme indiqué à la figure suivante.

The screenshot shows a web browser window with the address bar displaying "localhost:8088/pro_jsf/inscription.xhtml". The main content is a form titled "Inscription". It contains four input fields with asterisks indicating they are required:

- Adresse email *: coyote@sdz.fr
- Mot de passe *: ****
- Confirmation du mot de passe *: ****
- Nom d'utilisateur *: abcdefg

Below the inputs is a large blue "Inscription" button.

Après un clic sur le bouton d'inscription, l'inscription fonctionne et le message de succès est affiché (voir la figure suivante).

Inscription

Adresse email *

Mot de passe *

Confirmation du mot de passe *

Nom d'utilisateur *

• Succès de l'inscription !

Inscription

Nous constatons alors que :

- c'est la page courante qui est rechargée. Comme je vous l'ai déjà expliqué, par défaut et sans règle de navigation particulière précisée par le développeur, c'est la page courante qui est automatiquement utilisée comme action d'un formulaire JSF ;
- le contenu des champs de saisie des mots de passe n'est pas réaffiché, alors que l'expression EL est bien présente dans l'attribut **value** des champs. Ce comportement est voulu, car il ne faut jamais retransmettre un mot de passe après validation d'un formulaire. Nous avions d'ailleurs pris garde à ne pas le faire dans notre ancienne page JSP d'inscription, si vous vous souvenez bien. Avec JSF, le composant **<h:inputSecret>** est programmé pour ne pas renvoyer son contenu, il n'y a donc plus d'erreur d'inattention possible ;
- le message de succès est bien décoré avec le style décrit par la classe **info** de notre feuille CSS. JSF a donc bien utilisé la classe précisée dans l'attribut **infoClass** de la balise **<h:messages>**, ce qui est une preuve que le *framework* a bien attribué par défaut le niveau **Severity**. **INFO** au message que nous avons construit depuis notre **backing-bean**.

Essayons maintenant de nous inscrire en entrant des informations invalides, comme par exemple une adresse email déjà utilisée (celle que vous venez de saisir pour réaliser l'inscription précédente ira très bien).

Nous constatons alors l'échec de notre système, qui plante et affiche un joli message de debug JSF. Toutefois, pas d'inquiétude, c'était prévu : puisque nous n'avons encore mis en place aucun contrôle, l'inscription a été tentée sans vérifier auparavant si l'adresse email existait déjà en base. MySQL a retourné une exception lors de cette tentative, car il a trouvé une entrée contenant cette adresse dans la table Utilisateur.

C'est très fâcheux, mais nous n'allons pas nous occuper de ce problème tout de suite. Poursuivons nos tests, et essayons cette fois-ci de nous inscrire en laissant plusieurs champs du formulaire vides. Par exemple, retournons sur le formulaire, saisissons uniquement une adresse email et cliquons sur le bouton d'inscription (voir la figure suivante).

Inscription

Adresse email *

Mot de passe *

Confirmation du mot de passe *

Nom d'utilisateur *

j_idt7:motdepasse: Validation Error: Value is required.
j_idt7:confirmation: Validation Error: Value is required.
j_idt7:nom: Validation Error: Value is required.

Inscription

Nous constatons alors l'affichage de messages d'erreurs à côté de chacun des champs laissés vides : c'est le fruit du composant **<h:message>** ! En outre, ces messages étant spécifiés comme étant des erreurs par JSF, le *framework* utilise l'attribut **errorClass** et les décore avec la classe **erreur** de notre feuille CSS : voilà pourquoi ces messages apparaissent en rouge.

Par contre, nous observons que ces messages automatiquement générés par JSF sont vraiment bruts de décoffrage... Il nous faut trouver un moyen de les rendre plus **user-friendly**, et c'est ce à quoi nous allons nous atteler dès maintenant.

Amélioration des messages affichés lors de la validation

Les messages automatiques générés par JSF sur chaque champ de notre formulaire sont vraiment laids, et cela s'explique très simplement : par défaut, JSF fait précéder ses messages d'erreurs des identifiants des objets concernés. En l'occurrence, il a concaténé l'identifiant de notre formulaire (**j_id7**) et celui de chaque champ (**motdepasse**, **confirmation** et **nom**) en les séparant par le caractère :.



Quand avons-nous donné cet id barbare à notre formulaire ?

Eh bien en réalité, nous ne lui avons jamais donné d'identifiant, et JSF en a donc généré un par défaut, voilà pourquoi il est si laid. Ainsi, pour rendre ces messages moins repoussants, nous pouvons donc commencer par donner un **id** à notre formulaire. Nous allons par exemple l'appeler "formulaire", en changeant sa déclaration dans notre Facelet de `<h:form id="formulaire">`. Appliquez cette modification au code, puis tentez à nouveau le test précédent.

C'est déjà mieux, mais ça reste encore brut. Si nous regardons la documentation du composant `<h:inputText>`, nous remarquons qu'il présente un attribut intitulé **label**, qui est utilisé pour représenter un champ de manière littérale. Nous allons donc ajouter un attribut **label** à chacune des balises déclarant un composant `<h:inputText>` ou `<h:inputSecret>` dans notre Facelet :

Code : HTML-Ajout d'un label sur chaque composant

```
<h:inputText id="email" value="#{inscrireBean.utilisateur.email}" ...
... label="Email" />
<h:inputSecret id="motdepasse"
value="#{inscrireBean.utilisateur.motDePasse}" ... label="Mot de
passe" />
<h:inputSecret id="confirmation"
value="#{inscrireBean.utilisateur.motDePasse}" ...
label="Confirmation" />
<h:inputText id="nom" value="#{inscrireBean.utilisateur.nom}" ...
label="Nom" />
```

Effectuez ces modifications, puis faites à nouveau le test (voir la figure suivante).

C'est un peu mieux, mais c'est encore brut, et c'est toujours en anglais... Pour régler ce problème une fois pour toutes, nous allons utiliser l'attribut **requiredMessage** des composants de saisie JSF qui, d'après leur documentation, permet de définir le message utilisé lors de la vérification de la règle définie par l'attribut **required**. En d'autres termes, nous allons y spécifier directement le message d'erreur à afficher lorsque le champ est laissé vide ! Nous allons donc laisser tomber nos attributs **label**, et les remplacer par ces nouveaux attributs **requiredMessage** :

Code : HTML-Ajout d'un requiredMessage sur chaque composant

```
<h:inputText id="email" value="#{inscrireBean.utilisateur.email}" ...
... requiredMessage="Veuillez saisir une adresse email" />
<h:inputSecret id="motdepasse"
value="#{inscrireBean.utilisateur.motDePasse}" ...
requiredMessage="Veuillez saisir un mot de passe" />
<h:inputSecret id="confirmation"
value="#{inscrireBean.utilisateur.motDePasse}" ...
requiredMessage="Veuillez saisir la confirmation du mot de passe" />
```

```
<h:inputText id="nom" value="#{inscrireBean.utilisateur.nom}" ...  
requiredMessage="Veuillez saisir un nom d'utilisateur" />
```

Effectuez ces modifications, puis faites à nouveau le test... Nous y voilà, les messages sont finalement propres et compréhensibles pour les utilisateurs. Petit bémol toutefois, du point de vue de la qualité du code, c'est un peu sale de définir directement en dur dans la vue les messages d'erreurs à afficher... Comme le hasard fait très bien les choses, il existe justement une fonctionnalité dans JSF qui permet de définir des messages dans un fichier externe, et d'y faire référence depuis les Facelets. Ce système s'appelle un **bundle**, et nous allons en mettre un en place dans notre exemple.

Mise en place d'un bundle

Un bundle n'est rien d'autre qu'un fichier de type Properties, contenant une liste de messages. La première étape dans la création d'un bundle consiste donc à créer un fichier Properties et à y placer nos différents messages de validation. Nous allons par exemple nommer notre fichier **messages.properties**. Il faut le placer dans les sources du projet, aux côtés du code de l'application. En l'occurrence nous allons le placer dans un package nommé com.sdzee.bundle :

Code : Properties - com.sdzee.bundle.messages

```
inscription.email = Veuillez saisir une adresse email  
inscription.motdepasse = Veuillez saisir un mot de passe  
inscription.confirmation = Veuillez saisir la confirmation du mot de passe  
inscription.nom = Veuillez saisir un nom d'utilisateur
```

Nous avons ici placé nos quatre messages, identifiés par le nom de la Facelet qui en fait usage (inscription) suivi d'un point et du nom du champ concerné. La syntaxe à respecter est celle d'un fichier de type Properties Java classique.

Une fois ce dossier en place, il faut maintenant le charger depuis notre Facelet pour qu'elle puisse faire référence à son contenu. Pour ce faire, nous allons utiliser le composant `<f:loadBundle>`. Celui-ci doit être placé dans la page **avant** les balises qui en feront usage, typiquement nous pouvons le mettre dans le header de notre page :

Code : HTML - Ajout du chargement du bundle dans le header de la Facelet inscription.xhtml

```
<h:head>
    ...
    <f:loadBundle basename="com.sdzee.bundle.messages" var="msg"/>
</h:head>
```

Cette balise attend uniquement deux attributs :

- **basename**, qui contient le chemin complet dans lequel est placé le fichier (le package suivi du nom du fichier) ;
 - **var**, qui permet de définir par quel nom le bundle sera désigné dans le reste de la page.

Nous avons donc précisé le chemin `com.sdzee.bundle.messages`, et nous utiliserons le nom `msg` pour faire référence à notre bundle. Effectuez cet ajout dans le header de votre Facelet.

Il ne nous reste maintenant plus qu'à remplacer nos messages, actuellement en dur dans les attributs **requiredMessage**, par une référence vers les messages présents dans le bundle. Vous vous en doutez peut-être déjà, il suffit pour cela d'utiliser des expressions EL ! Voilà comment nous allons procéder :

Code : HTML - Remplacement des messages en dur dans les attributs requiredMessage

```
<h:inputText id="email" value="#{inscrireBean.utilisateur.email}" ...
... requiredMessage="#{msg['inscription.email']}" />
<h:inputSecret id="motdepasse"
value="#{inscrireBean.utilisateur.motDePasse}" ...
requiredMessage="#{msg['inscription.motdepasse']}" />
<h:inputSecret id="confirmation"
value="#{inscrireBean.utilisateur.motDePasse}" ...
requiredMessage="#{msg['inscription.confirmation']}" />
<h:inputText id="nom" value="#{inscrireBean.utilisateur.nom}" />
```

```
requiredMessage="#{msg['inscription.nom']} " />
```

La forme de l'expression EL utilisée est simple : nous ciblons le bundle via son nom **msg**, puis nous utilisons la notation avec les crochets pour cibler le nom souhaité.

Effectuez ces dernières modifications dans le code de votre Facelet, puis testez à nouveau votre formulaire. Si vous n'avez rien oublié et si vous avez correctement positionné votre fichier **bundle**, vous devriez observer exactement le même comportement que lors du test effectué avec les messages écrits en dur.



L'intérêt de cette technique, c'est bien évidemment d'écrire un code plus propre dans vos Facelets, mais surtout de regrouper tous vos messages de validation dans un seul et unique fichier. Pratique, notamment pour l'internationalisation ! Par exemple, si vous souhaitez traduire votre application dans une langue différente, vous n'avez alors qu'à changer de bundle : inutile de repasser sur chacune de vos Facelets pour traduire les messages un par un !

Nous allons nous arrêter là pour les tests et améliorations. Passons maintenant au paragraphe suivant, afin de rendre tout ce mécanisme... ajaxisé !

Une inscription ajaxisée

Présentation



Qu'est-ce que c'est, une inscription "ajaxisée" ?

AJAX est l'acronyme d'**Aynchronous Javascript and XML**, ce qui en français signifie littéralement « Javascript et XML asynchrones ». Derrière cette appellation se cache un ensemble de technologies qui permettent la mise à jour d'un fragment d'une page web sans que le rechargement complet de la page web visitée par l'utilisateur ne soit nécessaire. C'est ce type de technologie qui permet à certains sites de proposer des fonctionnalités avancées et intuitives à leurs utilisateurs. Citons par exemple le Site du Zéro qui propose l'auto-complétion lors de la saisie dans le champ de recherche d'un membre, ou encore le site Stackoverflow avec son système de vote en direct sur les réponses posées et questions apportées.

Ainsi, lorsque je parle d'inscription ajaxisée, je désigne en réalité le fait de pouvoir valider le contenu de chacun des champs de notre formulaire d'inscription sans nécessiter un clic sur le bouton d'envoi, ni nécessiter un rechargement de la page entière.

L'AJAX avec JSF

Si nous travaillions toujours à la main, il nous faudrait mettre les mains dans le cambouis et mettre en place du JavaScript, des traitements spéciaux dans nos servlets pour ne déclencher l'actualisation que d'un morceau de la page visitée par l'utilisateur, etc. Heureusement, avec JSF nous allons pouvoir garder nos mains propres : le *framework* nous propose un moyen ultra-simple pour court-circuiter le processus classique de traitement d'une requête, et permettre à un composant de s'actualiser de manière indépendante, et non pas dans le flot complet de l'arbre des composants présents dans la vue courante comme c'est le cas traditionnellement.

La solution offerte se matérialise sous la forme... d'un composant ! C'est cette fois une balise de la bibliothèque Core que nous allons utiliser : la bien nommée **<f:ajax>**. Sans plus attendre, je vous propose le nouveau code de notre Facelet, et nous en discutons ensuite.

Code : HTML - /inscription.xhtml

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html">
    <h:head>
        <meta charset="utf-8" />
        <title>Inscription</title>
        <h:outputStylesheet library="default" name="css/form.css" />
        <f:loadBundle basename="com.sdzee.bundle.messages" var="msg"/>
    </h:head>
```

```

<h:body>
    <h:form id="formulaire">
        <fieldset>
            <legend>Inscription</legend>
            <h:outputLabel for="email">Adresse email <span
class="requis">*</span></h:outputLabel>
            <h:inputText id="email"
value="#{inscrireBean.utilisateur.email}" required="true" size="20"
maxlength="60" requiredMessage="#{msg['inscription.email']}">
                <f:ajax event="blur" render="emailMessage" />
            </h:inputText>
            <h:message id="emailMessage" for="email"
errorClass="erreur" />
            <br />

            <h:outputLabel for="motdepasse">Mot de passe <span
class="requis">*</span></h:outputLabel>
            <h:inputSecret id="motdepasse"
value="#{inscrireBean.utilisateur.motDePasse}" required="true"
size="20" maxlength="20"
requiredMessage="#{msg['inscription.motdepasse']}">
                <f:ajax event="blur" render="motDePasseMessage" />
            </h:inputSecret>
            <h:message id="motDePasseMessage" for="motdepasse"
errorClass="erreur" />
            <br />

            <h:outputLabel for="confirmation">Confirmation du mot de
passe <span class="requis">*</span></h:outputLabel>
            <h:inputSecret id="confirmation"
value="#{inscrireBean.utilisateur.motDePasse}" required="true"
size="20" maxlength="20"
requiredMessage="#{msg['inscription.confirmation']}">
                <f:ajax event="blur" render="confirmationMessage" />
            </h:inputSecret>
            <h:message id="confirmationMessage" for="confirmation"
errorClass="erreur" />
            <br />

            <h:outputLabel for="nom">Nom d'utilisateur <span
class="requis">*</span></h:outputLabel>
            <h:inputText id="nom"
value="#{inscrireBean.utilisateur.nom}" required="true" size="20"
maxlength="20" requiredMessage="#{msg['inscription.nom']}">
                <f:ajax event="blur" render="nomMessage" />
            </h:inputText>
            <h:message id="nomMessage" for="nom" errorClass="erreur" />
            <br />

            <h:messages globalOnly="true" infoClass="info" />

            <h:commandButton value="Inscription"
action="#{inscrireBean.inscrire}" styleClass="sansLabel">
                <f:ajax execute="@form" render="@form" />
            </h:commandButton>
            <br />
        </fieldset>
    </h:form>
</h:body>
</html>

```

Les différences apparentes avec la version précédente de la Facelet sont minimes, seules quelques lignes font leur apparition :

- chaque balise `<h:inputText>` et `<h:inputSecret>` contient dorénavant un corps, et dans ce corps est placée la fameuse balise `<f:ajax>`;
- de même, la balise `<h:inputCommand>` contient dorénavant elle aussi un corps, dans lequel est logée une balise `<f:ajax>`.

Étudions la construction de cette nouvelle balise. Seuls deux de ses attributs nous sont utiles dans les champs de saisie :

- **event**, qui nous permet de définir l'action déclenchant l'envoi de la requête AJAX au serveur ;
- **render**, qui nous permet de définir le ou les composants dont le rendu doit être effectué, une fois la requête traitée par le serveur.

Autrement dit, c'est extrêmement simple : il suffit de placer la balise `<f:ajax>` dans le corps du champ qui est concerné par l'événement déclencheur, et de préciser dans l'attribut **render** le(s) composant(s) pour le(s)quel(s) nous souhaitons relancer un rendu graphique.

Dans notre cas, ce qui nous intéresse c'est de faire valider le contenu d'un champ dès que l'utilisateur a terminé la saisie, et d'actualiser le message associé au champ en conséquence :

- l'**event** que nous utilisons pour considérer que l'utilisateur a terminé la saisie du contenu d'un champ s'intitule **blur**. Cette propriété JavaScript va permettre de déclencher une requête dès que le champ courant perd le focus, c'est-à-dire dès que l'utilisateur clique dans un autre champ, ou ailleurs sur la page, ou bien lorsqu'il navigue en dehors du champ via la touche tabulation de son clavier par exemple ;
- pour désigner quel composant actualiser, nous devons simplement préciser son id dans l'attribut **render**.

La balise `<f:ajax>` présente sur chaque champ de saisie va soumettre (et donc valider !) seulement le contenu du champ courant, et déclencher un réaffichage du message associé lorsque le champ va perdre le focus. La balise `<f:ajax>` placée sur le bouton de validation est un peu différente des autres : elle permet d'effectuer un envoi complet, mais là encore ajaxisé et non pas simplement un envoi classique avec recharge de la page. Pour ce faire, vous constatez l'emploi d'un mot-clé un peu spécial : `@form`. Il existe quatre marqueurs de la sorte :

- `@this` : désigne le composant englobant la balise `<f:ajax>` ;
- `@form` : désigne le formulaire entier ;
- `@all` : désigne l'arbre des composants de la page entière ;
- `@none` : ne désigne aucun composant.

En précisant `@form` dans l'attribut **render**, nous nous assurons ainsi que tout le formulaire va être actualisé lors d'un clic sur le bouton d'envoi. Par ailleurs, nous n'utilisons plus l'attribut **event** comme nous le faisions sur les champs de saisie, car nous savons très bien que c'est un clic sur le bouton qui va déclencher l'action. Nous utilisons cette fois l'attribut **execute**, qui permet de définir la portée de l'action à effectuer : en l'occurrence, nous souhaitons bien traiter le formulaire complet.

Effectuez ces modifications dans votre Facelet, puis testez à nouveau votre formulaire d'inscription. À chaque fois que vous allez cliquer dans un champ de saisie, puis en dehors de ce champ, vous pourrez observer l'actualisation des éventuels messages d'erreurs associés à chaque champ, et ce presque en temps réel !



En fin de compte, JSF offre un masquage total de ce qui se passe sous la couverture ! Il suffit d'inclure une simple balise dans le corps d'un composant, et le tour est joué. C'est d'une facilité à la fois admirable et déconcertante ! 😊

L'importance de la portée d'un objet

Avant de passer à la suite, je tiens à vous faire remarquer quelque chose de très inquiétant : nous avons oublié de nous soucier de la portée de notre **Backing-bean** ! Souvenez-vous, celle que nous avions déclarée via l'annotation `@RequestScoped`. Nous ne nous en étions pas inquiétés plus que ça sur le moment, mais maintenant que nous avons ajaxisé notre formulaire, cette notion de portée va prendre de l'importance.

Actuellement, notre objet est dans la portée requête. Cela signifie que notre bean ne vit que le temps d'une requête : il est créé par JSF à l'arrivée d'une requête sur le serveur, puis détruit une fois la réponse renvoyée. Autrement dit, notre formulaire provoque la création et la destruction d'un bean sur le serveur à chaque fois que l'utilisateur change de champ ! C'est du gâchis de performances.

Nous pourrions envisager de mettre notre objet en session plutôt que dans la portée requête. Oui, mais la session est un objet qu'il est coûteux de maintenir pour le serveur, car il faut conserver un unique objet pour chaque visiteur utilisant l'application. Voilà pourquoi en pratique, il ne faut utiliser la session que lorsque c'est impérativement nécessaire, comme pour réaliser un panier d'achats par exemple, où les items commandés doivent absolument être conservés tout au long de la navigation du visiteur. Dans notre simple système d'inscription, garder nos informations serait du pur gâchis de mémoire.

Nous devons donc impérativement réfléchir à ce type de problématique lorsque nous développons une application : car plus il y a d'utilisateurs, plus le gaspillage de ressources va être susceptible de poser des problèmes que vous ne pouvez pas identifier par de simples tests momo-utilisateur. En d'autres termes, votre application peut très bien donner l'impression de fonctionner au poil et d'être parfaitement codée, jusqu'au jour où un nombre critique d'utilisateurs simultanés va faire surgir des problèmes imprévus.

Pour revenir à notre cas, il existe une portée qui se place entre la requête et la session, et qui semble donc parfaite pour ce que nous faisons ici : le scope de conversation. Il permet de conserver un objet tant qu'une même vue est utilisée par un même utilisateur. Cela signifie que tant qu'un utilisateur effectue des requêtes depuis une seule et même page, alors l'objet est conservé sur le serveur et réutilisé. Dès que l'utilisateur effectue une requête vers une autre vue, alors l'objet est finalement détruit. Dans notre cas, c'est parfait : notre formulaire provoque l'envoi de plusieurs requêtes vers le serveur pour la validation de chacun des champs, et toutes constituent un échange continu entre le serveur et une unique vue. Nous allons donc placer notre objet dans cette portée, pour limiter le gâchis de ressources sur le serveur.

Pour déclarer un **backing-bean** dans cette portée, il faut l'annoter avec `@ViewScoped`, et non plus avec `@RequestScoped`. Voilà tout ce qu'il nous faut changer pour optimiser notre application !



À titre d'exercice, et si vous avez déjà parcouru l'annexe sur le déboggage de projet avec Eclipse, je vous encourage à utiliser le mode debug pour vérifier de vos propres yeux la conservation d'une instance du backing-bean d'une requête à l'autre avec le scope de conversation, et sa destruction avec le scope de requête.

Une inscription contrôlée

Déporter la validation de la vue vers l'entité

Nous nous sommes jusqu'à présent contentés d'effectuer une simple vérification sur les champs du formulaire, directement dans la vue grâce aux attributs **required** et **requiredMessage**. Nous allons déplacer ces conditions de la vue vers notre modèle, à savoir notre entité, grâce à de simples annotations ! Le serveur GlassFish fournit par défaut un moyen de validation, identifié sous le nom de **JSR 303** : il contient différentes contraintes de validation sous forme d'annotations, par exemple `@NotNull` qui permet d'indiquer qu'une propriété ne peut être laissée vide.

Première étape, nous allons donc supprimer de notre Facelet les quelques **required** et **requiredMessage** présents sur les champs de notre formulaire, et que nous utilisions justement pour indiquer que nos champs ne pouvaient pas être laissés vides. Il nous suffit ensuite d'éditer notre entité **Utilisateur** et d'y ajouter les annotations sur les attributs correspondants pour remettre en place ces contraintes. Voici le code de notre entité reprise et complétée (j'omets ici les méthodes *getters/setters* pour ne pas encombrer le code inutilement) :

Code : Java - com.sdzee.entities.Utilisateur

```
package com.sdzee.entities;

import java.sql.Timestamp;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.validation.constraints.NotNull;

@Entity
public class Utilisateur {

    @Id
    @GeneratedValue( strategy = GenerationType.IDENTITY )
    private Long id;
    @NotNull
    private String email;
    @Column( name = "mot_de_passe" )
    @NotNull
    private String motDePasse;
    @NotNull
    private String nom;
    @Column( name = "date_inscription" )
    private Timestamp dateInscription;
```

```
    ...
}
```

Les seuls ajouts effectués sont les trois annotations `@NotNull`, issues du package `javax.validation.constraints`.



Par contre, comprenez bien que ceci ne fonctionnera que si vous avez suivi mon conseil en début de chapitre, et ajouté `javax.faces.INTERPRET_EMPTY_STRING_SUBMITTED_VALUES_AS_NULL` en paramètre de contexte. Sans cette configuration, les champs du formulaire laissés vides seraient traités comme des chaînes vides par votre serveur, et par conséquent celles-ci ne seraient pas détectées comme nulles par ces annotations fraîchement mises en place dans l'entité.

Le seul cas où vous pourriez vous passer de cette configuration se présenterait si vous utilisiez Hibernate en guise de framework ORM, et que vous utilisiez l'annotation spécifique à Hibernate intitulée `@NotEmpty`. Cependant comme je vous l'ai déjà expliqué, si vous commencez à mettre du code spécifique à Hibernate dans votre application, vous ne pourrez plus revenir en arrière, c'est-à-dire changer d'implémentation de JPA, sans modifier votre code...

Une fois ces légères modifications effectuées, ouvrez à nouveau la page d'inscription dans votre navigateur, et tentez une nouvelle fois de vous inscrire en laissant des champs vides. Vous constaterez alors que les contraintes fonctionnent bien, mais que les messages de validation sont une nouvelle fois trop génériques (voir la figure suivante).

Pas d'inquiétude, nous allons pouvoir les personnaliser directement depuis notre entité en complétant nos annotations :

Code : Java - com.sdzee.entities.Utilisateur

```
...
@NotNull( message = "Veuillez saisir une adresse email" )
private String email;
@Column( name = "mot_de_passe" )
@NotNull( message = "Veuillez saisir un mot de passe" )
private String motDePasse;
@NotNull( message = "Veuillez saisir un nom d'utilisateur" )
private String nom;
```

S

Il suffit comme vous pouvez l'observer dans cet exemple de préciser entre parenthèses un attribut `message` à l'annotation `@NotNull`.

Nous voilà de retour à une situation similaire à celle que nous observions lorsque nous effectuions la validation directement depuis notre vue (voir la figure suivante).

Inscription

Adresse email *	<input type="text" value="coyote@bipbip.fr"/>	Veuillez saisir un mot de passe
Mot de passe *	<input type="password"/>	Veuillez saisir un mot de passe
Confirmation du mot de passe *	<input type="password"/>	Veuillez saisir un mot de passe
Nom d'utilisateur *	<input type="text"/>	Veuillez saisir un nom d'utilisateur

Inscription



Quelle est la méthode de validation à préférer : depuis la vue, ou depuis l'entité ?

Cela dépend principalement des contraintes du projet. Par exemple, si l'application doit pouvoir fonctionner sur un serveur léger comme Tomcat sans support des EJB ni de JPA, alors il ne faudra pas utiliser la validation depuis l'entité mais lui préférer la validation depuis la vue JSF.

Affiner les contrôles effectués

Allons un petit peu plus loin, et essayons de nous rapprocher davantage du fonctionnement de notre formulaire lorsque nous utilisons une page JSP et notre objet métier fait maison.

Nous souhaitons affiner la validation des différents champs de notre formulaire. Dans notre ancien objet métier, nous procédions aux vérifications suivantes :

- que l'adresse respecte bien le format standard d'une adresse email ;
- que le mot de passe soit long de 3 caractères ou plus ;
- que le nom soit long de 3 caractères ou plus.

Le hasard fait encore une fois bien les choses : l'annotation `@Pattern` fournie par la [JSR 303](#) est parfaite pour vérifier le format de l'adresse grâce à l'utilisation d'expressions régulières, et l'annotation `@Size` est parfaite pour vérifier la taille des champs ! Voici le code complété :

Code : Java - com.sdzee.entities.Utilisateur

```

...
@NotNull( message = "Veuillez saisir une adresse email" )
@Pattern( regexp = "([^.@]+)(\\.[^.@]+)*@([^.@]+\\\.)([^^.@]+)" ,
message = "Merci de saisir une adresse mail valide" )
private String email;
@Column( name = "mot_de_passe" )
@NotNull( message = "Veuillez saisir un mot de passe" )
@Size( min = 3, message = "Le mot de passe doit contenir au moins 3
caractères" )
private String motDePasse;
@NotNull( message = "Veuillez saisir un nom d'utilisateur" )
@Size( min = 3, message = "Le nom d'utilisateur doit contenir au
moins 3 caractères" )
private String nom;
...

```

Les documentations respectives des deux annotations expliquent que l'attribut `regexp` permet de définir une expression régulière qui sera appliquée au champ ciblé par l'annotation `@Pattern`, et que l'attribut `min` permet de définir la taille minimale autorisée pour le champ ciblé par l'annotation `@Size`.

Effectuez ces ajouts, puis tentez à nouveau de vous inscrire en saisissant un nom ou un mot de passe trop court, et une adresse email dont le format est incorrect.



Rendez-vous bien compte de la simplicité avec laquelle nous avons mis en place ces quelques vérifications : de simples annotations placées sur les champs de l'entité suffisent ! Alors qu'auparavant, nous avions dû écrire pas loin d'une centaine de lignes de code dans notre objet métier, rien que pour mettre en place ces vérifications et gérer les exceptions proprement...

Par ailleurs, nous n'avons pas mis en place ce type de contrôle dans notre application, mais nous pourrions tout à fait vérifier que le mot de passe fourni par l'utilisateur présente un niveau de sécurité suffisamment élevé. Pour ce faire, l'annotation `@Pattern` se révèle une nouvelle fois très utile. Voici par exemple le code à mettre en place afin de s'assurer que le mot de passe entré par l'utilisateur contient au moins 8 caractères, dont au moins un chiffre, une lettre minuscule et une lettre majuscule :

Code : Java - Vérification du niveau de sécurité du mot de passe saisi

```
@NotNull( message = "Veuillez saisir un mot de passe" )
@Pattern(regexp = ".*(?=.{8,})(?=.*\d)(?=.*[a-z])(?=.*[A-Z]).*", message = "Le mot de passe saisi n'est pas assez sécurisé")
private String motDePasse;
```

Simple et rapide, n'est-ce pas ? La puissance des expressions régulières associée à la simplicité des annotations, c'est tout simplement magique ! Alors qu'il nous aurait fallu écrire une barbante méthode de traitement supplémentaire et l'exception associée dans notre ancien objet métier, nous pouvons désormais nous contenter d'une courte annotation sur le champ à valider, directement dans notre entité !

Ajouter des contrôles "métier"

Jusqu'à présent, nous avons réussi à appliquer des contrôles sur le format des données de manière étonnamment simple. Cela dit, nous avons lâchement évité deux contrôles qui étaient pourtant en place dans notre ancien objet métier, et qui sont nécessaires pour réaliser une inscription valide :

- le mot de passe et sa confirmation doivent être égaux ;
- l'adresse email ne doit pas exister dans la base de données.

Ces contraintes se distinguent des simples vérifications de format que nous avons mises en place jusqu'à présent, car elles ont trait à l'aspect métier de notre application. Nous devons donc impérativement trouver un moyen propre de mettre en place ces contrôles dans notre nouvelle architecture.

Comme toujours, JSF propose un outil dédié à notre besoin. Le *framework* fournit une interface nommée `javax.faces.validator.Validator`, qui permet de créer une classe contenant une méthode de validation, qui pourra alors être liée à un composant très simplement depuis la vue, via un attribut placé dans une balise.

Nous allons pour commencer mettre en place la vérification de l'existence de l'adresse email dans la base de données. Pour ce faire, nous devons créer un nouvel objet implémentant cette interface `Validator`, que nous allons nommer `ExistenceEmailValidator` et placer dans un nouveau package `com.sdzee.validators` :

Code : Java - com.sdzee.validators.ExistenceEmailValidator

```
package com.sdzee.validators;

import javax.faces.application.FacesMessage;
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.validator.Validator;
import javax.faces.validator.ValidatorException;

public class ExistenceEmailValidator implements Validator {

    @Override
    public void validate( FacesContext context, UIComponent
```

```
component, Object value ) throws ValidatorException {  
    ...  
}  
}
```

La seule méthode qu'il est nécessaire de surcharger est la méthode **validate()** : c'est elle qui va contenir le code métier chargé d'effectuer le contrôle de l'existence de l'adresse dans la base. Nous savons déjà comment réaliser cette tâche, nous l'avions déjà fait dans notre ancien objet métier. Il nous suffit donc d'adapter le code que nous avions alors écrit. Voici un exemple de solution :

Code : Java - com.sdzee.validators.ExistenceEmailValidator

```
package com.sdzee.validators;

import javax.ejb.EJB;
import javax.faces.application.FacesMessage;
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.validator.Validator;
import javax.faces.validator.ValidatorException;

import com.sdzee.dao.DAOException;
import com.sdzee.dao.UtilisateurDao;

public class ExistenceEmailValidator implements Validator {

    private static final String EMAIL_EXISTE_DEJA = "Cette adresse
email est déjà utilisée";

    @EJB
    private UtilisateurDao utilisateurDao;

    @Override
    public void validate( FacesContext context, UIComponent
component, Object value ) throws ValidatorException {
        /* Récupération de la valeur à traiter depuis le paramètre
value */
        String email = (String) value;
        try {
            if ( utilisateurDao.trouver( email ) != null ) {
                /*
                * Si une adresse est retournée, alors on envoie une exception
                * propre à JSF, qu'on initialise avec un FacesMessage de
                * gravité "Erreur" et contenant le message d'explication. Le
                * framework va alors gérer lui-même cette exception et s'en
                * servir pour afficher le message d'erreur à l'utilisateur.
                */
                throw new ValidatorException(
                    new FacesMessage(
                        FacesMessage.SEVERITY_ERROR, EMAIL_EXISTE_DEJA, null ) );
            }
        } catch ( DAOException e ) {
            /*
            * En cas d'erreur imprévue émanant de la BDD, on prépare un message
            * d'erreur contenant l'exception retournée, pour l'afficher à
            * l'utilisateur ensuite.
            */
            FacesMessage message = new FacesMessage(
                FacesMessage.SEVERITY_ERROR, e.getMessage(), null );
            FacesContext facesContext =
                FacesContext.getCurrentInstance();
            facesContext.addMessage( component.getClientId(
                facesContext ), message );
        }
    }
}
```

Vous devriez comprendre sans problème avec les commentaires présents dans le code. Vous découvrez ici de nouveaux objets du framework JSF, comme **UIComponent** ou **ValidatorException**, et en retrouvez d'autres que vous connaissez déjà comme **FacesContext** et **FacesMessage**. N'hésitez pas à parcourir leur documentation, via Eclipse ou dans la Javadoc en ligne directement depuis votre navigateur, si vous souhaitez en apprendre davantage sur les constructeurs et méthodes ici utilisés.

Vous retrouvez en outre l'injection de notre EJB Stateless, le DAO Utilisateur, dans notre objet via l'annotation JPA **@EJB**.

Maintenant que notre objet est prêt, il nous faut le déclarer auprès de JSF afin qu'il le rende accessible à notre vue. Oui, parce que pour le moment c'est bien gentil d'avoir codé un objet implémentant l'interface **Validator**, mais encore faut-il que notre vue puisse s'en servir pour l'appliquer au champ de saisie de l'adresse email ! Comme d'habitude, les annotations sont là pour nous sauver, et JSF propose l'annotation **@FacesValidator**. Celle-ci permet de déclarer auprès du framework un objet comme étant un **Validator**, et permet ainsi de rendre cet objet accessible depuis une balise dans nos Facelets.

Malheureusement, nous n'allons pas pouvoir utiliser cette annotation... Pourquoi ? Eh bien il s'agit là d'un comportement étrange de JSF : les objets annotés avec **@FacesValidator** ne sont pas pris en charge par le conteneur d'injection.



Qu'est-ce que cela veut dire, concrètement ?

Formulé autrement, cela signifie qu'il n'est pas possible d'injecter un EJB avec l'annotation **@EJB** dans un **Validator** JSF annoté avec **@FacesValidator**. C'est un problème, car nous faisons usage de notre DAO Utilisateur dans notre validateur, et puisque nous travaillons avec JPA nous avons besoin de pouvoir y injecter cet EJB.



D'après les informations qui circulent sur la toile, il semblerait que la communauté des développeurs chargés du maintien de JSF et de JPA travaille sur ce point, et qu'une correction soit prévue pour la prochaine version à venir de JSF (JSF 2.2, pas encore sorti lors de la rédaction de ce cours).

Il nous faut donc contourner cette limitation nous-mêmes. Il existe plusieurs moyens : nous pouvons conserver l'annotation **@FacesValidator** et récupérer manuellement notre EJB depuis notre validateur, ou encore laisser tomber l'annotation et nous débrouiller autrement. Nous allons opter pour la seconde méthode, et en ce qui concerne la première je vous renvoie vers cet excellent article pour plus de détails.

Nous n'allons donc pas utiliser cette annotation. À la place, nous allons déclarer notre objet comme un simple **Backing-bean** ! Ainsi, notre vue pourra y accéder, c'est le principe même du **Backing-bean**, et nous pourrons y injecter notre EJB sans problème. Nous devons donc ajouter les annotations suivantes à notre objet :

Code : Java - Annotations de notre validateur

```
...
@ManagedBean
@RequestScoped
public class ExistenceEmailValidator implements Validator {
    ...
}
```

Au sujet de la portée requête ici utilisée, puisque notre validateur ne sera utilisé que pour valider un champ, nous pouvons utiliser la portée par défaut sans souci.

Maintenant que notre objet est déclaré et donc accessible depuis la vue, nous devons ajouter une balise à notre Facelet pour qu'elle fasse appel à ce validateur pour le champ de saisie de l'adresse email. Pour ce faire, nous allons utiliser la balise **<f:validator>**. En regardant sa documentation, nous découvrons qu'elle possède un attribut **validatorId** permettant de préciser quel objet utilisé en tant que validateur. Oui mais voilà, cet attribut ne fonctionne que si notre objet est annoté avec **@FacesValidator**, et pour des raisons techniques évoquées un peu plus tôt nous avons décidé de ne pas utiliser cette annotation...

Heureusement, la balise propose un autre attribut intitulé **binding**, qui permet de préciser un **Backing-bean** à utiliser en tant que validateur, à la condition que cet objet implémente l'interface **Validator**. Bingo ! C'est exactement notre cas, et il ne nous reste donc plus qu'à ajouter cette balise dans notre Facelet. Le code responsable du champ de saisie de l'adresse email va donc devenir :

Code : HTML - Ajout du validateur sur le champ de saisie de l'adresse email

```
<h:inputText id="email" value="#{inscrireBean.utilisateur.email}" size="20" maxlength="60">
    <f:ajax event="blur" render="emailMessage" />
    <f:validator binding="#{existenceEmailValidator}" />
</h:inputText>
```

Sans surprise, nous utilisons une expression EL pour cibler notre **Backing-bean**, comme nous l'avons déjà fait pour **inscriptionBean** dans les autres balises. Sa dénomination est par défaut, je vous le rappelle, le nom de la classe de l'objet débutant par une minuscule, à savoir **existenceEmailValidator** dans notre cas.

Une fois cette modification effectuée, rendez-vous à nouveau sur la page d'inscription depuis votre navigateur, et essayez de vous inscrire avec une adresse email qui existe déjà dans votre base de données, comme indiqué sur la figure suivante.

Vous constatez alors que la validation est effectuée comme prévu : en plus des contrôles sur le format de l'adresse email, votre application vérifie maintenant que l'adresse saisie n'existe pas déjà dans la base de données, et affiche un message d'erreur le cas échéant ! Par ailleurs, vous remarquerez que puisque nous avons donné le niveau `FacesMessage.SEVERITY_ERROR` à notre message d'erreur depuis la méthode de validation, le message est bien considéré comme tel par JSF et est coloré en rouge (la classe CSS **erreur** lui est appliquée).

Il nous reste encore un contrôle à effectuer avant d'en avoir terminé avec notre formulaire : vérifier que le mot de passe et la confirmation saisis sont égaux. De la même manière que précédemment, nous allons créer un validateur dédié à cette tâche. Seulement cette fois-ci, nous n'allons pas avoir besoin d'injecter un EJB dans notre objet. En effet, pour vérifier que les deux champs sont égaux, nous n'avons absolument pas besoin de faire appel à notre DAO Utilisateur. Nous allons donc pouvoir utiliser l'annotation `@FacesValidator` pour lier notre validateur à la vue :

Code : Java - com.sdzee.validators ConfirmationMotDePasseValidator

```
package com.sdzee.validators;

import javax.faces.component.UIComponent;
import javax.faces.component.UIInput;
import javax.faces.context.FacesContext;
import javax.faces.validator.FacesValidator;
import javax.faces.validator.Validator;
import javax.faces.validator.ValidatorException;

@FacesValidator( value = "confirmationMotDePasseValidator" )
public class ConfirmationMotDePasseValidator implements Validator {

    @Override
    public void validate( FacesContext context, UIComponent component, Object value ) throws ValidatorException {
        ...
    }
}
```

Celle-ci attend en paramètre le nom qui sera utilisé comme identifiant depuis la vue pour désigner ce validateur : nous lui donnons ici le même nom que la classe, avec la première lettre en minuscule pour respecter la règle à suivre pour les **Backing-**

bean et avoir des dénominations homogènes dans notre Facelet.

Voilà tout pour la forme. Le plus important maintenant, c'est de trouver un moyen de comparer les contenus de deux champs différents. Car c'est bien cela que nous cherchons à faire : comparer le contenu du champ de saisie du mot de passe avec celui de la confirmation. Pour ce faire, nous allons commencer par regarder comment procéder depuis notre Facelet, avant de revenir sur notre objet et de coder la méthode de validation.

Commençons tout simplement par lier notre validateur, même s'il n'est pas encore terminé, au composant en charge de la confirmation :

Code : HTML

```
<h:inputSecret id="confirmation"
value="#{inscrireBean.utilisateur.motDePasse}" size="20"
maxlength="20">
    <f:ajax event="blur" render="confirmationMessage" />
    <f:validator validatorId="confirmationMotDePasseValidator" />
</h:inputSecret>
```

Puisque cette fois nous avons pu nous servir de l'annotation `@FacesValidator`, nous pouvons donc utiliser l'attribut `validatorId` de la balise `<f:validator>`. Celle-ci, à la différence de l'attribut `binding`, n'attend pas une expression EL mais directement le nom du validateur. Il s'agit de celui que nous avons défini en tant que paramètre de l'annotation dans notre validateur, en l'occurrence `confirmationMotDePasseValidator`. En mettant en place cette balise, nous nous assurons ainsi que notre validateur est associé au composant en charge du champ de confirmation.

La prochaine étape va consister à déclarer le composant en charge du champ de mot de passe **en tant qu'attribut du composant** en charge de la confirmation. L'intérêt de réaliser cette association est de rendre disponible le contenu du champ de mot de passe depuis le composant en charge du champ de confirmation. Ne vous embrouillez pas trop pour le moment, regardez simplement comment procéder :

Code : HTML

```
<h:inputSecret id="confirmation"
value="#{inscrireBean.utilisateur.motDePasse}" size="20"
maxlength="20">
    <f:ajax event="blur" render="confirmationMessage" />
    <f:attribute name="composantMotDePasse"
value="#{composantMotDePasse}" />
    <f:validator validatorId="confirmationMotDePasseValidator" />
</h:inputSecret>
```

Nous utilisons la balise `<f:attribute>` pour mettre en place l'association. Sa propriété `name` permet de définir le nom de l'objet qui va être créé en tant qu'attribut au composant courant, et sa propriété `value` permet de définir son contenu. Nous avons donc nommé l'objet `composantMotDePasse`, et avons lié la propriété `value` directement avec sa valeur en utilisant l'expression EL `# {composantMotDePasse}`.

Cette déclaration permet donc de créer un objet représentant le champ mot de passe en tant qu'attribut du composant de confirmation, mais il faut encore faire en sorte que la valeur du champ mot de passe soit bien affectée à la valeur de cet objet. Pour ce faire, nous allons modifier la déclaration du composant en charge du mot de passe de cette manière :

Code : HTML

```
<h:inputSecret id="motdepasse"
value="#{inscrireBean.utilisateur.motDePasse}"
binding="#{composantMotDePasse}" size="20" maxlength="20">
    <f:ajax event="blur" render="motDePasseMessage" />
</h:inputSecret>
```

En ajoutant un attribut `binding` et en le liant à la valeur de l'objet via l'expression EL `# {composantMotDePasse}`, nous nous assurons alors que la valeur saisie dans le champ de mot de passe sera affectée à l'objet déclaré en tant qu'attribut du composant de confirmation !

La dernière étape consiste maintenant à accorder nos violons. Actuellement, nos validations AJAX déclenchent le rendu de leur champ respectif uniquement. Maintenant que nous vérifions l'égalité entre les deux champs, pour rendre l'expérience utilisateur plus intuitive et logique, il faut faire en sorte que la validation AJAX de l'un entraîne la validation de l'autre, et vice-versa. Pour ce faire, nous devons modifier les balises `<f:ajax>` que nous avions mises en place, et le code final devient alors :

Code : HTML - Vérification de l'égalité des mots de passe

```
...
<h:inputSecret id="motdepasse"
value="#{inscrireBean.utilisateur.motDePasse}"
binding="#{composantMotDePasse}" size="20" maxlength="20">
    <f:ajax event="blur" execute="motdepasse confirmation"
render="motDePasseMessage confirmationMessage" />
</h:inputSecret>

...
<h:inputSecret id="confirmation"
value="#{inscrireBean.utilisateur.motDePasse}" size="20"
maxlength="20">
    <f:validator validatorId="confirmationMotDePasseValidator" />
    <f:attribute name="composantMotDePasse"
value="#{composantMotDePasse}" />
    <f:ajax event="blur" execute="motdepasse confirmation"
render="motDePasseMessage confirmationMessage" />
</h:inputSecret>

...
```

Seules les balises `<f:ajax>` ont changé. Deux choses importantes à remarquer :

- l'ajout effectué dans les attributs **render**. Avec cette configuration, l'affichage du message associé au mot de passe ET de celui associé à la confirmation sera actualisé quoi qu'il arrive. Pour information, mais vous pouvez trouver cela vous-mêmes dans la documentation de la balise `<f:ajax>`, il suffit de séparer les composants à actualiser par un espace.
- l'utilisation de l'attribut **execute**. Je vous ai déjà expliqué qu'il permet de définir la portée de l'action réalisée. C'est exactement ce dont nous avons besoin ici : nous voulons que lorsque l'événement **blur** est déclenché, à la fois le composant mot de passe et le composant confirmation soient traités. Voilà pourquoi nous y précisons les identifiants de nos deux champs de saisie.

Notre Facelet est enfin terminée, il ne nous reste maintenant plus qu'à coder la méthode de validation, dans notre Validator associé au champ de confirmation. Le petit challenge qui nous attend, c'est de trouver un moyen pour récupérer le contenu du champ mot de passe depuis le composant traité par la méthode, c'est-à-dire le composant de confirmation.

En regardant la documentation de l'objet `UIComponent`, nous y trouvons la méthode `getAttributes()` qui permet de récupérer une liste des attributs du composant. C'est justement ce que nous cherchons à faire ! Eh oui, rappelez-vous : nous avons déclaré le composant mot de passe comme attribut du composant confirmation dans notre Facelet, par le biais de la balise `<f:attribute>`. Nous apprenons que les éléments de cette liste sont accessibles par leur nom, il va donc nous suffire de cibler l'élément nommé `composantMotDePasse` et le tour est joué ! Voici le code final de notre validateur :

Code : Java - com.sdzee.validators.ConfirmationMotDePasseValidator

```
package com.sdzee.validators;

import javax.faces.application.FacesMessage;
import javax.faces.component.UIComponent;
import javax.faces.component.UIInput;
import javax.faces.context.FacesContext;
import javax.faces.validator.FacesValidator;
import javax.faces.validator.Validator;
import javax.faces.validator.ValidatorException;

@FacesValidator( value = "confirmationMotDePasseValidator" )
```

```

public class ConfirmationMotDePasseValidator implements Validator {

    private static final String CHAMP_MOT_DE_PASSE =
"composantMotDePasse";
    private static final String MOTS_DE_PASSE_DIFFERENTS = "Le mot
de passe et la confirmation doivent être identiques.';

    @Override
    public void validate( FacesContext context, UIComponent
component, Object value ) throws ValidatorException {
        /*
        * Récupération de l'attribut mot de passe parmi la liste des
attributs
        * du composant confirmation
        */
        UIInput composantMotDePasse = (UIInput)
component.getAttributes().get( CHAMP_MOT_DE_PASSE );
        /*
        * Récupération de la valeur du champ, c'est-à-dire le mot de passe
        * saisi
        */
        String motDePasse = (String) composantMotDePasse.getValue();
        /* Récupération de la valeur du champ confirmation */
        String confirmation = (String) value;

        if ( confirmation != null && !confirmation.equals(
motDePasse ) ) {
            /*
            * Envoi d'une exception contenant une erreur de validation JSF
            * initialisée avec le message destiné à l'utilisateur, si les mots
            * de passe sont différents
            */
            throw new ValidatorException(
                new FacesMessage( FacesMessage.SEVERITY_ERROR,
MOTS_DE_PASSE_DIFFERENTS, null ) );
        }
    }
}

```

La seule petite difficulté ici est de penser à convertir l'objet récupéré depuis la liste des attributs du composant confirmation, à la ligne 23, en tant que `UIInput`, afin de pouvoir récupérer ensuite simplement sa valeur avec la méthode `getValue()`. Quant au reste de la méthode, c'est le même principe que dans le validateur de l'adresse email que nous avons codé un peu plus tôt.

Une fois ces modifications effectuées dans le code de votre projet, rendez-vous une ultime fois sur la page d'inscription depuis votre navigateur. Tentez alors de rentrer des mots de passe différents, égaux, de rentrer la confirmation avant le mot de passe, de rentrer des mots de passe égaux puis d'en modifier un des deux, etc. Vous observerez alors, si vous n'avez rien oublié, une actualisation en direct des messages d'erreur associés aux champs mot de passe et confirmation !

Nous en avons enfin terminé avec notre formulaire d'inscription ! Tout cela a dû vous paraître bien long, mais c'est avant tout parce que nous avons pris le temps de décortiquer toutes les nouveautés qui interviennent. Si vous prenez un peu de recul, et que vous regardez le code final de votre application, vous remarquerez alors qu'il est bien moins volumineux et bien plus organisé que lorsque nous travauillons sur MVC à la main. Mission réussie pour le tandem JSF + JPA : la gestion des formulaires est maintenant une partie de plaisir, et seules les vérifications métier nécessitent encore l'écriture de méthodes de validation !

- Grâce à notre travail effectué avec JPA, la couche de données est totalement indépendante du reste de l'application.
- Pour gérer correctement les champs de formulaire laissés vides par l'utilisateur, il est recommandé de configurer le paramètre `javax.faces.INTERPRET_EMPTY_STRING_SUBMITTED_VALUES_AS_NULL` dans le web.xml du projet.
- Pour gérer un formulaire simple, il suffit d'une Facelet, d'un backing-bean et d'une entité.
- Le backing-bean contient une référence à l'entité, et une ou plusieurs méthodes d'actions appelées lors de la soumission du formulaire associé.
- La portée du backing-bean ne doit pas être définie à la légère, il est important d'utiliser la plus petite portée possible pour limiter le gâchis de ressources sur le serveur.
- La méthode `FacesContext.addMessage()` permet de définir un message géré par JSF, et de lui attribuer un niveau de gravité.
- La Facelet contient des balises qui représentent les composants JSF, et qui sont liées aux propriétés de l'entité par l'intermédiaire du backing-bean, à travers des expressions EL.

- La validation du format des données saisies est entièrement prise en charge par JSF, et il est possible de l'effectuer :
 - depuis la vue, via des attributs et balises dédiées.
 - depuis le modèle, via des annotations sur les propriétés des entités.
- Les contrôles métier se font par le biais d'un objet `Validator`, dont la création est simple et guidée par le *framework*.
- L'ajaxisation de la validation des champs d'un formulaire avec JSF est incroyablement simple.
- Avec le tandem JPA et JSF, nous pouvons construire un formulaire efficace sans SQL et sans JavaScript.

L'envoi de fichiers avec JSF

Dans cet ultime chapitre, nous allons découvrir comment gérer l'envoi de fichiers depuis un formulaire avec JSF. Ce sujet va en réalité nous servir de prétexte pour découvrir les fameuses bibliothèques de composants JSF ! En guise de clôture, je vous donnerai ensuite des indications pour réaliser la huitième et dernière étape du fil rouge, et nous ferons enfin un rapide point sur ce qui vous attend dans le monde du développement web avec Java EE !

Le problème

 « J'ai beau chercher dans la documentation, je ne trouve aucune balise qui ressemble de près ou de loin à quelque chose qui s'occupe des fichiers ! »

Ne perdez pas davantage votre temps, aucun composant standard n'existe actuellement dans JSF pour générer une balise `<input type="file">`, ni pour gérer correctement les requêtes de types **multipart** impliquées par les transferts de données occasionnés.

Cependant, c'est au [programme de la version JSF 2.2](#), prévue pour fin mars 2013 au moment de l'écriture de ce cours. Une balise standard `<h:inputFile>` va faire son apparition, et apporter notamment un support AJAX pour l'envoi de fichiers ! Mais en attendant cette nouveauté, nous devons trouver un autre moyen...

 Au passage, si vous parcourez ce programme vous apprendrez que cette nouvelle mouture de JSF apportera enfin le support de l'injection via l'annotation `@EJB` dans tous les objets JSF, problème auquel nous nous étions heurtés dans le chapitre précédent. Vous pouvez par ailleurs noter que le support des requêtes de type GET et des éléments HTML5 est également annoncé.

 « Avec JSF, il est possible de définir ses propres composants. Ne peut-on pas en créer un pour l'occasion ? »

C'est une très bonne idée sur le papier, et effectivement JSF permet la création de composants personnalisés. Cependant rien que pour un simple et unique composant, c'est une charge de travail assez conséquente. Afin de mener à bien un tel objectif, il est notamment nécessaire de connaître les concepts de `FacesRenderer` et de `FacesComponents`. Autrement dit, il faut mettre les mains dans le code d'une bibliothèque ou d'une balise existante pour comprendre comment c'est fait, et en déduire comment l'adapter pour notre besoin. C'est un exercice très formateur je vous l'accorde, et je vous encourage à vous y atteler (voir la bulle d'information ci-après), mais dans le cadre de notre cours c'est trop d'efforts pour ce que nous souhaitons faire.

En outre, un autre obstacle se dresse sur notre chemin : par défaut, JSF et sa `FacesServlet` n'embarquent rien qui permette de gérer les requêtes de type **multipart**. Qu'est-ce que cela signifie exactement ? Si vous vous souvenez bien, dans notre servlet d'upload « faite maison », nous avions fait intervenir une annotation particulière, nommée `@MultipartConfig`, afin de préciser au conteneur que notre servlet était équipée pour traiter des requêtes de ce type. Ensuite seulement, nous pouvions y utiliser les méthodes `request.getParts()`, etc.

Eh bien voilà ce qu'il manque actuellement à JSF : la `FacesServlet` n'est pas annotée avec `@MultipartConfig`. Elle n'est capable dans les coulisses que d'utiliser les méthodes de récupération de paramètres traditionnelles, comme `request.getParameter()`. Par conséquent, il n'est pas possible de se baser directement sur elle pour réaliser un système d'envoi de fichiers.

Il existe bien entendu une parade. Il faudrait pour commencer réaliser tout le travail de conversion et de gestion des **Parts** à la main, comme nous l'avions fait dans notre servlet d'upload auparavant. Seulement cela n'est pas suffisant : pour intégrer ça proprement avec JSF, il faudrait créer un filtre, qui s'occupera de précharger les traitements nécessaires en amont de la `FacesServlet`, de manière à rendre disponibles les données d'une requête **multipart** à travers les méthodes traditionnelles que la `FacesServlet` sait utiliser.. Elle pourrait ainsi, comme si de rien n'était et peu importe le type de requête entrante (normale ou **multipart**), permettre de manipuler les données envoyées.

Bref, vous devez vous rendre compte que c'est un travail massif qui nous attend, difficile à faire d'une part, et encore plus difficile à faire proprement. Autant de raisons pour nous pousser à choisir une autre solution : utiliser un composant permettant l'envoi de fichiers, prêt à l'emploi et fourni... dans **une bibliothèque de composants externe** !

Pour information, un bienfaiteur très actif dans le développement et la promotion de JSF surnommé **BalusC**, a déjà :

- codé des filtres prêts à l'emploi pour la gestion des requêtes **multipart**, et les a rendu disponibles dans [ce premier article de blog](#) ;
- créé un composant personnalisé pour permettre la gestion de l'envoi de fichiers avec JSF, et expliqué la démarche sur [ce second article de blog](#).

C'est en anglais, mais je vous encourage à jeter un œil à ses codes et explications. C'est clair, propre et très formateur !

Les bibliothèques de composants

De manière générale, quand vous avez besoin de quelque chose qui semble assez commun, et que vous vous rendez compte que ça n'existe pas dans le standard JSF, vous devez prendre le réflexe de vous dire que quelqu'un a déjà dû se poser la question avant vous, que quelqu'un y a probablement déjà apporté une réponse, et qu'il existe sûrement une ou plusieurs manières de résoudre votre problème.



Ça paraît bête à première vue, mais nombreux sont les développeurs qui ont tôt fait de l'oublier et qui s'épuisent à réinventer la roue avant, bien souvent, de se rendre compte du temps qu'ils ont perdu !

D'ailleurs si vous vous souvenez du charabia que je vous avais raconté en introduisant le *framework* JSF, vous savez que nous avons déjà parlé de **PrimeFaces**, **RichFaces**, **IceFaces**, etc. Ces fameuses bibliothèques de composants évolués et arborant des sites vitrines parfois époustouflants. Vous pensez vraiment que des solutions capables de proposer des fonctionnalités avancées comme du drag & drop, de la génération de graphiques ou du dessin vectoriel en temps réel clé en mains, ne permettent pas de gérer l'envoi de fichiers ? Vous pensez bien, c'est un des premiers travaux qu'elles ont pour la plupart entrepris ! Ainsi, si vous cherchez un peu sur la toile vous tomberez notamment sur :

- PrimeFaces, qui offre la balise `<p:fileUpload>` ;
- RichFaces, qui offre la balise `<rich:fileUpload>` ;
- IceFaces, qui offre la balise `<ice:inputFile>`.

Ce sont là les trois mastodontes, les trois bibliothèques de composants JSF les plus fournies, les plus reconnues, les plus utilisées et les plus maintenues.

Seulement ils sont un peu massifs, ces mastodontes. Et sortir une telle machinerie (qui se matérialise dans votre projet par une flopée d'archives jar à placer dans le dossier **/WEB-INF/lib**, et qui alourdit donc votre application) simplement pour réaliser de l'envoi de fichiers, c'est un peu sortir un tank pour chasser une mouche.

À côté de ça, il existe des projets un peu moins ambitieux, mais tout aussi utiles comme le célèbre **Tomahawk**, qui offre la balise `<t:inputFileUpload>`. Nous allons donc nous pencher sur cette solution "légère", si tant est que l'on puisse vraiment la qualifier ainsi, et nous en servir pour mettre en place l'envoi de fichiers via un formulaire.

L'envoi de fichier avec Tomahawk

Préparation du projet

Récupérez Tomahawk en choisissant sur [cette page de téléchargement](#) la dernière version disponible, dans le format qui vous convient (archive .zip ou .tar.gz). Le fichier se nomme **MyFaces Tomahawk 1.1.14 for JSF 2.0**, le numéro de version pouvant changer si une nouvelle version existe lorsque vous lisez ce cours.

Décompressez ensuite le contenu de l'archive sur votre poste, vous obtiendrez alors un dossier portant le même nom que l'archive. Dans mon cas, ce dossier s'intitule **/tomahawk20-1.1.14**. Dans ce dossier se trouve un répertoire intitulé **/lib**, qui contient à son tour 18 archives jar ! Ce sont tous ces fichiers que vous allez devoir copier dans le répertoire **/WEB-INF/lib** de votre projet afin d'y intégrer Tomahawk.

Comme je vous l'ai déjà expliqué, il est nécessaire de mettre en place un filtre afin de rendre la gestion des requêtes **multipart** possible. Pour notre plus grand bonheur, Tomahawk en fournit un prêt à l'emploi, qu'il nous suffit de déclarer dans le fichier web.xml de notre application :

Code : XML - Déclaration du filtre multipart de Tomahawk dans le fichier web.xml

```
<filter>
    <filter-name>MyFacesExtensionsFilter</filter-name>
    <filter-
        class>org.apache.myfaces.webapp.filter.ExtensionsFilter</filter-
        class>
    </filter>
    <filter-mapping>
        <filter-name>MyFacesExtensionsFilter</filter-name>
        <servlet-name>Faces Servlet</servlet-name>
    </filter-mapping>
```

Notez bien que le nom précisé dans le champ <**servlet-name**> doit absolument être celui que vous avez donné à votre FacesServlet. Si vous avez repris à la lettre le fichier web.xml que je vous ai proposé lors de notre premier projet JSF, alors vous pouvez également reprendre cette configuration telle quelle.

Création de la Facelet

Créez ensuite une Facelet d'upload, en prenant exemple sur la page JSP que nous avions créée à cet égard lorsque nous avions découvert l'envoi de fichiers via une servlet :

Code : HTML - Facelet d'upload

```

<!DOCTYPE html>
<html lang="fr"
      xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:t="http://myfaces.apache.org/tomahawk">
    <h:head>
      <title>Tomahawk - Envoi de fichier</title>
      <h:outputStylesheet library="default" name="css/form.css" />
    </h:head>
    <h:body>
      <h:form enctype="multipart/form-data">
        <fieldset>
          <legend>Envoi de fichier</legend>

          <h:outputLabel for="description">Description du
          fichier</h:outputLabel>
          <h:inputText id="description"
          value="#{uploadBean.fichier.description}" />
          <h:message id="descriptionMessage" for="description"
          errorClass="erreur" />
          <br />

          <h:outputLabel for="fichier">Emplacement du fichier
          <span class="requis">*</span></h:outputLabel>
          <t:inputFileUpload id="fichier"
          value="#{uploadBean.fichier.contenu}" />
          <h:message id="fichierMessage" for="fichier"
          errorClass="erreur" />
          <br />

          <h:messages globalOnly="true" infoClass="info" />

          <h:commandButton value="Envoyer"
          action="#{uploadBean.envoyer}" styleClass="sansLabel"/>
          <br />
        </fieldset>
      </h:form>
    </h:body>
</html>
```

La seule nouveauté ici est l'utilisation du composant <**t:inputFileUpload**> de la bibliothèque Tomahawk à la ligne 22, que nous déclarons dans l'en-tête <**html**> à la ligne 6 de la même manière que les bibliothèques de composants standard de JSF.

En ce qui concerne les trois expressions EL employées aux lignes 17, 22 et 28, elles vous donnent une idée de l'architecture que nous allons mettre en place derrière cette Facelet : un backing-bean nommé **UploadBean**, qui contient un JavaBean intitulé **fichier** en tant que propriété, qui à son tour contient deux propriétés nommées **description** et **contenu**. Le backing-bean contient également une méthode **envoyer()**, appelée lors du clic sur le bouton de validation du formulaire.



Vous n'oublierez pas de remarquer à la ligne 12 l'ajout de l'attribut **enctype="multipart/form-data"** à la balise <**h:form**>, c'est indispensable pour que le formulaire envoie correctement le fichier au serveur.

Création du JavaBean

Commençons donc par créer un objet intitulé **Fichier**, contenant deux propriétés :

- la **description** du fichier, stockée sous forme d'une simple String ;
- le **contenu** du fichier, stockant les données envoyées par l'utilisateur.

Un rapide parcours de la documentation de la balise **<t:inputFileUpload>** nous apprend que l'objet utilisé pour stocker le fichier envoyé par l'utilisateur est de type **UploadedFile**.

Nous allons également reprendre les vérifications que nous effectuons dans notre système d'upload basé sur les servlets, à savoir :

- un fichier doit obligatoirement être transmis ;
- une description, si elle est renseignée, doit contenir au moins 15 caractères.

Pour les mettre en place, nous allons à nouveau pouvoir utiliser de simples annotations comme nous l'avons découvert dans le chapitre précédent. Voici le code du bean en résultant :

Code : Java - com.sdzee.entities.Fichier

```
package com.sdzee.entities;

import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

import org.apache.myfaces.custom.fileupload UploadedFile;

public class Fichier {

    @Size( min = 15, message = "La phrase de description du fichier
doit contenir au moins 15 caractères" )
    private String description;
    @NotNull( message = "Merci de sélectionner un fichier à envoyer"
)
    private UploadedFile contenu;

    public String getDescription() {
        return description;
    }

    public void setDescription( String description ) {
        this.description = description;
    }

    public UploadedFile getContenu() {
        return contenu;
    }

    public void setContenu( UploadedFile contenu ) {
        this.contenu = contenu;
    }
}
```

Vous retrouvez sans surprise les annotations **@Size** et **@NotNull** appliquées aux propriétés, accompagnées de leur message d'erreur respectif.

Création du backing-bean

La dernière étape consiste à créer le backing-bean qui va mettre tout ce petit monde en musique. Celui-ci doit donc s'intituler **UploadBean**, contenir et initialiser une instance d'un bean de type **Fichier**, ainsi qu'une méthode intitulée **envoyer()** :

Code : Java - com.sdzee.beans.UploadBean

```
package com.sdzee.beans;

import java.io.IOException;
import java.io.Serializable;

import javax.faces.application.FacesMessage;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.RequestScoped;
import javax.faces.context.FacesContext;

import org.apache.commons.io.FileUtils;
import org.apache.commons.io.FilenameUtils;

import com.sdzee.entities.Fichier;

@ManagedBean
@RequestScoped
public class UploadBean implements Serializable {
    private static final long serialVersionUID = 1L;

    private Fichier fichier;

    // Initialisation du bean fichier
    public UploadBean() {
        fichier = new Fichier();
    }

    public void envoyer() throws IOException {
        String nomFichier = FilenameUtils.getName(
fichier.getContenu().getName());
        String tailleFichier = FileUtils.byteCountToDisplaySize(
fichier.getContenu().getSize());
        String typeFichier = fichier.getContenu().getContentType();
        byte[] contenuFichier = fichier.getContenu().getBytes();

        /*
        * Effectuer ici l'enregistrement du contenu du fichier sur le
        * disque,
        * ou dans la BDD (accompagné du type du contenu, éventuellement),
        * ou
        * tout autre traitement souhaité...
        */

        FacesContext.getCurrentInstance().addMessage( null, new
FacesMessage(
            String.format( "Fichier '%s', de taille '%s' et de
type '%s' envoyé avec succès !",
                nomFichier, tailleFichier, typeFichier ) )
    );
}

    public Fichier getFichier() {
        return fichier;
    }

    public void setFichier( Fichier fichier ) {
        this.fichier = fichier;
    }
}
```

Vous reconnaisserez dans cet exemple la structure d'un backing-bean, très semblable à celui que nous avons créé dans le chapitre précédent : les annotations **@ManagedBean** et **@RequestScoped**, le constructeur pour initialiser le bean de type **Fichier** et

la paire de *getter/setter* associée.

C'est dans la méthode `envoyer()` que vous devrez appeler les traitements que vous souhaitez effectuer sur votre fichier : l'écrire sur le disque comme nous l'avions fait dans notre système basé sur les servlets, ou pourquoi pas l'enregistrer dans une table dans votre base de données, etc.

J'ai, pour ma part, choisi d'effectuer de simples récupérations d'informations concernant le fichier dans cet exemple, afin de ne pas encombrer le code inutilement. Cela me permet tout de même de vous montrer l'utilisation des méthodes de l'objet `UploadedFile` :

- à la ligne 29, je récupère le nom du fichier via `UploadedFile.getName()` et le convertis dans un format propre grâce à la méthode utilitaire `FilenameUtils.getName()` ;
- à la ligne 30, je récupère la taille du fichier via `UploadedFile.getSize()` et la convertis dans un format lisible grâce à la méthode utilitaire `FileUtils.byteCountToDisplaySize()` ;
- à la ligne 31, je récupère directement le type du fichier via `UploadedFile.getContentType()`.

Je génère finalement un simple `FacesMessage`, initialisé avec un identifiant à `null` et un message récapitulant le nom, la taille et le type du fichier envoyé, que j'ajoute au `FacesContext`. Tout ceci est fait, comme vous devez vous en douter, en prévision d'un affichage dans notre Facelet via la balise `<h:messages globalOnly="true">`.

Tests et vérifications

Tout est prêt, nous pouvons tester notre formulaire d'envoi. Rendez-vous sur la page `http://localhost:8088/pro_jsf/upload.xhtml`. Vous devrez alors observer le même formulaire vierge que celui mis en place dans notre ancien système d'upload basé sur les servlets (voir la figure suivante).

The screenshot shows a web browser window with the URL `localhost:8088/pro_jsf/upload.xhtml` in the address bar. The page title is "Envoyer de fichier". There are two input fields: one for "Description du fichier" which is empty, and one for "Emplacement du fichier *". The "Emplacement du fichier" field contains the text "Choose File No file chosen". Below these fields is a "Envoyer" button.

L'affichage du champ de sélection du fichier peut varier selon le navigateur que vous utilisez et la langue de votre système. Ici, il s'agit de l'affichage sous le navigateur Chrome sur un système en langue anglaise. Ne vous inquiétez donc pas si vous obtenez un rendu sensiblement différent.

Essayons pour commencer de valider sans rien saisir ni sélectionner, comme indiqué à la figure suivante.

The screenshot shows the same web browser window as the previous one, but now with an error message "Merci de sélectionner un fichier à envoyer" displayed in red text next to the "Emplacement du fichier" field. The rest of the form appears identical to the first screenshot.

Nous observons alors un message d'erreur sur le champ du fichier, et rien sur la description. C'est bien le comportement attendu, puisque nous n'avons mis une annotation `@NotNull` que sur la propriété `contenu`, et pas sur la description.

Essayons ensuite de valider en saisissant une description de moins de 15 caractères, sans sélectionner de fichier (voir la figure suivante).

Cette fois, l'annotation `@Size` entre en jeu, et nous sommes prévenus par un message d'erreur sur le champ `description` que nous devons saisir au moins 15 caractères.

Essayons enfin de valider en saisissant une description assez longue, et en sélectionnant un fichier léger quelconque depuis notre disque, ainsi qu'indiqué à la figure suivante.

J'ai, pour ma part, sélectionné un fichier de type PDF et nommé très sobrement **1.pdf**. Cette fois, l'envoi est validé et nous observons le message d'information que nous avons construit dans la méthode `envoyer()` de notre backing-bean. En l'occurrence, le message me précise bien le titre de mon fichier, sa taille et son type MIME.

 Nous avons donc réussi à mettre en place un système d'envoi de fichiers très simplement, grâce au composant fourni par la bibliothèque Tomahawk. Si vous souhaitez reproduire fidèlement le comportement de notre ancien système d'upload, il ne vous reste plus qu'à intégrer une méthode d'écriture du fichier sur le disque directement dans votre backing-bean, ou bien dans une classe utilitaire que vous appellerez depuis la méthode `envoyer()` !

Limitation de la taille maximale autorisée

Avant d'en terminer avec ce formulaire, je tiens à vous préciser une information importante au sujet du composant que nous utilisons pour l'envoi de fichiers : il ne sait pas très bien gérer une limitation de la taille maximale autorisée. Voyons cela ensemble en détail.

Dans [la documentation du filtre de Tomahawk](#), nous apprenons que dans la section `<filter>` de la déclaration du filtre dans le `web.xml`, nous avons la possibilité d'insérer un paramètre d'initialisation afin de définir la taille maximale autorisée par fichier :

Code : XML - Ajout d'une limite de taille maximale pour les fichiers

```

<filter>
    <filter-name>MyFacesExtensionsFilter</filter-name>
    <filter-
class>org.apache.myfaces.webapp.filter.ExtensionsFilter</filter-
class>
    <init-param>
        <param-name>uploadMaxFileSize</param-name>
        <param-value>10m</param-value>
    </init-param>
</filter>

```

Le format à respecter dans le champ `<param-value>` est le suivant :

- 10 -> 10 octets de données ;
- 10k -> 10 Ko de données ;
- 10m -> 10 Mo de données ;
- 1g -> 1 Go de données.

Le problème est que lorsque cette limite est dépassée, Tomahawk ne prépare pas un joli FacesMessage prêt à être affiché dans notre Facelet à côté du composant `<t:inputFileUpload>`. Il envoie à la place une exception de type `SizeLimitExceededException`, qui sort du cadre normal d'exécution de JSF. Je ne rentre pas dans les détails, mais pour faire simple il n'est tout bonnement pas possible de gérer proprement cette exception !

Vous pouvez faire le test si vous le souhaitez : mettez en place cet ajout dans la déclaration du filtre dans votre web.xml, par exemple avec une taille maximale de 1Mo, et essayez ensuite d'envoyer via votre formulaire un fichier dont la taille dépasse cette limite. Vous ne visualiserez en retour aucun message d'erreur sur votre formulaire, ni aucun message de validation. La seule erreur que vous pourrez trouver sera affichée dans le fichier `/glassfish/domains/domain1/logs/server.log` de GlassFish, qui contiendra une ligne mentionnant l'exception dont je viens de vous parler...



Comment faire pour gérer cette exception ?

Je vous l'ai déjà dit, ce n'est pas possible proprement. Ce qu'il est possible de faire par contre, c'est de mettre en place une parade : nous pouvons mettre en place une limite de taille très grande, par exemple 100Mo de données, et nous occuper de la vérification de la taille du fichier envoyé grâce à un Validator JSF.



Alors bien évidemment, cela ne règle pas entièrement le problème : dans l'absolu, un utilisateur pourra toujours tenter d'envoyer un fichier trop volumineux (pesant plus de 100Mo), et se retrouver face à un formulaire sans erreur apparente. Mais nous pouvons considérer que si un utilisateur s'amuse à envoyer un fichier aussi gros, alors ça sera bien fait pour lui, il ne mérite pas d'être proprement informé de l'erreur !



Mettons donc en place une limitation large dans le web.xml :

Code : XML - Ajout d'une limite de taille de 100Mo

```
<filter>
    <filter-name>MyFacesExtensionsFilter</filter-name>
    <filter-
class>org.apache.myfaces.webapp.filter.ExtensionsFilter</filter-
class>
    <init-param>
        <param-name>uploadMaxFileSize</param-name>
        <param-value>100m</param-value>
    </init-param>
</filter>
```

Il nous faut alors créer un Validator, comme nous l'avons fait à deux reprises dans le chapitre précédent. Nous allons le nommer `TailleMaxFichierValidator` et le placer comme ses frères dans le package `com.sdzee.validators` :

Code : Java - com.sdzee.validators.TailleMaxFichierValidator

```
package com.sdzee.validators;

import javax.faces.application.FacesMessage;
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.validator.FacesValidator;
import javax.faces.validator.Validator;
import javax.faces.validator.ValidatorException;

import org.apache.myfaces.custom.fileupload UploadedFile;

@FacesValidator( "tailleMaxFichierValidator" )
public class TailleMaxFichierValidator implements Validator {
```

```

    private static final long TAILLE_MAX_FICHIER = 1 * 1024 *
1024; // 1Mo
    private static final String MESSAGE_ERREUR = "La taille
maximale autorisée est de 1Mo";

    public void validate( FacesContext context, UIComponent
component, Object value ) throws ValidatorException {
        if ( value != null && ( (UploadedFile) value ).getSize() >
TAILLE_MAX_FICHIER ) {
            throw new ValidatorException( new FacesMessage(
FacesMessage.SEVERITY_ERROR, MESSAGE_ERREUR, null ) );
        }
    }
}

```

Vous retrouvez l'annotation `@FacesValidator`, permettant de déclarer l'objet comme tel auprès de JSF.

Le code de la méthode `validate()` est ensuite très simple. Nous nous contentons de vérifier que la taille du fichier ne dépasse pas la limite définie - en l'occurrence j'ai fixé cette limite à 1Mo - et envoyons le cas échéant une exception initialisée avec un `FacesMessage` de gravité erreur et contenant un message d'avertissement destiné à l'utilisateur.

Pour terminer, nous devons lier notre Validator au composant d'envoi de fichiers dans notre Facelet, par l'intermédiaire de la balise `<f:validator>` que vous connaissez déjà. Remplacez tout bêtement la déclaration de la balise `<t:inputFileUpload>` par :

Code : HTML

```

<t:inputFileUpload id="fichier"
value="#{uploadBean.fichier.contenu}">
    <f:validator validatorId="tailleMaxFichierValidator" />
</t:inputFileUpload>

```

Pour rappel, le nom passé dans l'attribut `validatorId` est le nom défini dans l'annotation `@FacesValidator` de notre objet `TailleMaxFichierValidator`.

Nous pouvons maintenant vérifier le bon fonctionnement de notre système de contrôle de la taille d'un fichier. Rendez-vous à nouveau sur le formulaire d'envoi depuis votre navigateur, et essayez d'envoyer un fichier dont le poids est supérieur à 1Mo (et bien évidemment, inférieur à 100Mo).

Vous observez alors que le fichier est correctement envoyé au serveur, ce qui est normal puisque la limite de taille autorisée est fixée à 100Mo. Une fois le fichier récupéré, JSF applique alors le Validator que nous avons créé et constate que le fichier est trop volumineux. Il affiche donc le message d'erreur à l'utilisateur, celui que nous avons préparé dans l'exception envoyée depuis la méthode `validate()`, sur le champ de sélection du fichier.



Vous savez maintenant vérifier la taille d'un fichier, malgré l'incapacité du composant à gérer proprement la taille maximale qu'il est permis de définir dans le filtre.

Et plus si affinités...

TP Fil rouge - Étape 8

Il vous reste encore un bout de chemin à arpenter seuls, et vous serez prêts pour la huitième et dernière étape du fil rouge ! Eh oui, je vous avais à moitié menti en vous disant que la septième étape du fil rouge était la dernière. Pourquoi à moitié ? Parce qu'il y a bel et bien une huitième étape, mais cette fois, je ne vous fournirai pas de correction ! Vous allez devoir vous débrouiller tous seuls pour intégrer JSF dans votre application, et vous devrez notamment apprendre à utiliser les composants standard suivants :

- la balise `<h:selectOneRadio>`, pour générer le bouton radio de choix entre l'utilisation d'un client existant et la saisie d'un nouveau client ;
- la balise `<h:selectOneMenu>`, pour générer la liste déroulante de sélection d'un client existant ;
- la balise `<h:link>`, pour générer les liens HTML présents dans les différentes vues ;
- la balise `<h:dataTable>`, pour générer le tableau récapitulatif des clients existants et des commandes passées ;
- la balise `<ui:include>`, pour remplacer l'inclusion du menu jusqu'à présent réalisée via une balise de la JSTL.

Nous avons déjà appris les rudiments de JSF ensemble, ce petit exercice en solitaire ne devrait pas vous poser de problème particulier : vous allez simplement devoir chercher par vous-mêmes dans la documentation et sur le web pour trouver les informations qui vous manquent.



Pourquoi est-ce que nous n'apprenons pas ensemble à utiliser ces quelques composants ?

La première raison de ce choix est simple : je ne souhaite pas vous donner l'impression d'avoir terminé l'apprentissage de JSF ! Ce que j'entends par là, c'est qu'il ne faut surtout pas vous imaginer qu'une fois capables de reproduire le fil rouge avec JSF, vous n'aurez plus rien à apprendre. Alors certes, j'aurais pu vous tenir la main jusqu'à la fin du TP, et vous avertir ensuite comme je suis en train de le faire maintenant, mais je suis persuadé que cela n'aurait pas eu le même effet. En ne vous donnant que des pistes pour cette ultime étape du fil rouge, je vous prépare en douceur à ce qui vous attend ensuite : l'apprentissage par vous-mêmes.

La seconde raison de ce choix est que pour couvrir l'intégralité de JSF, tout en gardant ce rythme lent, pédagogique et adapté aux novices, il faudrait une collection de livres entière ! Il y a bien trop d'aspects à aborder, de fonctionnalités à découvrir, de ressources à apprendre à manipuler, etc. De toute manière, maîtriser le *framework* de A à Z n'est nécessaire que si vous envisagez de travailler à son amélioration ! Pour un usage commun, c'est-à-dire l'utilisation de JSF dans vos projets, il vous suffit d'en connaître suffisamment pour pouvoir vous débrouiller seuls lorsqu'un problème survient : cela inclut la connaissance de l'API Servlet, du processus de traitement des requêtes par JSF, des Facelets, des principaux composants, etc. Bref, tout ce que nous avons découvert ensemble dans ce cours !

Enfin, vous ne devez jamais oublier que l'expérience, les tests, les erreurs et la recherche de solutions sont les outils les plus pédagogiques qui puissent être. Lorsque vous rencontrez un problème, posez-vous les bonnes questions et interrogez-vous avant tout sur son origine avant de songer à y trouver une solution.

Ce que l'avenir vous réserve

Ce qu'il vous manque pour pouvoir voler de vos propres ailes

Les bibliothèques de composants JSF sont richissimes, et vous devez absolument vous y intéresser si vous comptez vous lancer dans le développement avec JSF. Nous les avons citées à plusieurs reprises dans le cours : PrimeFaces, RichFaces, IceFaces, OpenFaces, Tomahawk... Il existe même des solutions intermédiaires comme [OmniFaces](#), qui se focalise sur la simplification du développement avec JSF et qui gagne en maturité et en popularité.

Sans parler de ces bibliothèques, ni même des composants standard restants que nous n'avons pas étudiés, il y a encore bon nombre de concepts JSF que nous n'avons pas eu la chance de pouvoir aborder ensemble dans ce cours, notamment : la création de composants personnalisés avec JSF, la création de FacesConverters, la compréhension des portées `@Custom` et `@None` sur un backing-bean, le découpage d'une page en plusieurs templates...

Plus important encore, il y a surtout bon nombre de concepts Java/Java EE au sens large que nous n'avons pas eu la chance de pouvoir aborder ensemble dans ce cours, notamment : l'envoi de mail depuis une application web, la réécriture d'URL (ou *url-rewriting*), la mise en place de webservices, l'écriture de tests unitaires, la génération d'une base de données directement depuis un modèle d'entités JPA...

Toutes ces mentions ne sont que des exemples, pour vous montrer que même lorsque vous savez créer une application web, il existe toujours des domaines que vous ne connaissez ou ne maîtrisez pas encore, et qui n'attendent que vous ! 😊

Bientôt dans les bacs...

Pour clore ce triste paragraphe aux allures de testament, n'oubliez pas de rester attentifs aux mises à jour apportées régulièrement aux technologies Java et Java EE. La sortie de Java EE 7 est proche (prévue pour mars 2013 lors de la rédaction de ce cours), et par conséquent celles de JSF 2.2, de JPA 2.1, de GlassFish 4, etc. Un lot de nouveautés fera comme toujours son apparition, et vous devrez vous y frayer un chemin si vous souhaitez rester efficace !



Vous devez maintenant en avoir pleinement conscience : le monde Java EE est immensément vaste ! Vous avez acquis, en suivant ce cours, les bases, les bonnes pratiques et les réflexes nécessaires pour être autonomes dans votre progression future. Par-dessus tout, j'espère que ce cours vous a donné l'envie de continuer et d'en apprendre plus, d'aller plus loin, que ce soit pour simplifier le développement de vos applications, ou pour développer des fonctionnalités toujours plus avancées.

- Dans sa version courante (JSF 2.1), le *framework* ne fournit pas de composant pour l'envoi de fichiers, et ne sait pas gérer les requêtes de type **multipart**.
- La prochaine version du *framework* (JSF 2.2) est prévue à ce jour pour mars 2013 et devrait pallier ce manque.
- La plupart des bibliothèques de composants proposent une balise dédiée à l'envoi de fichiers, et un filtre pour la gestion des requêtes **multipart**.
- Avec la bibliothèque Tomhawk, il suffit de :
 - déclarer le filtre **ExtensionsFilter** dans le web.xml du projet ;
 - mettre en place la balise **<t:inputFileUpload>** dans le formulaire ;
 - utiliser un objet de type **UploadedFile** pour récupérer le fichier envoyé.
- Tomahawk ne permet pas de gérer proprement la limitation de la taille maximale autorisée pour l'envoi de fichiers.
- Un contournement simple et rapide consiste à relever la limite autorisée, et à mettre en place un **Validator** pour vérifier manuellement la taille du fichier envoyé.
- Les univers JSF, JPA et Java EE en général sont immenses : faire preuve d'autonomie, de curiosité et d'initiative vous permettra de progresser efficacement !

Avec ce bagage technique supplémentaire, nous voilà maintenant équipés pour nous lancer dans le développements d'applications web basées sur des standards conçus pour faciliter et accélérer notre travail. Nous avons toutes les clés en mains pour construire nos projets de manière efficace et professionnelle !

Partie 7 : Annexes

Dans ces deux chapitres indépendants, vous retrouverez des informations pratiques concernant le déploiement d'une application web Java EE d'une part, et le débogage et l'analyse d'une application d'autre part.

Débugger un projet

Dans cette légère annexe, je vous présente les principaux outils à connaître pour tester et débugger votre application Java EE.

Les fichiers de logs

Quels fichiers ?

Les premiers outils à connaître, si tant est que l'on puisse les qualifier d'outils, sont les fichiers de logs des différents composants de votre application. Ils contiennent les messages, avertissements et erreurs enregistrés lors du démarrage des composants et lors de leur fonctionnement. Pour des applications basiques, comme celles que nous développons dans ce cours, trois fichiers différents peuvent vous être utiles :

- les logs du serveur d'applications ;
- les logs du SGBD ;
- les logs de la JVM.

Fichier de logs du serveur d'applications

Le fichier dont vous aurez le plus souvent besoin est le fichier de logs de votre serveur d'applications. Il contient des informations enregistrées au démarrage du serveur, lors du déploiement de votre application et lors de l'utilisation de votre application. C'est ici par exemple qu'atterrissent toutes les exceptions imprévues, c'est-à-dire celles qui ne sont pas interceptées dans votre code, et qui peuvent donc être synonymes de problèmes.

Selon le serveur que vous utilisez et le système d'exploitation sur lequel il s'exécute, le fichier se situera dans un répertoire différent. Vous pourrez trouver cette information sur le web ou tout simplement dans la documentation de votre serveur. À titre d'exemple, le fichier de logs d'une application web sous GlassFish se nomme **server.log** et se situe dans le répertoire **/glassfish/domains/nom-du-domaine/logs/**, où *nom-du-domaine* dépend de la manière dont vous avez configuré votre serveur lors de son installation. Par défaut, ce répertoire se nomme **domain1**.

Voici un exemple de contenu possible :

Code : Console - Extrait de logs de GlassFish

```
Dec 3, 2012 2:16:40 PM com.sun.enterprise.admin.launcher.GFLauncherLogger info
INFO: Successfully launched in 16 msec.
[#|2012-12-03T14:43:33+0800|INFO|glassfish3.1.2|com.sun.enterprise.server.loggi
Running GlassFish Version: GlassFish Server Open Source Edition 3.1.2.2 (build 5)|#
[#|2012-12-03T14:43:43.629+0800|INFO|glassfish3.1.2|javax.enterprise.system.core.cc
Grizzly Framework 1.9.50 started in: 61ms - bound to [0.0.0.0:4848]|#]

...
[#|2012-12-09T21:55:54.774+0800|SEVERE|glassfish3.1.2|javax.enterprise.system.tools
Exception while preparing the app : Exception [EclipseLink-4002] (Eclipse Persisten
Internal Exception: java.sql.SQLException: Error in allocating a connection. Cause:
JDBC url = jdbc:mysql://localhost:3306/bdd_sdzee, username = java.
Terminating connection pool. Original Exception: -----
com.mysql.jdbc.exceptions.jdbc4.CommunicationsException: Communications link failur
The last packet sent successfully to the server was 0 milliseconds ago. The driver
at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
....
```

Vous pouvez définir le niveau de détail des informations enregistrées via la console d'administration **asadmin** que je vous fais

découvrir dans le chapitre du cours portant sur JPA. En ce qui concerne le framework de persistance EclipseLink, il suffit d'ajouter une section dans le fichier de configuration **persistence.xml** de votre application :

Code : XML - Niveau de détails de logs d'EclipseLink

```
<persistence-unit ...>
  ...
  <properties>
    <property name="eclipselink.logging.level.sql"
      value="FINE"/>
    <property name="eclipselink.logging.parameters"
      value="true"/>
  </properties>
</persistence-unit>
```

En procédant à cet ajout, vous pourrez visualiser les requêtes SQL générées par votre application au sein du fichier de logs de GlassFish.

Fichier de logs du SGBD

Votre gestionnaire de base de données peut également générer des fichiers de logs. Il existe généralement différents niveaux de détails : l'enregistrement des erreurs uniquement, l'enregistrement de toutes les connexions établies et requêtes effectuées, l'enregistrement des requêtes longues (on parle alors de *Slow Query*), etc.

Là encore, selon la BDD utilisée et le système d'exploitation sur lequel elle s'exécute, les fichiers se trouveront dans un répertoire différent. À titre d'exemple toujours, MySQL n'enregistre par défaut aucune information (à l'exception des erreurs sous Windows uniquement). Il est alors possible d'activer les différents modes de *logging* en ajoutant des paramètres lors du lancement du serveur SQL :

- pour activer le mode *Error Log*, il suffit d'ajouter `--log-error[=nom_du_fichier]` à la commande de lancement de MySQL, où *nom_du_fichier* est optionnel et permet de spécifier dans quel fichier enregistrer les informations ;
- pour activer le mode *General Query Log*, c'est-à-dire l'enregistrement des connexions et requêtes effectuées, alors il suffit d'ajouter `--log[=nom_du_fichier]` à la commande de lancement de MySQL, où *nom_du_fichier* est optionnel et permet de spécifier dans quel fichier enregistrer les informations.

Voici un exemple de contenu d'un fichier Error Log :

Code : Console - Extrait de logs de MySQL

```
InnoDB: Setting log file ./ib_logfile1 size to 5 MB
InnoDB: Database physically writes the file full: wait...
InnoDB: Doublewrite buffer not found: creating new
InnoDB: Doublewrite buffer created
InnoDB: 127 rollback segment(s) active.
InnoDB: Creating foreign key constraint system tables
InnoDB: Foreign key constraint system tables created
121114  0:29:59  InnoDB: Waiting for the background threads to start
121114  0:30:00  InnoDB: 1.1.8 started; log sequence number 0
121114  0:30:00  [Note] Server hostname (bind-address): '0.0.0.0'; port: 3306
```

Par défaut sous les systèmes de type Unix, le fichier de logs d'erreurs se trouve dans `/usr/local/mysql/data/`.

Fichier de logs de la JVM

Dans certains cas, vous pouvez être confrontés à des problèmes liés aux fondations de votre application, à savoir la JVM utilisée pour faire tourner Java sur votre machine. Elle aussi peut enregistrer des logs d'erreurs, et vous pouvez spécifier dans quel répertoire et quel fichier en ajoutant `-XX:ErrorFile=nom_du_fichier` lors de son lancement. Par exemple, si vous souhaitez que votre JVM crée un fichier nommé **jvm_error_xxx.log** dans le répertoire **/var/log/java/**, où **xxx** est l'identifiant du processus courant, il faut lancer Java ainsi :

Code : Console

```
java -XX:ErrorFile=/var/log/java/java_error_%p.log
```

Si l'option `-XX:ErrorFile=` n'est pas précisée, alors le nom de fichier par défaut sera `hs_err_pidxxx.log`, où `xxx` est l'identifiant du processus courant. En outre, le système essaiera de créer ce fichier dans le répertoire de travail du processus. Si jamais le fichier ne peut être créé pour une raison quelconque (espace disque insuffisant, problème de droits ou autre), alors le fichier sera créé dans le répertoire temporaire du système d'exploitation (`/tmp` sous Linux, par exemple).

Comment les utiliser ?

Si vous avez intégré votre serveur d'applications à Eclipse, alors son fichier de logs sera accessible depuis le volet inférieur de l'espace de travail Eclipse, ce qui vous permettra de ne pas avoir à quitter votre IDE pour régler les problèmes liés au code de votre application. Par exemple, si vous utilisez le serveur GlassFish depuis Eclipse il vous suffit de faire un clic droit sur le nom de votre serveur, et de suivre `GlassFish > View Log File`.

En ce qui concerne les logs des autres composants, il vous faudra le plus souvent aller les chercher et les parcourir par vous-mêmes, manuellement. Pour ne rien vous cacher, ce travail est bien plus aisé sous les systèmes basés sur Unix (Mac OS, Linux, etc.), qui proposent, via leur puissant terminal, quelques commandes très pratiques pour la recherche et l'analyse de données, notamment `tail -f` et `grep`.

Cela dit, quel que soit le système d'exploitation utilisé, il est souvent intéressant de pouvoir colorer le contenu d'un fichier de logs, afin de pouvoir plus facilement repérer les blocs d'informations intéressantes parmi le flot de messages qu'il contient. Pour ce faire, vous pouvez par exemple utiliser un logiciel comme `SublimeText`. La coloration syntaxique vous aidera grandement au déchiffrage des sections qui vous intéressent. Observez par exemple sur la figure suivante un extrait du fichier `server.log` de GlassFish ouvert avec SublimeText, coloration syntaxique activée.

```

1326      at java.net.Socket.<init>(Socket.java:375)
1327      at java.net.Socket.<init>(Socket.java:218)
1328      at com.mysql.jdbc.StandardSocketFactory.connect(StandardSocketFactory.java:257)
1329      at com.mysql.jdbc.MysqlIO.<init>(MysqlIO.java:298)
1330      ... 82 more
1331
1332
1333      at com.jolbox.bonecp.BoneCP.<init>(BoneCP.java:353)
1334      at com.jolbox.bonecp.BoneCPDataSource.maybeInit(BoneCPDataSource.java:150)
1335      at com.jolbox.bonecp.BoneCPDataSource.getConnection(BoneCPDataSource.java:111)
1336      at com.sun.gjc.spi.DSManagedConnectionFactory.createManagedConnection(DSManagedConnectionFactory.java:115)
1337      ... 63 more
1338 Caused by: com.mysql.jdbc.exceptions.jdbc4.CommunicationsException: Communications link failure
1339
1340 The last packet sent successfully to the server was 0 milliseconds ago. The driver has not received any packets
1341 at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
1342 at sun.reflect.NativeConstructorAccessorImpl.newInstance(NativeConstructorAccessorImpl.java:39)
1343 at sun.reflect.DelegatingConstructorAccessorImpl.newInstance(DelegatingConstructorAccessorImpl.java:27)
1344 at java.lang.reflect.Constructor.newInstance(Constructor.java:513)
1345 at com.mysql.jdbc.Util.handleNewInstance(Util.java:411)
1346 at com.mysql.jdbc.SQLError.createCommunicationsException(SQLError.java:1116)
1347 at com.mysql.jdbc.MysqlIO.<init>(MysqlIO.java:348)

```

C'est déjà bien moins austère ainsi, n'est-ce pas ? 😊

Le mode debug d'Eclipse

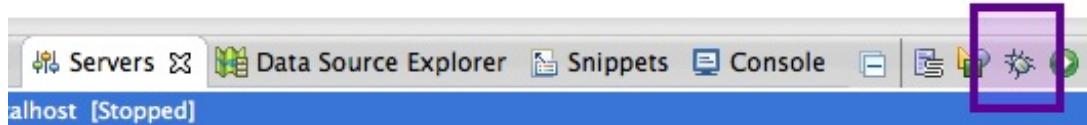
Principe

Le second outil le plus utile pour le développement d'une application avec Eclipse est le mode **Debug** que ce dernier fournit. Il s'agit d'un mode d'exécution **pas à pas**, que vous avez la possibilité de contrôler manuellement et de manière aussi fine que souhaitée. Le principe est simple : Eclipse prend la main sur votre serveur, et vous offre la possibilité de littéralement mettre en pause l'exécution du code de votre application.

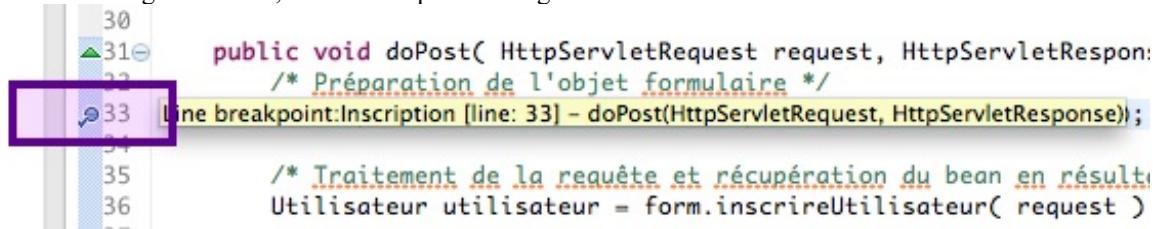
Le système se base sur des **breakpoints**, de simples marqueurs que vous avez la possibilité de poser sur les lignes de code que vous souhaitez analyser.

Interface

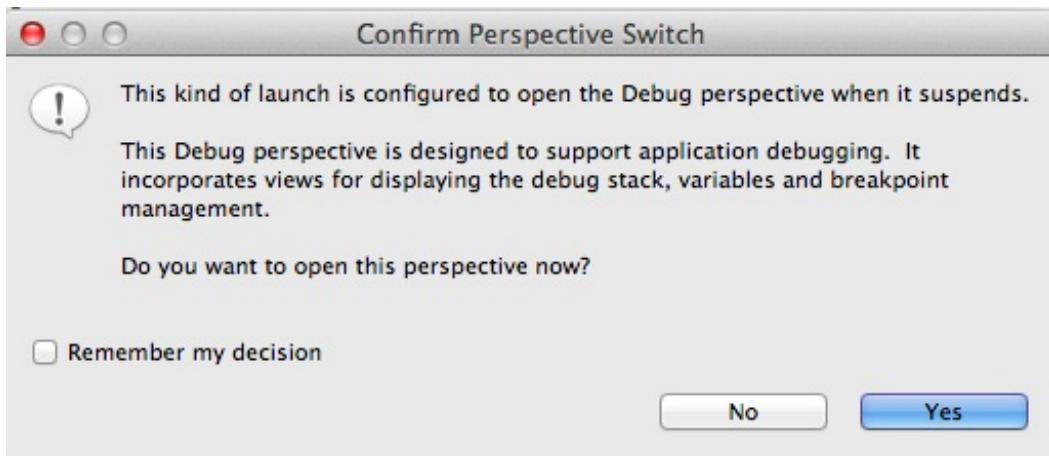
Pour lancer le serveur en mode debug, il ne faut plus utiliser le bouton classique mais cette fois celui qui ressemble à un petit insecte, ou *bug* en anglais. Le serveur va alors démarrer, et vous pourrez commencer à utiliser l'application que vous y avez déployée sans différence apparente avec le mode classique (voir la figure suivante).



Pour pouvoir mettre en pause l'exécution de votre application et étudier son fonctionnement, il faut au minimum ajouter un **breakpoint** dans la portion de code que vous souhaitez analyser. Cela se fait tout simplement en double-cliquant sur l'espace vide situé à gauche d'une ligne de code, comme indiqué sur la figure suivante.



Un petit rond bleu apparaît alors dans l'espace libre et, au survol de ce marqueur, une infobulle vous informe de la ligne de code et de la méthode ciblée par le breakpoint. Rendez-vous à nouveau dans le navigateur depuis lequel vous testez l'application, et effectuez à nouveau la ou les actions faisant intervenir la portion de code sur laquelle vous avez posé un marqueur. Cette fois, votre navigateur va rester en attente d'une réponse de votre serveur, et une fenêtre d'avertissement va alors apparaître dans Eclipse (voir la figure suivante).

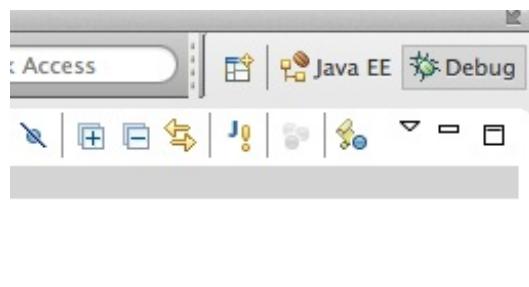


Eclipse vous informe que vous vous apprêtez à ouvrir un espace visuel dédié au mode debug. Vous devrez alors cliquer sur Yes pour accepter, et Eclipse vous affichera alors une nouvelle interface depuis laquelle vous allez pouvoir contrôler votre application.



Pendant ce temps, le navigateur reste logiquement en attente d'une réponse du serveur, tant que vous n'avez pas fait avancer le déroulement jusqu'au bout du cycle en cours, c'est-à-dire jusqu'au renvoi d'une réponse au client.

Depuis l'espace de debug, vous avez pour commencer la possibilité de revenir vers la vue de travail classique, sobrement intitulée **Java EE**, en cliquant sur le bouton situé dans le volet en haut à droite de l'écran (voir la figure suivante).



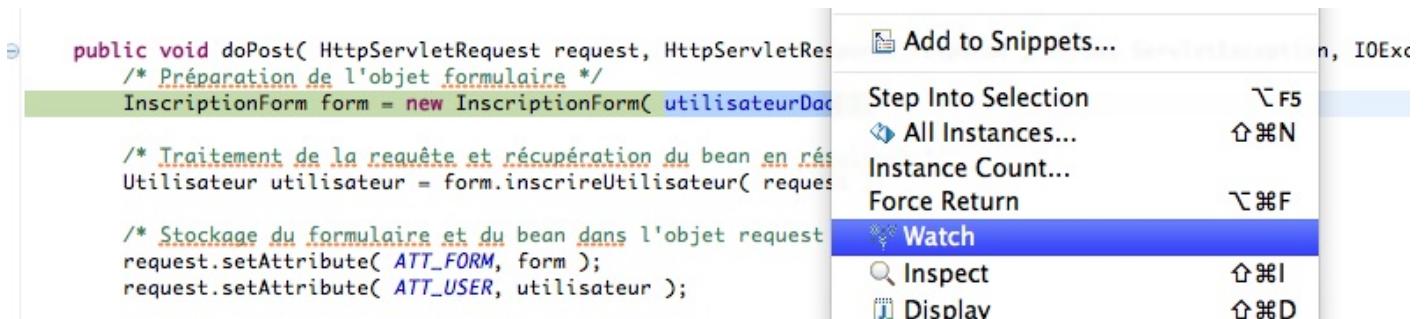
Réiproquement, vous pourrez passer de la vue d'édition vers la vue de debug en cliquant sur le bouton intitulé **Debug** depuis la vue de travail classique.

Deuxième volet important, celui situé en haut à gauche de l'écran de debug (voir la figure suivante).

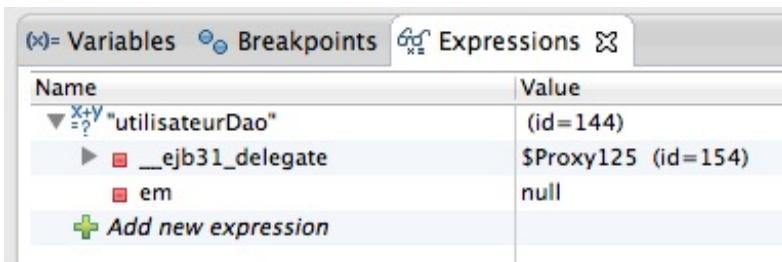


C'est via ces quelques boutons que vous allez pouvoir commander le déroulement de l'exécution de votre code.

Une autre fonctionnalité importante du mode debug est la possibilité de surveiller des variables ou des objets, d'inspecter leur contenu en temps réel afin de vérifier qu'elles se comportent bien comme prévu. Il faut pour cela sélectionner l'objet à observer dans le code affiché dans le volet gauche central de la fenêtre de debug, puis effectuer un clic droit et choisir **Watch** (voir la figure suivante).



Le volet supérieur droit de l'écran de debug va alors se mettre à jour et afficher les informations concernant l'objet que vous avez sélectionné (voir la figure suivante).



Vous pouvez bien entendu surveiller plusieurs objets simultanément. C'est dans cette petite fenêtre que vous pourrez vérifier que les données que vous créez et manipulez dans votre code sont bien celles que vous attendez, au moment où vous l'attendez !

Exemple pratique

Assez disserté, passons à l'action. Nous allons mettre en place un projet constitué d'une seule et unique servlet, dans le but de faire nos premiers pas avec le mode debug d'Eclipse, mais pas seulement : nous allons également en profiter pour observer le cycle de vie d'une servlet, et son caractère *multithreads*. D'une pierre deux coups !

Pour commencer, nous allons créer un nouveau projet web dynamique depuis Eclipse, et le nommer **test_debug**. Nous allons ensuite y créer une nouvelle servlet nommée **Test** et placée dans un package intitulé **com.test**, dont voici le code :

Code : Java - Servlet de test

```

package com.test;

import java.io.IOException;

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

@WebServlet( "/test" )
public class Test extends HttpServlet {

    private int compteur = 0;

    @Override
    public void init() throws ServletException {
        System.out.println( ">> Servlet initialisée." );
    }

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

    int compteurLocal = 0;
    ++compteurLocal;
    System.out.println( ">> Compteurs incrémentés." );
}

    @Override
    public void destroy() {
        System.out.println( ">> Servlet détruite." );
    }
}

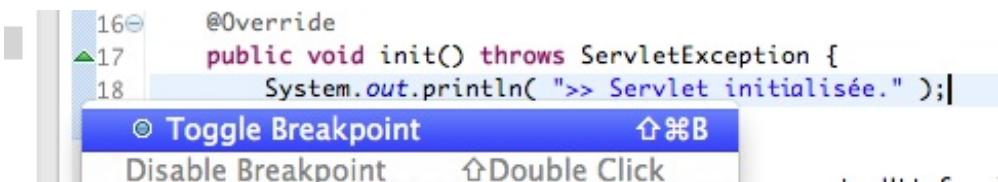
```

Avant de passer aux tests, attardons-nous un instant sur le code ici mis en place. Vous devez vous souvenir des méthodes `init()` et `destroy()`. Nous les avons déjà rencontrées dans nos exemples, que ce soit dans des servlets ou dans des filtres. Pour rappel, elles sont présentes par défaut dans la classe mère de toute servlet, `HttpServlet` (qui les hérite elle-même de sa classe mère `GenericServlet`), et ne doivent pas nécessairement être surchargées dans une servlet, contrairement aux méthodes `doXXX()` dont une au moins doit être présente dans chaque servlet.

Dans cet exemple, nous surchargeons donc ces méthodes `init()` et `destroy()` aux lignes 17 à 19 et 30 à 32, dans lesquelles nous ne faisons rien d'autre qu'afficher un simple message dans la sortie standard, à savoir la console du serveur. Ne vous inquiétez pas, vous allez très vite comprendre pourquoi nous nous embêtons à surcharger ces deux méthodes !

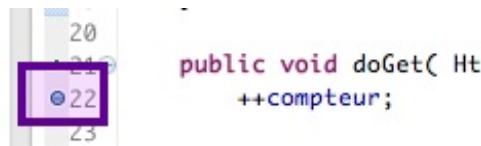
Reste alors le code de la méthode `doGet()`. Là encore, rien de bien compliqué : nous incrémentons deux compteurs, l'un étant déclaré localement dans la méthode `doGet()`, et l'autre étant déclaré en tant que variable d'instance.

Voilà tout ce dont nous allons avoir besoin. Pas besoin de JSP, ni de fichier web.xml d'ailleurs puisque nous avons utilisé l'annotation `@WebServlet` pour déclarer notre servlet auprès de Tomcat ! Nous pouvons donc démarrer notre projet en mode debug, en cliquant sur le bouton ressemblant à un petit insecte dont je vous ai parlé précédemment. Avant d'accéder à notre servlet depuis notre navigateur, nous allons placer quelques **breakpoints** dans le code, afin de pouvoir mettre en pause son exécution aux endroits souhaités. Nous allons choisir les lignes 18, 22, 26 et 31, et nous allons donc effectuer un clic droit dans l'espace vide sur la gauche de chacune d'elles et choisir "Toggle breakpoint", comme indiqué sur la figure suivante.



Le même effet est réalisable en double-cliquant simplement dans cet espace vide. Une fois les **breakpoints** positionnés, nous

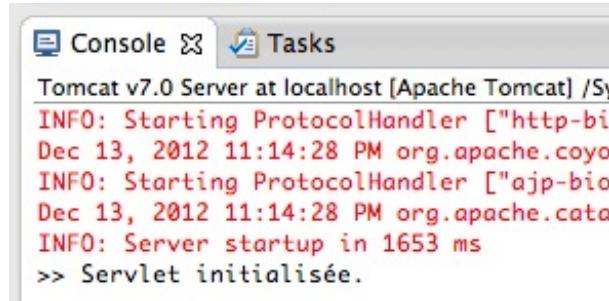
visualisons alors quatre petits ronds bleus (voir la figure suivante).



Notre environnement est maintenant prêt, nous pouvons commencer à tester. Pour ce faire, il nous suffit de nous rendre depuis notre navigateur sur la page http://localhost:8080/test_debug/test. Dès lors, Eclipse va prendre le dessus et va nous avertir que nous allons changer de vue de travail, nous cliquons alors sur Yes. Dans la vue de debug qui s'affiche, nous pouvons remarquer que l'exécution du code s'est mise en pause sur la ligne 18 du code. C'est bien le comportement que nous attendions, car comme nous accédons à notre servlet pour la première fois, sa méthode `init()` est appelée et notre premier **breakpoint** est activé.

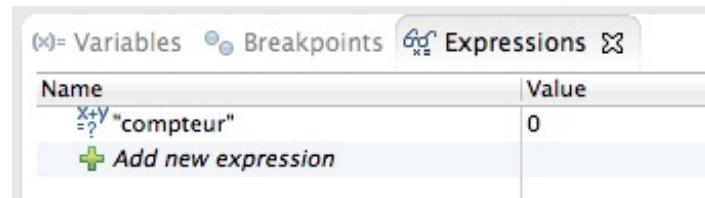
Pour poursuivre le déroulement, nous appuyons sur la touche F8 du clavier, ou sur le bouton correspondant dans le volet de contrôle en haut à gauche de la vue de debug. L'exécution continue alors et se met aussitôt en pause sur la ligne 22. Logique, puisqu'il s'agit là de notre second **breakpoint**, et qu'il est activé à chaque passage d'une requête dans la méthode `doGet()` ! En d'autres termes, à ce moment précis, la requête GET émise par notre navigateur lorsque nous avons appelé l'URL de notre servlet, a déjà été acheminée dans notre méthode `doGet()`.

Au passage, vous en profiterez pour observer que la méthode `init()` a bien été appelée en vérifiant le contenu du volet inférieur de la vue de debug. Il s'agit de la console, et si tout s'est bien passé vous devriez y trouver le message écrit par la ligne 18 de notre code (voir la figure suivante).

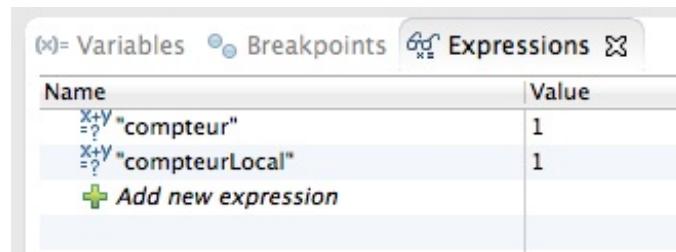


Nous allons alors sélectionner la variable **compteur** dans le code affiché dans le volet central, faire un clic droit puis choisir **Watch**.

Dans le volet supérieur droit apparaît alors le contenu de notre variable (voir la figure suivante).



Puisque la ligne n'a pas encore été exécutée, la valeur est encore zéro. Nous appuyons alors à nouveau sur F8, et le cycle se poursuit jusqu'au prochain **breakpoint** à la ligne 26. La valeur du compteur a alors changé et est maintenant logiquement 1. Nous allons également surveiller la variable **compteurLocal**, en la sélectionnant, en effectuant un clic droit et en choisissant **Watch**. Le volet supérieur droit nous affiche alors les contenus des deux variables (voir la figure suivante).



Un nouvel appui sur F8 permet de terminer le cycle : le navigateur affiche une page blanche, ce qui est logique puisque nous n'avons pas envoyé d'informations au client à travers la réponse HTTP. Aucune tâche n'étant en attente de traitement, la vue de travail de debug cesse d'afficher le contenu des variables. Nous allons maintenant changer de navigateur, et accéder une nouvelle fois à notre servlet via son URL http://localhost:8080/test_debug/test. **Du point de vue du serveur, tout se passera donc comme si la visite était issue d'un utilisateur différent.**

La vue de debug va alors s'activer à nouveau, l'exécution du code va se bloquer sur la ligne 22 et nous allons devoir appuyer sur F8 pour passer à la ligne 26. À cet instant, vous allez observer deux valeurs de compteurs différentes dans le volet des variables surveillées (voir la figure suivante).

Name	Value
X+Y "compteur"	2
X+Y "compteurLocal"	1
Add new expression	

Que s'est-il passé ?

Eh oui, à première vue nous sommes en droit de nous poser la question ! Dans notre code, nous initialisons bien nos deux compteurs à zéro. Pourquoi le compteur déclaré en tant que variable d'instance n'a-t-il pas été remis à zéro comme l'a été notre compteur local ? Eh bien tout simplement parce que comme je vous l'ai déjà expliqué à plusieurs reprises dans le cours, une servlet n'est instanciée qu'une seule et unique fois par votre serveur, et cette unique instance va ensuite être partagée par toutes les requêtes entrantes.

Concrètement, cela signifie que toutes les variables d'instance ne seront initialisées qu'une seule fois, lors de l'instanciation de la servlet. Les variables locales quant à elles sembleront réinitialisées à chaque nouvel appel. Voilà pourquoi la variable **compteur** a conservé sa valeur entre deux appels, alors que la variable **compteurLocal** a bien été réinitialisée.

"Sembleront ?"

Oui, sembleront. Car en réalité, c'est un petit peu plus compliqué que cela. Voilà comment tout cela s'organise :

- chaque requête entrante conduit à la création d'un **thread**, qui accède alors à l'instance de la servlet ;
- une variable d'instance n'est créée, comme son nom l'indique, que lors de l'instanciation de la servlet, et un unique espace mémoire est alloué pour cette variable dans la **heap** ;
- tous les threads utilisant cette instance, autrement dit toutes les requêtes entrantes dirigées vers cette servlet, partagent et utilisent cette même variable, et donc ce même espace mémoire ;
- une variable locale est initialisée à chaque utilisation par un **thread**, autrement dit par chaque nouvelle requête entrante, dans sa **stack** ;
- un espace mémoire différent est alloué pour chaque initialisation dans la stack, même comportement d'ailleurs pour les paramètres de méthodes.

Stack ? Heap ?

Vous devriez déjà connaître ces concepts si vous avez déjà programmé en Java, mais nous allons tout de même nous y pencher un instant. Dans une JVM, la mémoire est découpée en plusieurs sections : code, stack, heap et static. Pour faire simple, voici comment sont utilisées les fractions qui nous intéressent ici :

- la **stack** contient les méthodes, variables locales et variables références ;
- la **heap** contient les objets.

Voilà pourquoi notre variable d'instance est déclarée dans la **heap** : elle fait partie d'une classe, donc d'un objet.

Ce qu'il est très important de retenir ici, ce n'est pas uniquement le fait qu'une variable d'instance conserve son état d'un appel à l'autre. Ce qui est extrêmement important, c'est de bien comprendre qu'une variable d'instance est partagée par tous les clients ! Eh oui, vous avez bien constaté dans notre exemple que même en utilisant un autre navigateur, la variable **compteur** utilisée était la même que celle utilisée pour le premier appel.

Alors bien entendu, dans notre petit exemple ça ne vous paraît pas important. Mais imaginez maintenant que dans cette variable d'instance vous stockiez non plus un banal compteur mais un nom d'utilisateur, une adresse mail ou un mot de passe... De telles informations ainsi stockées seront alors partagées par tous les clients ! Et selon l'usage que vous faites de cette variable dans votre code, vous vous exposez à de graves ennuis.



Nous voilà maintenant au point sur deux aspects importants : nous avons pris en main le mode de debug d'Eclipse, et nous avons illustré le caractère *multithread* d'une servlet.

Conseils au sujet de la thread-safety

Avant de passer à la suite, revenons brièvement sur la problématique des threads et des servlets. Avec ce que nous avons posé, nous avons compris que la principale inquiétude se situe au niveau de la *thread-safety*. Nous savons maintenant que les servlets et les filtres sont partagés par toutes les requêtes. C'est un avantage du Java, c'est *multithread*, et des *threads* différents (comprendre ici "des requêtes HTTP") peuvent utiliser la même instance d'une classe. Cela serait par ailleurs bien trop coûteux (en termes de mémoire et de performances) d'en recréer une à chaque requête.

Rappelons que lorsque le conteneur de servlets démarre, il lit le fichier **web.xml** de chaque application web, et/ou scanne les annotations existantes, à la recherche des **url-pattern** associés aux servlets déclarées. Il place alors les informations trouvées dans ce qui s'apparente grossièrement à une map de servlets.



À ce sujet, sachez qu'en réalité une servlet peut être instanciée plusieurs fois par un même conteneur. En effet, si une même servlet est mappée sur plusieurs URL différentes, alors autant d'instances de la servlet seront créées et mises en mémoire. Mais le même principe de base tient toujours : une seule instance est partagée par toutes les requêtes adressant la même URL.

Ces servlets sont donc stockées dans la mémoire du serveur, et réutilisées à chaque fois qu'une URL appelée correspond à la servlet associée à l'*url-pattern* défini dans le **web.xml** ou l'annotation. Le conteneur de servlets déroule grossièrement ce processus pour chaque étape :

Code : Java

```
for (Entry<String, HttpServlet> entry : servlets.entrySet()) {  
    String urlPattern = entry.getKey();  
    HttpServlet servlet = entry.getValue();  
    if (request.getRequestURL().matches(urlPattern)) {  
        servlet.service(request, response);  
        break;  
    }  
}
```

La méthode `HttpServlet#service()` décide alors quelle méthode parmi `doGet()`, `doPost()`, etc. appeler en fonction de `HttpServletRequest#getMethod()`.

Nous voyons bien à travers ce code que le conteneur de servlets réutilise pour chaque requête **la même instance de servlet** (ici, elles sont représentées grossièrement stockées dans une map). En d'autres termes : **les servlets sont partagées par toutes les requêtes**. C'est pourquoi il est très important d'écrire le code d'une servlet de manière *threadsafe*, ce qui signifie concrètement : **ne jamais assigner de données issues des portées request ou session dans des variables d'instance d'une servlet, mais uniquement dans des variables déclarées localement dans ses méthodes**.

Comprenez bien de quoi il est ici question. Si vous assignez des données issues de la session ou de la requête dans une variable d'instance d'une servlet ou d'un filtre, alors un tel attribut se retrouverait partagé par toutes les requêtes de toutes les sessions... Il s'agit là en quelque sorte d'un débordement : alors que votre variable devrait rester confinée à sa portée, elle se retrouve accessible depuis un périmètre bien plus large. Ce qui écrase bien évidemment les principes résumés précédemment ! Ce n'est absolument pas *threadsafe* ! L'exemple ci-dessous illustre clairement cette situation :

Code : Java

```
public class MaServlet extends HttpServlet {
```

```
private Object objetNonThreadSafe; // Objet déclaré en tant que
variable d'instance

protected void doGet(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException {
    Object objetThreadSafe; // Objet déclaré localement dans la
méthode doGet()

    objetNonThreadSafe = request.getParameter("foo"); // MAUVAIS
!!! Cet objet est partagé par toutes les requêtes...
    objetThreadSafe = request.getParameter("foo"); // OK, c'est
threadsafe : l'objet reste bien dans la bonne portée.
}
}
```

Quelques outils de tests

Les tests unitaires

Un autre outil indispensable au bon développement d'une application est le test unitaire. La solution proposée par la plate-forme Java s'appelle **JUnit** : c'est un framework destiné aux tests.



Qu'est-ce qu'un test unitaire ?

Un test unitaire est une portion de code, écrite par un développeur, qui se charge d'exécuter une fonctionnalité en particulier. Le développeur doit donc en écrire autant qu'il existe de fonctionnalités dans son application, afin de tester l'intégralité du code. On parle alors de **couverture** du code ; dans un projet, plus le taux de couverture est élevé, plus le pourcentage de fonctionnalités testées est important.

Généralement dans les projets en entreprise, un niveau de couverture minimum est demandé : par exemple 80%. Cela signifie qu'au moins 80% des fonctionnalités du code doivent être couvertes par des tests unitaires.



Quel est l'intérêt de mettre en place de tels tests ?

En effet, la question est légitime. A priori, à partir du moment où le développeur teste correctement (et corrige éventuellement) ses classes et méthodes après écriture de son code, aucune erreur ne doit subsister. Oui, mais cela n'est vrai que pour une fraction de l'application : celle qui a été testée par le développeur pour vérifier que son code fonctionnait correctement. Mais rien ne garantit qu'après ajout d'une nouvelle fonctionnalité, la précédente fonctionnera toujours !

Voilà donc le principal objectif de la couverture du code : pouvoir s'assurer que le comportement d'une application ne change pas, après corrections, ajouts ou modifications. En effet, puisque chacun des tests unitaires s'applique à une seule fonctionnalité de manière indépendante, il suffit de lancer tous les tests un par un pour confirmer que tout se comporte comme prévu.



Pour information, c'est ce que l'on appelle des "tests de non régression" : s'assurer qu'après l'ajout de code nouveau ou la modification de code existant, l'application fonctionne toujours comme espéré et aucun nouveau problème - ou aucune régression - ne survient. La couverture du code par une batterie de tests unitaires permet d'automatiser la vérification, alors qu'il faudrait procéder à la main sinon, et donc être minutieux et bien penser à tester tous les cas de figure possibles.



Quelle partie du code d'une application doit être couverte ?

Seules **les classes métier et d'accès aux données** ont besoin d'être testées, car c'est ici que les fonctionnalités et résultats sont contenus. Les servlets n'étant que des aiguilleurs, elles ne sont pas concernées. Doivent donc être testés unitairement les JavaBeans, les objets métier, les EJB, les DAO, les éventuels WebServices, etc. Au sein de ces classes, seules **les méthodes publiques** doivent être testées : le fonctionnement des méthodes privées ne nous intéresse pas. Tout ce qui importe est que les méthodes publiques qui font appel à ces méthodes privées en interne retournent le bon résultat ou respectent le bon comportement.



À quoi ressemble un test unitaire ?

Les tests unitaires sont en principe codés dans un projet ou package à part, afin de ne pas être directement liés au code de l'application. Un test unitaire consiste en une méthode au sein d'une classe faisant intervenir des annotations spécifiques à JUnit. Imaginez par exemple que nous ayons écrit une classe nommée **MaClasse**, contenant des méthodes de calcul basiques nommées **multiplication()** et **addition()**. Voici alors à quoi pourrait ressembler la classe de tests unitaires chargée de vérifier son bon fonctionnement :

Code : Java - Exemple de test unitaire JUnit

```
import org.junit.*;  
  
public class Test{  
    @Test  
    public void testMultiplication() {  
        // MaClasse est ciblée par le test  
        MaClasse test = new MaClasse();  
  
        // Vérifie que multiplication( 6, 7 ) retourne bien 42  
        assertEquals( "Erreur", 42, test.multiplication( 6, 7 ) );  
    }  
  
    @Test  
    public void testAddition() {  
        // MaClasse est ciblée par le test  
        MaClasse test = new MaClasse();  
  
        // Vérifie que addition( 43, 115 ) retourne bien 158  
        assertEquals( "Erreur", 158, test.addition( 43, 115 ) );  
    }  
}
```

Chaque test unitaire suit de près ou de loin ce format, et les tests doivent pouvoir être exécutés dans un ordre arbitraire : aucun test ne doit dépendre d'un autre test. Comme vous pouvez l'observer, pour écrire un test JUnit il suffit d'écrire une méthode et de l'annoter avec `@Test`. La méthode utilisée pour vérifier le résultat de l'exécution, `assertEquals()`, est fournie par JUnit et se charge de comparer le résultat attendu avec le résultat obtenu.

Pour le lancement des tests, Eclipse permet d'intégrer intuitivement l'exécution de tests JUnit depuis l'interface utilisateur via un simple clic droit sur la classe de tests, puis Run As > JUnit Test. En ce qui concerne la couverture du code, il existe un excellent plugin qui permet de mesurer le taux de couverture d'une application, et de surligner en vert et rouge les portions de codes qui sont respectivement couvertes et non couvertes : [EclEmma](#). Pour vous donner une meilleure idée du principe, voici en figure suivante une capture de la solution en action.

The screenshot shows the Eclipse IDE interface. The top part displays the Java code for `addAll` method in `CursorableLinkedList.java`. The bottom part shows the Coverage tool window titled "TestAllPackages (31.10.2006 15:04:14)". The coverage report lists files under `java - commons-collections` with their respective coverage percentages and covered lines.

Element	Coverage	Covered Lines	Total
<code>java - commons-collections</code>	79,5 %	10927	
<code>org.apache.commons.collections</code>	74,1 %	3842	
<code>ArrayList.java</code>	86,5 %	32	
<code>BagUtils.java</code>	86,7 %	13	
<code>BeanMap.java</code>	72,4 %	155	
<code>BinaryHeap.java</code>	87,6 %	127	
<code>BoundedFifoBuffer.java</code>	93,2 %	82	
<code>BufferOverflowException.java</code>	55,6 %	5	

Nous allons nous arrêter là pour la découverte des tests unitaires, vous avez en main toutes les informations nécessaires pour savoir quoi chercher et comment mettre le tout en place dans vos projets ! 😊

Les tests de charge

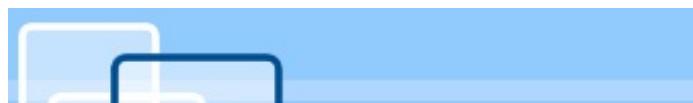
Pour conclure cette annexe, je vais vous présenter très brièvement deux solutions que vous serez un jour ou l'autre amenés à utiliser si vous développez des applications web.



La première se nomme **JMeter**, elle est éditée par la fondation Apache. Elle permet d'effectuer des tests de charge et des mesures de performances. Elle est particulièrement utile pour le test d'applications web : vérifier que le code d'une application se comporte de manière correcte est une chose, mais il faut également vérifier qu'une application fonctionne toujours de manière convenable lorsque des dizaines, centaines ou milliers de visiteurs l'utilisent simultanément.

C'est là qu'intervient JMeter : la solution permet d'automatiser et de simuler des charges importantes sur un serveur, un réseau ou un composant en particulier, ou pour analyser la performance et la réactivité globale d'une application sous différents niveaux de charge. Bref, elle vous permettra de contrôler la présence de problèmes très importants, que vous ne pouvez pas déceler manuellement. Elle permet également de générer des graphiques et rapports afin de vous faciliter l'analyse des comportements observés par la suite.

La seconde se nomme **JProfiler**. Plus complexe à maîtriser, elle n'est généralement mise en œuvre que lorsque des problèmes sont observés sur une application. Ses atouts les plus classiques sont l'analyse poussée de la mémoire utilisée lors



de l'utilisation d'une application, l'analyse de l'utilisation du CPU, l'analyse du temps passé dans chaque méthode ou portion de code, etc. Les rapports qu'elle génère permettent alors de découvrir qu'est-ce qui cause des problèmes, de comprendre pourquoi cela ne fonctionne pas comme prévu, et surtout sous quelles conditions.



Empaquetage et déploiement d'un projet

Vous trouverez dans cette annexe les informations à connaître pour packager et déployer simplement votre projet sur un serveur d'applications.

Mise en boîte du projet JAR, WAR ou EAR ?

Pour faciliter l'étape de déploiement d'un projet, il est coutume de mettre en boîte le code dans un fichier ressemblant en tout point à une archive, à l'exception de la compression. Une application web Java EE peut être empaquetée dans une archive Java (JAR), dans une archive Web (WAR) ou dans une archive dite "d'entreprise" (EAR). En réalité, sur le principe ces trois formats ne diffèrent que par leur extension : ce sont tous des fichiers JAR standard ! C'est uniquement par l'usage que les différences apparaissent.



Quand utiliser un format d'archive en particulier ?

L'utilisation de formats différents rend possible l'assemblage de différentes applications Java EE partageant des composants identiques. Aucune étape de codage supplémentaire n'est alors requise, seul l'assemblage (ou *packaging*) des différents éléments constituant une application dans des fichiers JAR, WAR ou EAR joue un rôle. Voilà comment se définit habituellement l'usage des trois formats :

- les modules EJB, qui contiennent les EJB (des classes Java) et leurs éventuels fichiers de description, sont packagés dans une archive **JAR** classique, qui contient un fichier de configuration nommé **ejb-jar.xml** dans son répertoire **/META-INF** ;
- les servlets, les pages JSP, les éléments de présentation et de design et les classes métier Java sont packagés dans une archive **WAR**, qui n'est rien d'autre qu'une archive JAR classique contenant un fichier **web.xml** dans son répertoire **/WEB-INF** ;
- les archives JAR et WAR sont à leur tour packagées dans une archive globale **EAR**, qui n'est rien d'autre qu'une archive JAR classique contenant un fichier **application.xml** dans son répertoire **/META-INF**, et qui sera finalement déployée sur le serveur.

L'intérêt majeur d'isoler les EJB dans une archive séparée est de pouvoir séparer convenablement les données et le code métier du reste de l'application. Et c'est exactement ce que l'archive EAR permet : les EJB vont dans un JAR, le contenu lié de près ou de loin au web va dans un WAR (qui au passage, n'est pas forcément un fichier, mais plutôt une structure représentant l'arborescence du projet). La conséquence de cette séparation, c'est que les deux modules JAR et WAR n'ont pas accès aux mêmes ressources : le module web peut accéder aux EJB (typiquement, des beans), le module EJB peut accéder aux éventuelles ressources définies globalement dans le module englobant EAR (typiquement, des bibliothèques), mais le module EJB ne peut pas accéder aux ressources définies dans le module web. Cette contrainte est voulue, et permet de rendre indépendant le module EJB qui doit être totalement découpé d'une quelconque information liée à la vue.



Cette isolation forcée permet ainsi de s'assurer qu'un développeur ne peut pas mélanger les concepts, que ce soit par mégarde ou par ignorance.

En outre, grâce à cette nette séparation il devient très simple de réutiliser le module EJB depuis d'autres applications web, Java SE ou autres. Si celui-ci contenait des références aux servlets ou Facelets JSF, par exemple, il serait alors compliqué de les utiliser dans d'autres contextes, en particulier dans un contexte Java SE.

Pour faire une analogie avec des concepts que vous maîtrisez déjà, vous pouvez comparer ce système avec le fait que je vous interdis d'écrire des scriptlets dans vos JSP, ou avec le principe de la programmation par contrat (interfaces et implémentations). En somme, **placer vos EJB dans un module séparé est une bonne pratique**.

Cependant, dans le cadre de projets de faible envergure, il est fréquent de ne pas se soucier de cette pratique. C'est encore plus courant avec des développeurs débutants, pour lesquels il est souvent difficile de comprendre et de se souvenir de la répartition des différents éléments. Ainsi, passer outre la contrainte de découpage et tout ranger dans un unique module WAR rend le déploiement plus aisés aux développeurs inexpérimentés, et leur facilite donc l'entrée dans le monde Java EE. Tôt ou tard, ils finiront quoi qu'il arrive par assimiler la notion de couches, et placeront leurs EJB dans un module séparé.

Par ailleurs, ce mode d'intégration simplifié fait partie d'un concept plus global intitulé **EJB Lite**, qui est destiné à rendre



possible l'utilisation d'EJB dans une application sans devoir charger l'intégralité des services que peut fournir un conteneur EJB. Cela sort du cadre de cette annexe, mais pour information sachez que c'est une des principales différences entre la version complète de GlassFish, et la version "web profile" que je vous ai fait installer.



Et donc pour déployer une application, quand utiliser un WAR, et quand utiliser un EAR ?

En résumé, deux possibilités s'offrent à vous :

- si votre projet utilise des EJB et est déployé sur un serveur d'application Java EE, alors vous pouvez suivre la recommandation et utiliser une archive EAR, qui sera elle-même constituée d'une ou plusieurs archives WAR et JAR. Vous pouvez également ignorer la bonne pratique courante et utiliser un unique fichier WAR, mais sachez que derrière les rideaux, votre serveur va en réalité automatiquement englober votre archive... dans une archive EAR !
- si votre projet web ne fait pas intervenir d'EJB, alors seule une archive WAR est requise. En outre, si votre serveur n'est pas un serveur d'applications Java EE au sens strict du terme, comme Tomcat ou Jetty par exemple, alors vous ne pourrez pas utiliser le format EAR, vous devrez obligatoirement packager vos applications sous forme d'archives WAR.

Mise en pratique

Nos projets sont de très faible envergure, ne font que très peu intervenir les EJB et nous débutons le Java EE : nous allons donc apprendre à déployer un projet sous forme d'une unique archive WAR.

Prenons pour exemple le projet intitulé **pro** que nous avons développé dans le cadre du cours. Pour l'exporter sous forme d'archive WAR depuis Eclipse, rien de plus simple ! Faites un clic droit sur votre projet dans le volet de gauche de votre espace de travail Eclipse, puis choisissez **Export > WAR File**.

Une fenêtre s'ouvre alors, dans laquelle vous devez choisir un emplacement pour enregistrer l'archive qui va être créée. Vous pouvez par ailleurs choisir d'optimiser votre WAR pour un type de serveur en particulier, et d'inclure les fichiers source de votre code dans l'archive générée. Cliquez alors sur le bouton **Finish**, et votre WAR apparaîtra alors dans le dossier que vous avez spécifié sur votre disque (voir la figure suivante).



pro.war

Voilà tout ce qu'il est nécessaire de faire pour exporter votre projet dans une archive WAR !

Déploiement du projet

Contexte

Habituellement, le développement d'une application et la première phase de tests (les tests unitaires, et éventuellement d'autres tests plus spécifiques) sont réalisés depuis un serveur local, très souvent intégré à Eclipse (ou Netbeans, ou autre IDE similaire).

Ensuite, il est courant qu'un premier déploiement ait lieu sur une seconde plate-forme, dite "de qualification". Celle-ci correspond en général à une réplique plus ou moins fidèle de l'environnement de production final.

Enfin a lieu le déploiement sur la plate-forme de production, la plate-forme finale sur laquelle va tourner l'application ouverte aux utilisateurs.

Par ailleurs, il est important de noter que parfois le développement et la qualification sont effectués sur un type de serveur d'applications, par exemple Tomcat ou GlassFish installés sur une distribution Debian, puis l'application est finalement déployée sur un serveur dont la configuration est différente en production (bien souvent pour des raisons de coûts ou de simplicité), par exemple un serveur WebLogic installé sur une distribution OpenSuse.

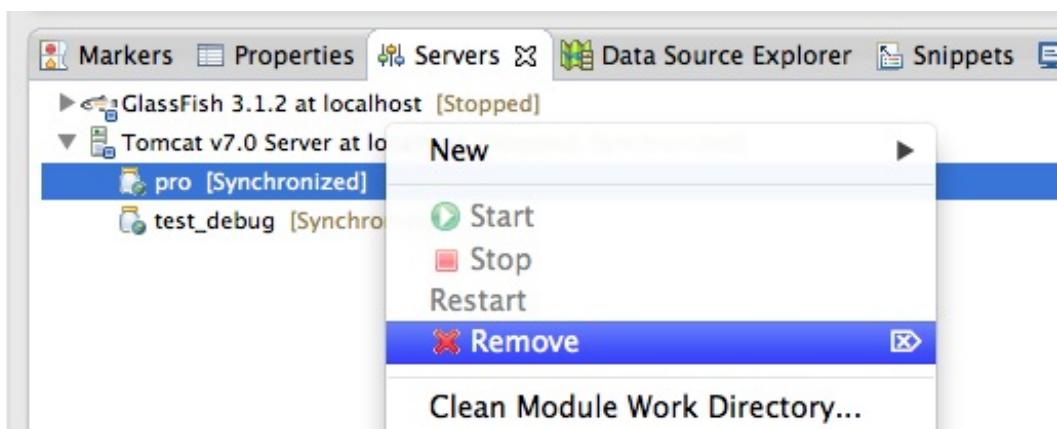


Bref, vous avez compris où je veux en venir : une même application est amenée à être utilisée dans différents contextes et sur différentes configurations, et il est donc nécessaire de pouvoir simplement installer et mettre à jour une application sur les plate-formes de qualification et de production sans contraintes.

Voilà pourquoi le système d'archives a été mis en place : en exportant votre application dans une archive WAR ou EAR, vous vous assurez de pouvoir la transporter et manipuler de manière extrêmement simple !

Mise en pratique

Pour commencer, si ce n'est pas déjà, fait retirez le projet **pro** de votre serveur Tomcat intégré à Eclipse. Pour ce faire, il vous suffit de faire un clic droit sur le projet dans le volet **Servers** et de choisir **Remove** (voir la figure suivante).



Dans la fenêtre d'avertissement qui s'ouvre alors, confirmez en appuyant sur Ok. Ainsi, nous sommes maintenant certains que notre application n'existe plus dans Tomcat. Fermez ensuite Eclipse.

Nous allons maintenant lancer Tomcat manuellement. La manipulation est différente selon le système d'exploitation que vous utilisez :

- depuis Windows, rendez-vous dans le dossier **/bin** de votre installation de Tomcat, et lancez le fichier intitulé **startup.bat** ;
- depuis un système Unix (Linux ou Mac OS), ouvrez un Terminal, déplacez-vous dans le dossier de votre installation de Tomcat, et exécutez la commande **bin/startup.sh** .

Votre serveur correctement lancé, vous pouvez dorénavant accéder à l'URL `http://localhost:8080` depuis votre navigateur pour vérifier que votre serveur fonctionne, et vous pouvez également vérifier que vous ne pouvez pas accéder à l'URL `http://localhost:8080/pro/inscription`, autrement dit que l'application **pro** n'est pas encore disponible.

Il ne nous reste plus qu'à déployer notre application sur notre server Tomcat. Pour cela, rien de plus simple : nous devons déposer notre archive WAR dans le répertoire **webapps** de Tomcat. Copiez-collez donc votre archive **pro.war** dans ce dossier, puis rendez-vous à nouveau sur l'URL `http://localhost:8080/pro/inscription` depuis votre navigateur. Vous devez alors constater que votre application fonctionne : **Tomcat a automatiquement analysé et déployé votre archive WAR !**

Par ailleurs, vous pourrez également vérifier que dans le répertoire **webapps** existe dorénavant un dossier intitulé **pro** : c'est Tomcat qui l'a créé à partir du contenu de votre fichier WAR. Vous y trouverez toutes vos classes compilées dans le répertoire **WEB-INF/classes**, vos JSP et autres éléments de design. Si vous aviez choisi d'intégrer les fichiers source de l'application à l'archive lors de l'export depuis Eclipse, alors vous y trouverez également vos fichiers source Java.

Sous GlassFish, l'opération est sensiblement la même :

- lancez votre serveur en vous rendant dans le dossier **/glassfish/bin**, et en exécutant le fichier **startserv** ou **startserv.bat** selon que vous utilisez Unix ou Windows ;
- déployez alors automatiquement votre archive en la déposant dans le répertoire **autodeploy** présent dans **/glassfish/domains/nom-de-votre-domaine/**.

Avancement du cours

Ce cours est terminé ! J'apporterai éventuellement de fines corrections par endroits lorsque nécessaire, mais le contenu et la structure du cours ne changeront plus.

Et après ?

J'espère que ce modeste cours vous ouvrira les portes vers des technologies non abordées ici, notamment :

- d'autres frameworks MVC, comme Spring ou Stripes ;
- d'autres frameworks de persistance, comme Hibernate ou MyBatis.

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 !