Les Servlets

Julien Gilli

12 mars 2001

Table des matières

Préface

Public visé

Les personnes susceptibles d'être intéressées par cet ouvrage sont :

- les programmeurs de scripts côté serveur utilisant des technologies différentes (CGI, PHP, scripts shell, etc.).
- les webmasters désirant prendre contact avec les technologies employées sur leur site.
- les débutants en programmation d'applications Web côté serveur (voir les connaissances requises à la section).
- les programmeurs Java désirant aborder un autre aspect de ce langage.

Ce projet étant effectué dans le cadre de mes études au département informatique de l'IUT de Montpellier, je m'adresse avant tout à ses étudiants. En m'adressant à ces étudiants, je touche, plus généralement, un public d'étudiants en informatique dans leur premier cycle universitaire.

Connaissances requises

Malgré les efforts que je fournirais pour être le plus clair et didactique possible, il est nécessaire pour le lecteur de posséder les connaissances de base en ce qui concerne l'informatique, et plus précisemment le langage de programmation Java, les concepts liés aux réseaux et plus particulièrement les protocoles les plus courants comme HTTP (bien que j'effectuerais un rappel lorsque nous entrerons dans le vif du sujet). Tout ce qui sort du cadre de la technologie des Servlets, c'est à dire qui n'a pas de répercution sur le comportement des Servlets en particulier, ne sera pas expliqué. Par exemple, je ne décrirais pas la syntaxe utilisée dans le code source qui sera fourni en tant qu'exemple à plusieurs reprises.

Objectifs de cet ouvrage

A la suite de la lecture de cet essai, et après l'utilisation de l'application de démonstration, le lecteur devra disposer de suffisamment de connaissances sur le sujet des Servlets pour avoir une idée précise de leur fonctionnement interne,

de leur mise en oeuvre, de leurs avantages et de leurs inconvénients. Mon objectif est qu'un étudiant puisse, avec la seule aide de cette documentation et de la documentation de référence sur la bibliothèque de classe spécifique aux Servlets, développer des applications complètes à base de Servlets (du moment qu'il satisfait les critères de connaissance vus dans la section précédente).

Ce document n'est en aucun cas une promotion ou une critique négative sur les Servlets. Je tenterais d'être le plus objectif pour fournir les informations pertinentes, en essayant de ne rien oublier.

Cependant, bien que je souhaiterais pouvoir le faire, il m'est impossible de traiter en détail des sujets liés aux Servlets mais qui ne sont pas spécifiques à celles-ci. Par exemple, je décrirais l'interfaçage entre les gestionnaires de bases de données et les Servlets, mais je n'expliquerai pas l'utilisation un gestionnaire de bases de données, et encore moins son optimisation, à moins que celle-ci soit spécialement utile pour le fonctionnement des Servlets. Un autre exemple : je traiterai de la sécurité liée au Servlets mais je ne ferais pas une description complète du protocole HTTPS (utilisant SSL). Cependant, je m'efforcerai autant que possible de donner des possibilités d'obtention d'informations relatives à ces sujets, de cette manière le lecteur pourra consulter une documentation spécifique de qualité bien meilleure que celle que j'aurais pu fournir en quelques pages.

Conventions utilisées

Les différentes typographies utilisées dans cet ouvrage sont le suivantes :

- une typographie ordinaire pour le texte.
- *une mise en valeur* pour les termes importants rencontrés pour la première fois.
- `une typographie différente` pour le code source des exemples.
- un **texte en gras** pour les commandes entrées (raccourcis clavier, commandes de menu, etc.)

Les outils utilisés pour matérialiser cet essai sont Lyx, un frontal pour l'ensemble de macros pour le langage Tex nommées Latex , Xfig et Dia pour la conception des figures, et quelques scripts perl pour convertir le document au format Lyx en d'autres formats (pdf, HTML, Latex et ASCII notamment).

Vous pouvez vous procurer ces logiciels sur le site Web freshmeat à `http://www.freshmeat.net/`. Tous ces outils de grande qualité sont disponibles gratuitement et sous licence GPL (pour Gnu Public Licence)[1]

[1] Il est important de bien lire un exemplaire de cette license à `http://www.gnu.org/` qui est à l'origine de nombreuses petites révolutions et a permit la création de quelques bijoux.

Chapitre 1

Présentation

1.1 L'origine des Servlets

Suite à l'incroyable essor pris par le développement d'applications Web côté serveur, Sun Microsystems se devait de ne pas rester à l'écart, et a donc produit un kit de développement Web, comprenant le serveur Web java et un ensemble de classes.

Cet ensemble de classe correspond à ce que l'on appelle les Servlets et le serveur Web java, programmé pour une grande partie grâce aux Servlets, fut un des premiers serveurs Web à faire fonctionner les Servlets.

Rapidement, grâce aux qualités intrinsèques au langage Java (que je mettrais en valeur au fur et à mesure de l'ouvrage) et aux spécificités du fonctionnement des Servlets, ces dernières sont parvenues à attirer l'attention des développeurs.

C'est bien entendu Sun Microsystems, avec son département logiciel Java Soft (comme pour le langage Java), qui a écrit la première version de la norme des Servlets, et qui continue à le faire de manière relativement ouverte, acceptant d'étudier toute propoposition raisonnable venant de n'importe qui. La dernière version de cette norme est actuellement la 2.2[1], et n'est pas supportée par tous les serveurs de Servlets.

Bien que les Servlets constituent une extension majeure du langage Java, leur API n'est pas intégrée à la version standard de la plate-forme Java, mais à l'édition entreprise `http://java.sun.com/j2ee/`.

1.2 Les technologies utilisées

La technologie des servlets n'est qu'un ensemble de classes. Elles ont besoin pour fonctionner convenablement, d'une machine virtuelle Java et de l'ensemble des autres classes intégrées à l'API standard du langage Java .Une servlet est une

[1] Vous pouvez télécharger les spécifications de cette version de l'API des servlets sur `http://java.sun.com/products/servlet/download.html`.

application développée dans un contexte client-serveur, et vient se greffer sur des applications existantes afin de les étendre ou des les implémenter différemment. Une servlet n'existe pas par elle-même, mais étend des applications utilisant des protocoles tels que HTTP, SMTP ou FTP par exemple.

1.3 La place des Servlets

1.3.1 A quoi ça sert ?

Les servlets permettent théoriquement d'étendre n'importe quel type de serveur. Il est donc théoriquement possible de développer toute sorte d'application susceptible de bénéficier à une application faisant office de serveur. Cependant dans la réalité il n'est pas possible, à ma connaissance, de développer des applications pour autre chose qu'un serveur Web. Ceci est du au simple fait qu'il est nécessaire de développer une passerelle entre la machine virtuelle Java et le serveur à étendre et que cela constitue un travail difficile et probablement inintéressant pour des raisons que j'évoquerai plus tard. De plus, il n'est pas certain que cet effort soit récompensé par les développeurs.

En pratique, les Servlets permettent donc exclusivement le développement d'applications Web côté serveur., ceci de manière à pouvoir fournir un contenu dynamique aux visiteurs de sites Web.

1.3.2 Public visé

La mise en place de cette technologie est gratuite et accessible à tous facilement. Le langage Java n'étant pas réputé pour son austérité, tous les ingrédients sont réunis pour en faire une solution de masse.

En effet, les programmeurs peuvent créer des applications utiles en très peu de temps et ne sont limités en aucune manière par les possibilités très étendues du langage utilisé. De plus, étant donné l'orientation objet de ce dernier, et l'utilisation de celle-ci comme support pour la diffusion d'ateliers de génie logiciel (comme Objectering), les Servlets sont adaptées à la programmation de grandes applications. Ces dernières bénéficient alors de la réutilisabilité des programmes écrits en Java et accélèrent le développement.

D'autres caractéristiques du langage Java favorisent la diffusion des Servlets dans tous les domaines : sa portabilité, sa stabilité et sa grande présence dans les milieux universitaires (chaque année, ce sont environ deux millions de développeurs Java qui sont formés).

Enfin, la gratuité des solutions permettant de faire tourner des Servlets sur une machine de manière performante est également un facteur de diffusion non négligeable.

Pour résumer, les propriétés du langage Java et des solutions de mise en place des Servlets permettent une diffusion de cette technologie dans les milieux professionnels autant que chez les utilisateurs amateurs. C'est d'ailleurs en réponse

à cet attrait de la part des programmeurs que de nombreux hébergeurs ont décidé de proposer une offre concernant les Servlets. Une liste d'une partie de ces hébergeurs est présente à l'adresse suivante : `http://staging.servlets.com/isps/servlet/ISPViewAll`.

1.3.3 Les possibilités

Comme je l'ai dit précédemment, l'énorme bibliothèque de classes permet au programmeur de Servlets une grande latitude quant à la fonction que remplira son application.

D'autres caractéristiques comme les possibilités évoluées de connexion aux principaux SGBD du marché[2], la persistance du processus de chaque Servlet, l'intégration des EJB (pour Enterprise Java Beans) ou encore la gestion de la sécurité permettent aux Servlets de prendre en charge des projets tels que des sites de commerce électronique, des plate-formes bancaires, des sites communautaires ou des outils de configuration de serveurs.

Je ne le répéterai jamais assez, les servlets ne sont pas exclusivement destinées à créer des applications Web, elles peuvent très bien étendre d'autres types de serveurs. On peut donc raisonnablement envisager la création de nouvelles commandes pour un serveur FTP, ou d'une application de filtrage de courriers électroniques directement présente sur le serveur et non sur le client. Ces applications ne sont pas réalisables dès aujourd'hui, mais elles le seront sans doute prochainement lorsque les fournisseurs de serveurs utilisant un autre protocole que HTTP adopteront la plate-forme J2EE[3].

1.4 Quels sont les acteurs sur le marché ?

Depuis leur création en 1996, de plus en plus d'acteurs majeurs de l'industrie de l'informatique ont proposé une offre autour des Servlets.

Tout d'abord en ce qui concerne les serveurs d'applications. Ces solutions sont très fortement liées à tous les domaines d'activité de l'entreprise. De nombreux composants "métiers" sont utilisés (comme les EJB qui sont les Enterprise Java Beans), et des critères comme la fiabilité, le support technique et l'adéquation avec les contraintes professionnelles de l'entreprise sont primordiaux. Il est donc compréhensible que d'importants fournisseurs d'applications aient investi ce secteur. Ces fournisseurs sont IBM (avec WebSphere), BEA (avec WebLogic), Oracle (avec Oracle Application Server) et iPlanet (avec iPlanet Application Server). Ces serveurs utilisent des moteurs de Servlets et des serveurs Web parfois indépendants. Par exemple IBM a développé un serveur Web pour WebSphere qui se base sur le serveur Web Apache.

[2] Une liste des SGBDs suportés est disponible à l'URL suivante : `http://industry.java.sun.com/products/jdbc/drivers`.

[3] J2EE signifie Java 2 Enterprise Edition, c'est à dire l'édition entreprise de la deuxième version de la plateforme Java. L'édition entreprise de la plate-forme Java introduit des possibilités comme la gestion des bus de partages d'objet, les transactions ou encore les Beans Java d'entreprise. Pour de plus amples renseignements sur les Beans Java, consultez le tutoriel Java à l'URL suivante : `http://java.sun.com/docs/tutorial/`.

Ensuite viennent les moteurs de Servlets, qui ne fonctionnent pas tous suivant le même modèle. Ce sont des applications qui implémentent toutes une version de la norme de l'API des Servlets, ils ne se différencient donc pas selon leurs fonctionnalités mais selon leurs performances. Les principaux moteurs de Servlets sont Tomcat (de la fondation Apache), Resin (de Caucho), JServ (de la fondation Apache également), Allaire avec JRun.

Enfin ces moteurs de Servlets sont souvent rattachés à des serveurs Web (car c'est à un serveur Web que sont destinés les requêtes provenant d'un navigateur dans de nombreux cas). Étant donné qu'un site Web ne se base pas exclusivement sur les Servlets, les serveurs Web les plus utilisés pour fournir d'autres services sont en général choisis. Ces serveurs sont Apache (de la fondation Apache), iPlanet Web Server (de iPlanet) et IIS (de Microsoft).

D'autres entreprises très présentes dans le secteur du développement traditionnel fournissent des outils aux développeurs désirant obtenir une forte intégration avec leur serveur supportant les Servlets. Je pense par exemple à Borland avec JBuilder, Macromedia UltraDev, et à PowerJ de Sybase. Le reste des outils est développé par les acteurs cités ci-dessus (Oracle avec JDeveloper, IBM avec Visual Age, Allaire avec JRun Studio).

On ne peut que constater l'absence de Sun et de Microsoft quant aux outils permettant le développement et le déploiement de Servlets. En ce qui concerne Sun, ce n'est pas tout à fait vrai. En effet, ils fournissent un moteur de Servlets intégré à un serveur Web nommé Java Web Server mais il constitue plus une solution pionnière qui a permis de disposer d'une première plate-forme que d'une solution destinée à être utilisée en production. Ils proposent également un outil de développement appelé Forté (anciennement Net Beans) qui, dans sa version entreprise, supporte les Servlets. Cet outil n'est pas véritablement utilisé à cause de sa lourdeur mais fut quand même le premier outil RAD permettant le développement de Servlets. Ils ne proposent donc ces outils que dans un but d'introduction aux concepts qu'ils proposent, ils ont par ailleurs fort à faire à normaliser l'API des Servlets tout en développant son implémentation et ses fonctionnalités.

Quant à Microsoft, il suffit de songer à leur mode de diffusion pour comprendre leur position. Les produits de Microsoft sont diffusés en partie car ils constituent un standard imposé par le caractère propriétaire de leurs applications. Ils fournissent des applications en étroite relation avec leur système : on choisit le système de Microsoft car on ne peut se passer de leurs applications et vice-versa. Avec Java, on peut disposer d'une plate-forme pouvant apporter des solutions globales (comme les serveurs d'applications) et ceci de manière portable : l'informaticien et l'utilisateur sont indépendants de l'architecture matérielle et logicielle. On peut donc très bien choisir d'autres produits que ceux de Microsoft. Afin de résister à la déferlante Java, Microsoft a donc annoncé le développement d'une plate-forme similaire : la plate-forme .NET. C'est un ensemble d'outils pour le développement, le déploiement et l'utilisation d'applications distribuées, qui met en valeur l'utilisation des réseaux (locaux ou Internet). Cette solution reprend de nombreuses caractéristiques de Java comme le langage C# dont la syntaxe est étonnament similaire à celle de Java, la présence d'une machine virtuelle qui interprète un langage intermédiaire, le caractère distribué des applications (un utilisateur peut utiliser une application distante) et

d'autres encore. On peut donc voir la plate-forme .NET comme un concurrent à la plate-forme J2EE[4].

Bien entendu cette situation n'est pas figée et elle sera probablement amenée à changer au fur et à mesure de l'arrivée des nouveaux acteurs et des départs des anciens, qui seront peut être attirés par d'autres plate-formes.

1.5 Les solutions de substitution

D'autres solutions que les Servlets permettent de réaliser des applications similaires. En effet, les développeurs n'ont pas attendu l'arrivée de ces dernières pour automatiser le traitement de données côté serveur en vue de les communiquer à des utilisateurs . Voici une liste des principales techniques pouvant se substituer aux Servlets :

- CGI (pour Common Gateway Interface) : c'est un ensemble de règles permettant de faire communiquer un serveur Web avec des applications externes. Cette possibilité a donc été rapidement utilisée pour proposer un contenu dynamique. Les scripts CGI peuvent être écrits dans n'importe quel langage (comme le C, le Perl, le Python, etc.).
- PHP (pour Pretty Hypertext Processor) : c'est un langage permettant d'écrire des scripts qui seront ensuite interprétés par un logiciel (on appelle PHP la réunion des deux). C'est un logiciel libre qui en est aujourd'hui à la version 4 et qui est un des plus employé. C'est une technologie très portable.
- ASP (pour Active Server Pages) : la solution proposée par Microsoft. C'est une norme qui permet le développement d'applications Web côté serveur sur la plate-forme Microsoft (système d'exploitation Windows et serveur Web IIS). Comme la norme CGI, ASP permet de développer des scripts en utilisant le langage de programmation de son choix. Cela n'est que théorique, et malgré le développement anecdotiques de support ASP pour le langage Perl en utilisant Apache par exemple, l'indépendance vis à vis du langage, du serveur Web et du système d'exploitation n'est pas une réalité. ASP dans sa version 4 (nommée ASP+) est intégrée à la plateforme .NET (voir 1.4 page ci-contre).
- les solutions propriétaires comme Cold Fusion ou WebDev (de PC Soft) qui comprennent tous les outils de la chaîne de développement d'applications Web côté serveur sont assez utilisées mais possèdent les inconvénients de toute architecture propriétaire. Cela dit ces solutions proposent également leur lot d'avantages, comme la gestion de flux de texte très évoluée pour Cold Fusion et le développement rapide pour WebDev.

Je ne compare pas ces possibilités aux Servlets pour l'instant car je ne vous ai pas donné assez d'éléments de réflexion, mais sachez que cela sera fait ultérieurement. Je ferai parfois, et tout au long de cet ouvrage, quelques comparaisons ponctuelles sur des points précis entre les Servlets et certaines des solutions sus-citées. Sachez seulement pour l'instant que toutes ces technologies ne sont

[4]Récemment un procès opposant Sun Microsystems à Microsoft a amené ce dernier à reverser vingt millions de dollars à Sun pour avoir utilisé le label "Java-compatible" pour des produits dont la base était la plateforme Java mais qui étaient modifiés par Microsoft de manière non compatible avec les idées de Sun (extensions spécifiques à Windows et propriétaires notamment).

utilisables que par l'intermédiaire du protocole HTTP, ce qui n'est pas le cas des Servlets, même si c'est cet aspect que je développerai en majorité.

Chapitre 2

Introduction au développement

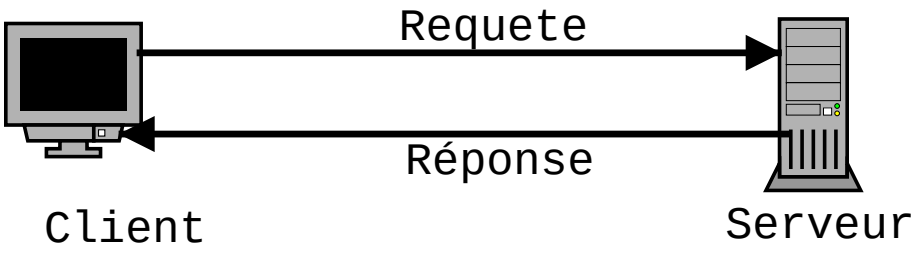
Maintenant que les présentations sont faites, je vous invite à manipuler les Servlets. Cette introduction au développement des Servlets vous donnera les connaissances de base sur les technologies constituant les fondements sur lesquels reposent les Servlets (comme le protocole HTTP). Elle décrira en détail l'installation du moteur de Servlets Tomcat couplé au serveur Apache après avoir fait le tour des solutions et justifié mon choix et vous expliquera précisemment sous quelle forme se présente une Servlet, comment et avec quels outils les développer.

2.1 Les bases des applications Web côté serveur

2.1.1 Les applications client-serveur

Le fonctionnement d'une application client-serveur peut être décrit de la manière suivante.

Un programme s'exécutant sur une machine logique proposant un support des connexions vers d'autres ordinateurs au travers d'un réseau offre ses services. Sur une autre machine logique , proposant elle aussi un support des connexions vers d'autres ordinateurs au travers d'un réseau, s'exécute un programme qui émet des requêtes vers des machines faisant office de serveur, afin d'utiliser les services proposés.



Un programme serveur peut gérer, en général, plusieurs requêtes à la fois provenant de clients différents ou identiques. Un programme client peut, en général, utiliser plusieurs connexions vers des serveurs différents ou identiques en même temps.

Ces deux machines logiques (car le programme client peut fonctionner sur la même machine physique que le programme serveur) utilisent un ensemble de protocoles pour communiquer entre elles. Ces protocoles sont un ensemble de règles à respecter pour pouvoir communiquer correctement selon le but recherché (transmission d'informations confidentielles, transmission rapide d'informations peu importantes etc.).

FIG3

Ces protocoles se placent à divers niveaux de la couche du modèle OSI qui définit l'architecture logique du support des communications au travers d'un réseau sur la plupart des machines. Ces couches vont des plus abstraites et orientées "applications" en haut aux plus concrètes et orientées "implémentation physique" en bas.

FIG4

C'est ainsi que les couches les plus basses du modèle OSI permettent de gérer toutes les communications au travers d'un réseau, alors que ce sont les couches les plus hautes qui fournissent des différences notables au programmeur et à l'utilisateur d'applications de haut niveau.

Ces couches les plus hautes définissent donc des protocoles spécifiques à chaque type d'applications que l'on peut trouver sur un réseau : transfert de fichiers (FTP), envoi et réception de courriers électroniques (SMTP), consultation de sites Web (HTTP) etc..

Le client et le serveur dialoguent donc à l'aide des mêmes protocoles, mais d'un côté le serveur fournit les services, de l'autre le client en bénéficie. Un serveur, grâce à la séparation des protocoles orientés "application" peut fournir le type de service qu'il veut. Il n'est pas obligé de fournir tous les services existants (simplement la consultation de sites Web par exemple), mais il peut le faire.

Un client peut en même temps faire office de serveur. Les communications entre ordinateurs ne sont pas unidirectionnelles.

FIG5

2.1.2 Le protocole HTTP

Afin d'illustrer et de mieux comprendre comment se déroule une communication entre un client web (qui peut être un navigateur ou toute autre application utilisant le protocole HTTP) et un serveur Web, je vais tenter d'expliquer un peu plus en détail le fonctionnement du protocole HTTP.

Le serveur se place en mode d'écoute sur un port bien défini, qui est généralement le port 80, même si cela n'a rien d'obligatoire. A chaque requête reçue de la part du client, il crée un nouveau processus[1] qui va prendre en charge cette requête. Le dialogue peut alors débuter, et le client doit alors émettre sa requête, qui doit être conforme au protocole HTTP.

Ces requêtes sont transmises au serveur web via la connexion établie entre le client et le serveur et prennent la forme de données classiques (textuelles). Il existe un nombre important de requêtes différentes, mais en réalité beaucoup de requêtes sont très peu utilisées. L'annexe A décrit tous les types de requêtes disponibles dans le protocole HTTP avec une courte description. Le client envoie donc sa requête, le serveur l'analyse et renvoie une réponse au client (adaptée à sa requête) au travers de la même connexion. La connexion est ensuite coupée. Le client doit émettre des requêtes conformes à la version du protocole HTTP implémentée par le serveur. En général les serveurs récents (comme Apache) implémentent la version 1.1 tandis que certains serveurs anciens ne supportent que la version 1.0. Les serveurs supportant la version 1.1 du protocole supportent aussi, en général, la version 1.0. La version du protocole est précisée dans l'entête de la requête émanant du client et dans la réponse faite par le serveur.

Afin de tester les comportements décrits dans les exemples que je donnerai, vous pouvez utiliser l'outil telnet (disponible sur tous les systèmes d'exploitation dignes de ce nom) de la manière suivante :

telnet adresse_ip_serveur_web 80

où adresse_ip_serveur est l'adresse ip du serveur web cible (je pense que vous le saviez dejà;-)). Il ne vous reste plus qu'à entrer la requête à l'invite qui vous est proposée.

La partie principale de la requête en ce qui concerne le client est la commande, placée sur la première ligne de la reqûete. Elle détermine l'action à effectuer. Ces commandes sont parfois accompagnées d'arguments qui précisent l'objet sur lequel porte la commande. L'action la plus courament utilisée est la récupération d'un document en utilisant la commande GET ou POST (GET et POSt sont deux *méthodes* HTTP différentes pour effectuer la même action). Cette commande prend comme argument la ressource à récupérer, c'est à dire le chemin vers le fichier à récupérer (ce n'est pas un chemin absolu, mais relatif à la racine du serveur). Par exemple :

```
GET /index.html HTTP/1.0
```

permet de récupérer le fichier index.html qui est situé à la racine du répertoire d'installation du serveur en utilisant la version 1.0 de HTTP.

[1] Il peut également créer un nouveau Thread, cela dépend de l'implémentation du serveur Web. Par exemple Apache crée de nouveaux processus sous Unix et de nouveaux Threads sous Windows.

De manière générale, les clients placent une entête dans leur requête, à la suite de la commande, décrivant par exemple qui ils sont (le nom et le numéro de version pour un navigateur par exemple) et ce qu'ils peuvent accepter (des images, du texte au format HTML pour le même type de logiciel par exemple). Certaines entêtes sont optionelles et d'autres obligatoires, cela dépend de la requête et de la configuration du serveur Web à qui la requête est transmise. Vous devrez par exemple ajouter une entête spécifiant un identifiant et un mot de passe si le serveur Web est configuré de telle manière qu'il n'accepte pas les personnes inconnues. Par exemple :

GET / index.html HTTP/1.0

Accept : text/html, image/gif

User-Agent : Mozilla/4.0 (compatible; MSIE 4.0; Linux X11 2.2.17)

nous indique que le client à l'origine de la requête est le navigateur netscape (basé sur Mozilla, d'où l'apparition de ce nom) tournant sous Linux et qu'il accepte le texte au format HTML et les images au format GIF. La requête d'un client Web doit être suivie de deux sauts à la ligne pour être prise en compte.

Une fois la requête du client analysée (entête et corps de la requête), le serveur dispose de tous les éléments pour produire une réponse. Le premier élément de la réponse est appelé l'état qui spécifie la version du protocole HTTP utilisée, un code d'état et une courte description de cet état. L'entête de la réponse vient après cette ligne d'état. Celle ci décrit par exemple le type de données qui est envoyé (texte au format ASCII, au format HTML, images au format gif, etc.). Le serveur peut ensuite envoyer les données au navigateur qui les analysera correctement en se conformant aux directives données par l'entête de la réponse du serveur. Par exemple :

HTTP/1.0 200 OK

Server : Apache/1.3.14 mod_php4

Content-type : text/html

nous indique que la requête peut être traitée avec succès (code d'état 200 et description du code à "OK") et qu'elle est traitée par le serveur Apache dans sa version 1.3.14. On sait également que la réponse contient exclusivement du texte au format HTML.

Une fois cette entête envoyée au client, le serveur peut débuter l'envoi des données (le fichier index.html si on se base sur la requête en exemple au dessus) en séparant l'entête et les données de deux lignes vides. Ces deux lignes vides sont indispensables indispensables.

Je le répète, après l'envoi des données au client, la connexion est interrompue, même si le client est susceptible d'émettre une requête pour accéder à une ressource sur ce même serveur immédiatement après. On dit que le protocole HTTP n'est pas orienté session : il n'y a aucun moyen d'identifier un utilisateur au cours d'une série de connexion à un site Web. Or il est fréquent que des utilisateurs effectuent une série d'actions logiques destinées à atteindre un but bien précis (acheter un produit sur un site marchand par exemple). Il est donc problématique d'identifier correctement cette suite logique avec les seuls outils mis à disposition par le protocole HTTP. Certains outils des Servlets que nous

verrons dans le chapitre consacré au suivi de session sont là pour corriger ce manque.

2.2 La mise en place du serveur

Vous avez désormais acquis assez de connaissance à propos du protocole HTTP et du fonctionnement des applications Web côté serveur pour passer à la pratique. Afin de pouvoir tester vos premières Servlets, vous devez installer le serveur Web et le moteur de Servlets qui les exécuteront. Je vais tenter dans un premier temps de vous donner un tour d'horizon des diverses solutions existantes, puis vous faire part de mon choix et des causes de ce choix, et enfin je vous guiderais au travers de l'installation du serveur.

2.2.1 Les différents types de solutions

Tout d'abord, définissons les différents types de serveurs. Il existe trois types différents de moteurs de Servlets :

1. les moteurs de Servlets indépendants.
2. les moteurs de Servlets embarqués.
3. les moteurs de Servlets externes.

Les moteurs de Servlets indépendants constituent une partie intégrante du serveur Web. Pour qu'une telle chose soit possible, il faut en principe que le serveur Web soit développé en Java. Un exemple de ce type de serveurs est le Java Web Server, ou encore Tomcat (dont nous analyserons la configuration en détail ultérieurement).

Les moteurs de Servlets embarqués sont une combinaison d'un ajout à un serveur Web et d'une implémentation de l'API Servlet. L'ajout (le plugin) au serveur Web permet d'aiguiller les requêtes venant des clients et qui concernent des Servlets vers une machine virtuelle contenue dans le processus du serveur Web. Cette machine virtuelle est exécutée dans un thread séparé ce qui implique le fonctionnement du serveur Web en multi-threads, ce qui n'est pas le cas de tous les serveurs Web (par exemple Apache sous Unix). Cette configuration apporte de bonnes performances, car les changements de contexte sont moins coûteux, mais est limitée en possibilité d'extensions (un serveur supportant un très grand nombre de requêtes au même moment aura du mal à pouvoir répondre à ces dernières).

Les moteurs de Servlets externes sont une combinaison entre un plugin "greffé" au serveur et une machine virtuelle tournant à l'extérieur de celui-ci. Afin de communiquer entre eux, le plugin et le processus associé à la machine virtuelle utilisent un mécanisme de communication inter-processus tel que les sockets TCP/IP. Si une requête passée au serveur Web concerne les Servlets, celui-ci passe la requête au moteur de Servlets en utilisant le mécanisme sus-cité. Un des inconvénients de cette méthode est la diminution des performances. Par contre, ce type de serveur est en général plus stable (un arrêt du serveur Web n'agit pas

sur le moteur de Servlets et vice-versa) et plus extensible. De plus, étant donné que la machine virtuelle est extérieure au serveur Web, elle peut tourner sur n'importe quelle machine, ce qui vous permet de faire tourner le serveur Web sur une machine différente de celle sur laquelle est lancé le moteur de Servlets. Vous pouvez également mettre en place plusieurs machines virtuelles pour un même serveur Web[2]. En bref, la souplesse que l'on gagne peut être intéressante à de nombreux points.

En général, les personnes désirant fournir un support pour les Servlets sur leur serveur fournissent déjà un support pour d'autres technologies de développement Web côté serveur, ou proposent tout simplement l'hébergement de pages statiques. Il est clair qu'un moteur de Servlets, même s'il peut faire office de serveur Web dans le cas où il peut tourner en mode indépendant (comme pour Tomcat ou le Java Web Server), est beaucoup moins performant que les produits spécialisés lorsqu'il faut fournir un contenu différent des Servlets (pages statiques, fichiers, etc.). On comprend donc facilement qu'il vaut mieux choisir un moteur de Servlets "greffable" sur un serveur Web existant, afin de rester performant en ce qui concerne le contenu statique (c'est le serveur Web qui le gère) et le contenu dynamique faisant appel à des Servlets (c'est le moteur de Servlets qui le gère). Cela nous laisse donc le choix entre la solution un et deux (embarqués ou externes) Bien sur, il est tout à fait possible de travailler avec des moteurs de Servlets indépendants à des fins de développement, je les déconseille simplement pour de la production.

Venons-en aux serveurs d'applications. J'ai dit précédemment qu'ils étaient souvent un assemblage entre un serveur Web existant, un moteur de Servlets existant et des outils orientés "métier" en plus, comme l'incorporation des EJBs et d'autres outils spécifique à une activité industrielle (gestion des transactions, relation client, etc.). ce sont des outils très pointus, qui coûtent souvent très chers, et qui requièrent de multiples compétences. Leur utilisation et leur configuration est similaire aux solution plus classiques, avec souvent des outils de configuration plus simples à utiliser pour les taches courantes, mais en ce qui concerne le développement strict de Servlets, ils n'apportent quasiment rien (à part des optimisations des performances ou de la fiabilité dans des environnements bien précis, souvent des gros systèmes comme les S/390). On voit très bien que les aspects intéressants de ces solutions sortent du cadre de mes compétences et de cet ouvrage, et c'est pour cette raison que je les écarterais de mon choix. Pour conclure à propos de ces solutions, elles sont souvent moins souples à utiliser que des applications plus classiques en raison de leur complexité et de leur course à l'utilisation facile. Les problèmes rapportés sur les listes de diffusion ayant trait à la configuration de moteurs de Servlets concernent la plupart du temps ce type de serveurs.

Il apparaît que le type de serveur réunissant le plus d'avantages est le moteur de Servlets externe relié à un serveur Web performant via un plugin. En effet, comme on a pu le voir, c'est une solution fiable, souple et assez performante. De plus, moyennant quelques efforts de configuration, ce type de serveur peut tout à fait inclure un nombre très important de fonctionnalités comme les EJBs, ce qui leur donne accès aux composants métiers si chers aux serveurs d'applications, ou le support de plusieurs machines virtuelles.

[2] Je reviendrais plus en détail sur cela lors de la configuration de la solution choisie.

Maintenant que nous avons choisi (même si je vous ai aidé) le type de serveur à utiliser, choisissons lequel utiliser dans cette catégorie.

2.2.2 Le choix de la solution

Le choix d'un serveur dépend essentiellement, en dehors des performances concernant le service de contenu autre que des Servlets, de deux critères : la version de l'API Servlet supportée et la version des JSP supportée. En effet, bien que les comparaisons de performances entre solutions soient abondantes, elles ne signifient rien si elles ne concernent pas directement l'application que vous voulez faire fonctionner. Plus la version de l'API supportée est récente, plus vous pourrez bénéficier de fonctionnalités utiles (par exemple la version 2.0 de l'API des Servlets ne supporte pas le RequestDispatcher). Le raisonnement est similaire en ce qui concerne les JSPs. Afin de bénéficier d'une vision claire de ces deux critères, je vous invite à consulter l'URL suivant : `http://java.sun.com/products/servlet/industry.html`. Ici, le système d'exploitation choisit n'entre pas en compte pour le choix du moteur de Servlets car, étant donné que ce type de serveur est (en général) programmé entièrement en Java, tout système d'exploitation disposant d'une implémentation de la machine virtuelle Java est censé fonctionner. Cependant, étant donné que nous avons choisis de coupler ce moteur de Servlets à un serveur Web, nous devrons prêter attention aux serveurs Web supportés, afin que nous puissions utiliser ce couple avec le système d'exploitation de notre choix.

Vous pouvez voir que quelques serveurs externes (c'est à dire qui peuvent être interfacés avec un serveur Web, à ne pas confondre avec indépendant) correspondent à ces deux critères. Le choix que je vais proposer ici ne dépend que de préférences auxquelles vous n'adhérerais pas forcément, mais je tenterais de justifier le rejet d'autres solutions (comme je l'ai fait pour le choix du type de serveur).

Tout d'abord WAIColRunner doit être utilisé conjointement à un serveur Web Netscape Enterprise Server ou Netscape Fast Track Server (le dernier est une version allégée du premier). Cela limite considérablement le choix quant au serveur Web, même si celui-ci est très largement diffusé au sein des entreprises. C'est un moteur de Servlets payant dès que l'on veut obtenir un support technique. En ce qui concerne JRun de Allaire, il propose, grâce à JRun Studio une suite intégrée séduisante pour le développement de Servlets (débogage à distance, développement rapide, etc.,) et propose quelques caractéristiques intéressantes comme la compatibilité avec la majorité des serveurs Web existants mais il est payant (le prix varie entre 495 dollars et 18 395 dollars). Il reste alors Resin de Caucho et Tomcat de la Fondation Apache. Le premier possède un nombre impressionnant d'avantages : il supporte les principaux serveurs Web du marche (iPlanet, IIS, Apache), est doté d'outils intéressants comme le support de XSL et XML pour gérer les modèles de documents (entre autres) et supporte même la version 2.3 de l'API des Servlets (qui n'est qu'au stade de proposition)[3]. Cependant l'utilisation est payante si elle entre dans un cadre commercial. Le dernier qui, vous l'aurez compris, est le moteur de Servlets sur

[3]Je vous accorde que ceci peut ne pas être utile pour une utilisation en production, mais montre la réactivité des développeurs de Resin.

lequel je porte mon choix, est Tomcat de la fondation Apache. Je l'ai choisi car il tourne sur un nombre très important de systèmes d'exploitation différents, qu'il est très bien intégré au serveur Apache (mais aussi à d'autres comme IIS), et qu'il est distribué dans les termes d'une licence en faisant en logiciel libre (il est donc disponible gratuitement). De ces caractéristiques découle qu'il est très simple de se documenter sur ce moteur de Servlets, de trouver des personnes sachant le manipuler et que son développement est très actif[4]. De plus, il n'est pas nécessaire de démontrer l'efficacité du serveur Web Apache auquel on l'associe en général.

Nous disposons maintenant d'une solution souple et performante basée sur le moteur de Servlets Tomcat et le serveur Web Apache. Je vais vous guider maintenant dans l'étape d'installation et de configuration de cette solution, en me concentrant sur ce qui peut vous être utile et en vous laissant consulter la documentation livrée avec le serveur pour les détails.

2.2.3 L'installation de Tomcat et Apache

Tomcat et Apache sont, comme nous l'avons dit précédemment, tous deux disponibles pour un nombre important de systèmes d'exploitation différents. Il m'est impossible de décrire le procédé d'installation de ces deux applications dans tous les environements possibles. Sachez seulement que l'installation sur des systèmes de la famille Unix ne changera que très peu au pire des cas, et que l'installation sous Windows ne différera qu'au niveau des fichiers à installer (il faudra télécharger les binaires pour Windows et non les sources pour Unix) et ne nécessitera pas de compilation. Je vais donc vous décrire l'installation de Tomcat et Apache sous un système Linux, en particulier une distribution Debian[5] dans sa version 2.2. Il est admis ici que vous disposez d'un compilateur de programmes écrits en C qui fonctionne et qui est compatible avec gcc. Pour vous en assurer, entrez la commande :

gcc –version

qui devrait afficher la version de gcc installée sur votre système.

L'installation de l'environnement de développement Java

Avant de procéder à l'installation de Tomcat, il est nécessaire d'installer un environnement de développement Java. Pour cela, je vous conseille d'utiliser le JDK1.3 fournit par Sun. Si vous disposez d'un kit de développement Java d'une version ultérieure ou égale à la version 1.1, vous pouvez ignorer cette étape (bien que pour des raisons évidentes de performances et de fonctionnalités je vous conseille d'installer la dernière version du kit de développement). Il est téléchargeable au format tar.gz (indépendant de la distribution) à l'URL suivant : `http://java.sun.com/j2se/1.3/download-linux.html`. Une fois ce

[4] Vous pouvez accéder à la communauté des développeurs de la plate-forme Java-Apache à l'URL suivant : `http://java.apache.org/`

[5] La distribution de Linux la plus libre, vous pouvez obtenir plus de renseignements sur `http://www.debian.org/`.

fichier téléchargé, vous devez le décompresser et le désarchiver dans le répertoire de votre choix (par exemple /usr/local/jdk1.3) grâce à la commande suivante :

tar xfvz j2sdk-1_3_0-linux.bin -C /usr/local

. Une fois ceci effectué, déplacez vous dans le répertoire $JAVA_HOME/bin où $JAVA_HOME est le répertoire dans lequel le JDK est installé (dans notre exemple : /usr/local/jdk1.3/bin) de la manière suivante :

cd /usr/local/jdk1.3/bin

Lancez alors la commande :

./javac

Si vous obtenez un message vous indiquant toutes les options disponibles pour le compilateur Java, c'est gagné. afin de faire bénéficier toutes vos applications susceptibles d'utiliser Java (comme Tomcat) de l'environnement installé, je vous conseille d'ajouter les outils du langage dans votre PATH comme ceci :

export PATH=$PATH :/usr/local/jdk1.3/bin

en considérant toujours que vous avez installé le JDK dans /usr/local/jdk1.3. Maintenant vous pouvez utiliser les outils du langage (le compilateur, le visionneur d'applets, etc.) où que vous soyez dans l'arborescence de votre système de fichiers.

Maintenant que l'environnement de développement pour Java est installé, nous allons passer à l'installation du moteur de Servlets Tomcat.

L'installation de Tomcat

Nous allons maintenant procéder à l'installation du moteur de Servlets. Pour cela vous devez d'abord télécharger la version 3.2.1 de Tomcat disponible à l'URL suivant : `http://jakarta.apache.org/builds/jakarta-tomcat/release/v3.2.1/bin/`. Cette version n'est pas la dernière version de Tomcat. En effet le développement de Tomcat est vraiment très actif, et de nouveaux concepts sont introduits régulièrement.

Par exemple, en ce qui concerne le protocole de communication entre le serveur Web Apache et le moteur de Servlets Tomcat, un nouveau procédé est apparu depuis la version 3.2.1 : JK. mod_jk (le module dynamique chargé par Apache au démarrage pour prendre en compte JK) est destiné, à moyen terme, au remplacement de mod_jserv (le protocole de communication entre le serveur Web et le moteur de Servlets que je vais décrire dans cette installation). JK apporte un certain nombre d'avantages comme le support de plusieurs serveurs Web différents via la même interface (IIS, Netscape Server, Apache 2.x, etc.), la gestion correcte du protocole HTTPS (protocole HTTP sécurisé grâce à l'utilisation de SSL). De plus, JK n'est pas une adaptation de d'Apache JServ au contraire de mod_jserv qui comporte donc du code inutile.

Cependant, le caractère nouveau de mod_jk ne me permet pas de le maîtriser suffisamment pour vous proposer une aide à propos des configurations complexes, contrairement à mod_jserv que je connais mieux. De plus, mod_jserv est encore largement plus répandu que mod_jk. Je vais donc vous expliquer la

configuration basique de Tomcat avec mod_jserv et mod_jk, et je vous proposerais ensuite une configuration avancée avec mod_jserv.

Afin d'installer Tomcat, je vous suggère de télécharger les sources de la dernière version stable à l'URL suivant : `http://jakarta.apache.org/builds/tomcat/release/v3.2.1/src/`. La dernière version stable est la 3.2.1, donc en ce moment vous devriez télécharger le fichier jakarta-tomcat-3.2.1-src.tar.gz.

Vous devrez ensuite télécharger d'autres outils :

- JSSE pour Java Secure Socket Extension permet d'établir des connexions sécurisées au travers d'un réseau (à l'aide de SSL par exemple) entre deux machines. Disponible à `http://java.sun.com/products/jsse/`.
- Ant : c'est un gestionnaire de compilation conditionnelle, au même titre que make sous Unix. Il permet, en écrivant des fichiers de description de compilation, de ne recompiler que les fichiers qui ont changé. C'est très utile pour compiler de gros programmes (comme Tomcat). Disponible à `http://jakarta.apache.org/builds/ant/release/v1.2/src/jakarta-ant-src.tar.gz`.
- JAXP pour Java API for XML Processing permet d'analyser des fichiers au format XML. Disponible à `http://java.sun.com/xml/download.html`.
- Servlet API : les classes qui implémentent l'API des Servlets. Disponible à `http://java.sun.com/products/servlet/download.html`.

Une fois ces outils téléchargés, vous devez créer un répertoire de base pour la création d'une version binaire de Tomcat : nous utiliserons /usr/local/tomcat-dist :

mkdir /usr/local/tomcat-dist

Décompressez ensuite tous les fichiers téléchargés dans ce répertoire :

tar xfvz jakarta-tomcat-3.2.1-src.tar.gz -C /usr/local/tomcat-dist

unzip jsse-1_0_2-gl.zip -d /usr/local/tomcat-dist

mkdir /usr/local/tomcat-dist/jakarta-ant

tar xfvz jakarta-ant-src.tar.gz -C /usr/local/tomcat-dist/jakarta-ant

unzip jaxp-1_0_1.zip -d /usr/local/tomcat-dist

unzip servlet-2_2b.zip -d /usr/local/jdk1.3/jre/lib/ext

Installons maintenant les packages Java nécessaires à la compilation de Tomcat. Nous avons déjà copié le package des Servlets dans /usr/local/jdk1.3/jre/lib, il faut faire de même avec les fichiers jaxp.jar et parser.jar de la version de JAXP que vous avez récupéré et avec les fichiers jnet.jar, jsse.jar et jcert.jar que vous avez récupéré dans l'archive de JSSE. Ce procédé vous évite d'ajouter le chemin vers ces fichiers dans la variable d'environnement CLASSPATH, mais vous pouvez également utiliser cette méthode.

Maintenant, il faut compiler Ant. Pour cela, allez dans le répertoire dans lequel vous l'avez décompressé et entrez :

./build.sh

la compilation devrait se dérouler sans aucun problème. Si une erreur de compilation mentionne qu'une classe ou qu'un package est introuvable, vous devriez

vérifier que les .jar sus-cités sont à un emplacement correct (ou que la variable d'environnement CLASSPATH contient de bonnes valeurs.).

Il faut maintenant compiler Tomcat. Pour cela positionnez vous dans le répertoire dans lequel vous l'avez décompressé, et lancez :

./build.sh dist

cette commande va construire une "distribution" binaire identique à celle qui est téléchargeable sur le site de la fondation Apache mais adaptée à votre configuration. Une fois la compilation effectuée, vous pouvez tester le fonctionnement de tomcat.

Pour cela, allez dans le répertoire "bin" de votre distribution de Tomcat qui devrait être $TOMCAT_HOME/build/tomcat/bin si $TOMCAT_HOME est /usr/local/tomcat-dist. Entrez en tant qu'utilisateur "root" :

./startup.sh

Vous devriez voir un série de messages apparaître mentionnant le démarrage de Tomcat. Ouvrez ensuite un navigateur Web, et demandez d'accéder à l'URL suivant : `http://127.0.0.1:8080/`. Si vous obtenez la page de garde du serveur Tomcat, tout fonctionne parfaitement.

A ce moment précis, Tomcat fonctionne en mode indépendant, ce qui n'est pas ce que nous souhaitons (voir 2.2.1 page 17). Nous allons maintenant décrire son association avec le serveur Web Apache de façon à en faire un moteur de Servlets externe.

Installation et configuration de Apache

Rassurez vous, cette partie de l'installation de l'environnement de travail est beaucoup plus simple que la précédente.

Tout d'abord, vous devez télécharger une version d'Apache supérieure à la 1.3.9, la dernière si possible (1.3.14) à l'URL suivant : `http://httpd.apache.org/dist/apache_1.3.14.tar.gz`.

Ensuite, vous devez la décompresser dans le répertoire de votre choix, par exemple /usr/local/src comme ceci :

tar xfvz apache_1.3.14.tar.gz -C /usr/local/src

ce qui devrait créer le répertoire /usr/local/src/apache-1.3.14. Allez dans ce répertoire avec la commande :

cd /usr/local/src/apache-1.3.14

et lancez la configuration du script de compilation comme ceci :

./configure –enable-module=so

ceci va activer le support des modules chargeables dynamiquement, ce qui sera nécessaire pour la communication entre Apache et Tomcat. Une fois le script de configuration terminé, lancez la commande suivante :

make install

qui va compiler et installer le serveur Web Apache dans le répertoire /usr/local/apache. Passons maintenant à la prise en charge de Tomcat. Comme je l'ai décrit plus haut, il existe deux façons différentes de procéder (voir 2.2.3 page 21).

Configuration de mod_jserv

Pour permettre à Apache d'utiliser mod_jserv afin de communiquer avec Tomcat vous devez compiler le module chargeable mod_jserv.so. Pour cela, repositionnez vous dans le répertoire contenant les sources de ce module :

cd /usr/local/tomcat-dist/jkarta-tomcat-3.2.1-src/src/native/apache/jserv

si vous avez effectué les mêmes manipulations que celles qui ont été décrites précédemment. Il faut ensuite utiliser l'utilitaire apxs fournit avec Apache et qui se trouve dans /usr/local/apache/bin (toujours si vous avez suivi mes instructions à la lettre) comme ceci :

/usr/local/apache/bin/apxs -o mod_jserv.so -c *.c

ce qui devrait créer un module chargeable mod_jserv.so. Copiez ce fichier dans le répertoire de stockage des modules chargeables de Apache comme ceci :

cp mod_jserv.so /usr/local/apache/libexec

et spécifiez à Apache que vous désirez utiliser ce module pour communiquer avec Tomcat. Pour cela, un fichier tomcat-apache.conf a été créé dans l'arborescence de la distribution de Tomcat que vous avez créé. Il suffit de l'inclure dans le fichier de configuration de Apache. Ceci peut être effectué en ajoutant la ligne suivante dans le fichier /usr/local/apache/conf/httpd.conf :

```
Include /usr/local/tomcat-dist/build/tomcat/conf/tomcat-apache.conf
```

et en validant la modification. Arrêtez ensuite tout processus concernant Apache ou Tomcat et lancez Tomcat puis Apache comme ceci :

/usr/local/tomcat-dist/build/tomcat/bin/startup.sh

/usr/local/apache/bin/apachectl start

ce qui devrait afficher quelques messages précisant que ces deux applications se sont correctement lancées. Vous pouvez maintenant tester votre installation en accédant à l'URL `http://127.0.0.1/test/`. Si vous obtenez une page Web vous souhaitant un joyeux Noël, la configuration est terminée.

La configuration de mod_jk

Mod_jk est lui aussi un module permettant la communication entre Apache et Tomcat, il fonctionne donc selon le même principe que mod_jserv dont nous venons d'aborder la configuration de base. Vous devez donc suivre la même démarche pour installer ce module. Les seules modifications à apporter concernent les fichiers à manipuler. je ne vais donc faire qu'énumérer les commandes nécessaires à son installation, car vous pouvez vous reporter aux explications données dans la section précédente pour d'éventuels détails.

Vous devrez exécuter dans l'ordre les commandes suivantes :

cd /usr/local/tomcat-dist/jkarta-tomcat-3.2.1-src/src/native/apache1.3

/usr/local/apache/bin/apxs -o mod_jserv.so -c *.c ../jk/*.c -I ../jk -I /usr/local/jdk1.3/include -I /usr/local/jdk1.3/include/linux

cp mod_jserv.so /usr/local/apache/libexec

et inclure le fichier /usr/local/tomcat-dist/build/tomcat/conf/mod_jk.conf dans le fichier de configuration d'Apache httpd.conf.

Redémarrez Tomcat et Apache comme précisé dans la section précédente et effectuez le même test.

La configuration des outils de communication entre Apache et Tomcat sont mis en place, mais je n'ai pas expliqué leur comportement en détail, ce qui peut être gênant si vous désirez personnaliser votre système. Je vous conseille donc de vous reporter à la documentation de Tomcat livrée avec les sources de celui-ci pour bénéficier d'avantages comme la répartition de charge, l'utilisation de machines virtuelles java distantes ou encore le réglage fin de la sécurité.

2.3 Ecrire sa première Servlet

L'environnement de travail est désormais correctement installé et configuré à votre goût : vous voilà fin prêt. Nous allons maintenant prendre connaissance avec les rudiments du développement des Servlets, ce qui nous mènera à l'écriture d'une première application : le désormais célèbre "Bonjour monde !". Nous verrons ensuite les différentes formes que peuvent prendre une Servlet et les outils de développement mis à votre disposition.

Comme nous l'avons dit précédemment, une Servlet n'est qu'un programme Java qui implémente une interface particulière : l'interface javax.servlet.Servlet. Une Servlet est donc un programme classique écrit en Java compilé avec le compilateur Javac. En général, une Servlet implémente l'interface javax.servlet.Servlet en étendant une des deux classes suivantes :

- javax.servlet.GenericServlet.
- javax.servlet.http.HttpServlet[6].

Le programme Java étendant une de ces deux classes peut également utiliser les classes de l'API standard ou d'autres extensions (Java Mail, JTA ou autres) comme n'importe quel programme Java. La nécessité d'implémenter l'interface javax.servlet.Servlet vient du fait que le moteur de Servlet qui gère les appels à votre Servlet doit pouvoir obtenir un point d'entrée dans votre programme (qui n'est pas la méthode main puisqu'une Servlet est instanciée une seule fois, à son premier lancement) pour chaque requête. Ce point d'entrée est une des méthodes définie dans l'interface javax.servlet.Servlet.

Voici un exemple de code pour vous donner une idée plus précise :

```
import javax.servlet.*;
import java.servlet.http.*;
public class BonjourMonde extends HttpServlet {
  public void doGet(HttpServletRequest req, HttpServletResponse res) throws
```

[6] C'est une sous-classe de javax.servlet.GenericServlet.

```
    ServletException, IOException) {
      res.setContentType("text/html");
      PrintWriter out = res.getWriter();
      out.println("<HTML>");
      out.println("<BODY>");
      out.println("<B>Bonjour monde !</B>");
      out.println("</BODY>");
      out.println("</HTML>");
    }
  }
```

Dans cet exemple, c'est la méthode **doGet()** qui sert de point d'entrée[7]. Elle est exécutée quand un client effectue une requête utilisant la méthode HTTP GET et construit une page HTML, qui est retournée au client.

Saisissez cet exemple dans un éditeur de texte quelconque, sauvegardez sous le nom "BonjourMonde.java" et compilez le à l'aide la commande suivante :

javac BonjourMonde.java

En principe, aucun message ne devrait apparaître. Si ce n'est pas le cas, vérifiez bien que vous avez bien suivi toutes les étapes du procédé d'installation de l'environnement de travail.

Votre première Servlet est maintenant compilée, vous devez la publier sur votre serveur. En ce qui concerne Tomcat, vous devez placer le fichier .class obtenu à la suite de la compilation du source à un endroit bien précis. Le choix de cet emplacement dépend du rôle de votre Servlet et de vos fichiers de configuration relatifs à Tomcat et Apache. Lorsque vous ne modifiez aucun de ces fichiers, vous pouvez par exemple copier les fichiers .class de votre Servlet dans $TOMCAT_HOME/webapps/test/WEB-INF/classes/ où $TOMCAT_HOME correspond au répertoire dans lequel Tomcat est installé. Dans ce cas précis, vous accéderez à votre Servlet via l'URL `http://127.0.0.1/test/servlet/BonjourMonde`. Un charmant message devrait apparaître sur votre navigateur : vous venez de programmer votre première Servlet.

Il existe d'autres manières de développer des Servlets. Ce sont les JSP (pour Java Server Pages) et SSI (pour Server Side Includes).

2.4 Les autres modes de programmation

Nous avons vu qu'une Servlet était un programme écrit en Java tout à fait classique. Afin de produire un contenu HTML, le programmeur emploie un flux d'écriture auquel il demande l'impression de balises HTML. Il en résulte, même pour une Servlet aussi simple que notre premier exemple, un nombre trop important d'instructions destinées à écrire sur la sortie ce fameux code HTML.

Ceci est un inconvénient car le contenu n'est pas séparé des traitements. Dans le cas où l'intégrateur de code HTML et le programmeur de Servlets sont deux

[7] Voir 3.2.2 page 42 pour obtenir plus d'informations à propos des points d'entrée.

(ou plusieurs) personnes différentes, cela peut devenir rapidement très gênant : le programmeur doit constamment s'assurer que le code HTML qu'il produit lors des traitements est conforme à ce que désire l'intégrateur (ou le webmaster). Deux alternatives à la programmation classiques de Servlets sont donc disponibles : SSI (pour Server Side Includes) et JSP (pour Java Server Pages).

2.4.1 SSI

C'est une méthode qui existe déjà pour d'autres techniques de programmation d'applications qui tournent sur un serveur Web. En effet, il est possible d'utiliser SSI en programmant des scripts CGI par exemple.

Afin de pouvoir utiliser cette technique, le serveur Web doit proposer un support pour les SSI. Ce support varie d'un serveur à l'autre. Une Servlet utilisée avec SSI se compose de deux parties : le code de la Servlet elle-même et le code HTML de la page Web qui inclue le résultat de cette Servlet. Voici un exemple de code HTML pour une page désirant utiliser la Servlet Bonjour :

```
<HTML>
<BODY>
<TITLE>Essai d'utilisation de SSI</TITLE>
<P>
Voici un exemple d'utilisation de la sortie produite par une Servlet.
</P>
<SERVLET CODE=Bonjour CODEBASE=http ://localhost :8080/>
<PARAM NAME=''nom'' VALUE=''Julien''>
Si vous lisez ce texte, c'est que votre serveur Web ne supporte pas
les Servlets utilisées via SSI.
</SERVLET>
</BODY>
</HTML>
```

Afin de signaler au serveur Web que cette page HTML comprend une directive SSI, il faut nommer le fichier avec l'extension ".shtml", et non ".html". Ceci est le comportement par défaut et dépend en réalité de la configuration de votre serveur Web.

Tout d'abord on remarque une balise inhabituelle : la balise <SERVLET>. C'est elle qui indique que l'on veut inclure le contenu produit par une Servlet dans la page Web. Cette balise reconnaît plusieurs paramètres. Le paramètre CODE indique le nom de la Servlet à invoquer. Le paramètre CODEBASE doit, juxtaposé à la valeur du paramètre CODE, former l'URL grâce auquel on peut accéder à la Servlet en temps normal (sans utiliser SSI). Ensuite, il est possible de passer des paramètres à cette Servlet, grâce à la balise PARAM. Le nom de ce paramètre est précisé par l'attribut NOM et la valeur par VALUE. Ces paramètres seront récupérés par la méthode **getParameter(String nomParametre)** de l'objet de type **HttpServletRequest** passé à la Servlet lors de son invocation. Le texte écrit juste avant la fermeture de la balise SERVLET

s'affiche au cas où le serveur Web ne supporte pas cette balise (et donc les Servlets invoquées via SSI). Il est également possible de déterminer les valeurs de certains paramètres dès l'initialisation grâce à un attribut placé dans la balise <**SERVLET**> de la forme **initParamètre1**=**valeur1**.

La servlet Bonjour, appelée lors de l'analyse de la page HTML ci-dessus, pourrait être la suivante :

import javax.servlet.*;

```
import java.servlet.http.*;
public class BonjourMonde extends HttpServlet {
  public void doGet(HttpServletRequest req, HttpServletResponse res) throws
  ServletException, IOException) {
        PrintWriter out = res.getWriter();
        String param1 = req.getParameter("nom");
        out.println("<B>Bonjour " + param1 + "!</B>");
  }
}
```

Il suffit de compiler cette Servlet de manière classique et de la placer à l'emplacement indiqué par la balise "CODEBASE" du code HTML de la page incluant la sortie de la Servlet. On note la compacité du code par rapport à une Servlet autonome produisant le même effet. Les Servlets invoquées via SSI sont intéressantes lorsque le contenu statique est plus important que le contenu dynamique. Cela permet à l'intégrateur du code HTML et au développeur de Servlets une plus grande liberté : ils n'ont rien à modifier pour que les modifications soient prisent en compte des deux côtés.

On remarque aussi qu'il ne faut pas définir le type MIME de la réponse, car la réponse du serveur Web a déjà débuté au moment de l'appel à la Servlet, et ce début de réponse contenait l'entête précisant son type MIME (voir 2.1.2 page 15). La ligne :

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

n'est donc plus nécessaire.

2.4.2 JSP

Une autre technique d'inclusion de code dans des pages HTML est JSP. Cette technique est née d'un besoin émanant des développeurs : ils avaient besoin d'un mode de développement similaire aux ASP de Microsoft (voir 1.5 page 11) et à PHP (voir la même section que pour ASP). De cette manière ils pourraient inclure directement du code (et non pas un marqueur appelant une Servlet) dans la structure de leurs documents écrits en HTML.

Les JSP permettent donc d'éviter au programmeur d'employer l'instruction **out.println(String chaine_a_imprimer)** trop souvent lorsque le contenu est en majorité statique. Cependant, elles ne garantissent pas une très bonne

coopération entre l'intégrateur HTML et le développeur de Servlets, dans le cas où ils ne sont pas la même personne. En effet si le programmeur désire changer son code, il doit ouvrir le fichier contenant le code HTML et si l'intégrateur HTML désire changer la structure ou le contenu de la page Web, il doit ouvrir le fichier contenant le code de la Servlet. C'est donc à priori moins souple mais il s'avère que cette technique est très bien adaptée pour les développements de petite envergure, et lorsque le développeur et l'intégrateur HTML ne sont qu'une seule personne.

2.4.3 La forme

Voici comment se présente une JSP qui produit le même message que les Servlets précédentes (une salutation amicale très célèbre) :

```
<HTML>
<HEAD>
<TITLE>Essai de JSP</TITLE>
</HEAD>
<BODY>
<% if (request.getParameter("nom") == null) {
      out.println("Bonjour monde !") ;
   } else {
      out.println("Bonjour " + request.getParameter("nom")  + " !") ;
   }
%>
</BODY>
</HTML>
```

Il suffit d'enregistrer ce fichier avec un nom ayant pour extension ".jsp" et de le placer dans le répertoire adéquat, ce qui peut dépendre de la configuration de votre serveur Web ou du moteur de Servlets que vous utilisez.

Je vais vous donner maintenant quelques explications sur les écritures pouvant vous sembler mystérieuses. Premièrement, la balise "<%" permet d'indiquer le début d'une section de code dynamique. La balise "%>" permet donc de marquer la terminaison d'une telle section (on appelle ce type de section une *scriptlet)*. Je garde une appellation aussi générale car il est possible d'écrire des JSP avec un autre langage que Java (comme javascript qui est complètement différent) même si ce n'est pas très courant. A l'intérieur de ce code, on peut placer plusieurs types d'instructions :

- les directives.
- les expressions.
- le code classique.

Les directives permettent de définir certains aspects structurels de la servlet de fond[8], comme l'importation d'un package, l'extension d'une classe, le type

[8] Pour l'instant, nous considérons que la servlet de fond est une servlet classique effectuant la même chose que la JSP correspondante. C'est à dire une traduction de la JSP en servlet. Le concept de servlet de fond est expliqué plus précisemment à la section suivante, patience ;-).

de contenu produit ou encore la méthode HTTP utilisée (voir 2.1.2 page 15 pour connaître ce qu'est une méthode HTTP). Les définitions sont effectuées par l'intermédiaire d'affectation de valeurs à des variables bien définies. Ces directives sont encadrées d'une balise ouvrante "<%@" et d'une balise fermante "%>". Par exemple, pour importer le package "monpackage" dans la JSP, il suffit d'insérer le code suivant :

```
<%@ import = "monpackage.*"%>
```

Pour définir une extension de classe, on utilise :

```
<%@ extends = "maClassePersoHttp" %>
```

si l'on veut que la servlet de fond étende la classe maClassePersoHttp. Je ne vais pas énumérer toutes les directives disponibles ici, pour cela je vous invite à vous reporter à l'annexe B et à la documentation de référence à propos de jsp disponible à `http://java.sun.com/products/jsp/`.

Les expressions permettent de convertir le résultat de l'expression évaluée en String et d'inclure ce résultat converti dans le code HTML de la page Web générée par la Servlet de fond. Cela permet d'éliminer les appels à la méthode **println(String chaine)**. Une expression débute par la balise ouvrante "<%=" et se termine par la balise fermante "%>". Par exemple, l'expression suivante :

```
<%= ma_variable %>
```

inclut la valeur de la variable **ma_variable** dans le code HTML généré. On peut bien entendu utiliser des méthodes comme expression à évaluer comme dans :

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

qui permet d'inclure la valeur du paramètre de nom **parametre** passé à la Servlet de fond. Remarquez que l'absence de " ;" à la fin de l'expression n'est pas un oubli.

Le code classique utilise les mêmes conventions que les Servlets traditionelles, mais vous pouvez remarquer l'emploi d'identifiants qui n'ont pas été définies. Ces identifiants sont :

– request : l'objet représentant la requête venant du client.
– out : l'objet représentant le flux d'impression en sortie.

Ils sont utilisables dans chaque page Web utilisant JSP.

Maintenant que nous avons décrit la forme du développement de Servlets grâce à JSP, intéressons-nous au fonctionement interne.

2.4.4 En coulisses

Je mentionne dans la section précédente le terme de "Servlet de fond". En effet, le moteur de Servlets ne procède pas un interpréteur spécifique aux Servlets et

un autre spécifique aux documents utilisant JSP. Le procédé d'exécution d'une JSP se déroule en plusieurs étapes.

Tout d'abord, le fichier correspondant à la JSP, qui comprend à la fois du code HTML, du code Java ainsi que les directives et déclarations, est traduit afin d'obtenir le code d'une Servlet aboutissant au même résultat. Cette traduction est quasiment directe pour le code dynamique tandis que le code statique est traduit grâce à des appels à la méthode **println(String chaine)**.

Une fois la traduction effectuée, la Servlet de fond obtenue (qui est alors une servlet tout à fait classique jusqu'à la fin de son cycle de vie) est déplacée dans un emplacement particulier du système de fichier du moteur de Servlet pour y être compilée, comme une Servlet traditionnelle.

Une fois compilée, la Servlet de fond est chargée, initialisée et exécutée.

De ce procédé d'exécution des JSP découlent plusieurs de leurs comportements propres. En effet, lors de la première exécution d'une JSP, vous constaterez qu'il est nécessaire de patienter un certain temps avant d'obtenir le résultat du traitement, alors que lors des appels ultérieurs, ce n'est pas le cas. Cela est dû au fait que lors du premier appel, il faut réaliser tout le procédé de la traduction à l'exécution (en passant par la compilation) alors que lors des appels suivants, seule l'exécution est nécessaire. Ensuite, il est maintenant clair et légitime de ne pas disposer d'une API spécifique aux JSP, car ce sont en définitive des Servlets classiques avant leur compilation. Seul le support proposé par le moteur de Servlets influe sur les possibilités dont vous pouvez bénéficier (comme de nouvelles directives permettant d'inclure des Java Beans par exemple). Enfin, vous pourrez comprendre que votre JSP se recharge automatiquement dès que le fichier contenant son code est modifié (le fichier possédant l'extension ".jsp") car une Servlet est rechargée par le moteur de Servlets pour les mêmes raisons.

Il me semble nécessaire de préciser que malgré les services que peuvent rendre les JSPs pour de petites applications, elles ont surtout été crées pour opposer une réponse aux ASPs de Microsoft. Ces dernières ressemblent en effet énormément aux JSPs à tous les niveaux, et sont appréciées des programmeurs Web côté serveur pour leur facilité d'utilisation.

Vous connaissez maintenant les principales formes de développement des Servlets, je vais maintenant vous proposer quelques références à propos d'outils de développement pouvant faciliter la programmation en tirant parti des avantages de la technologie de la plate-forme Java côté serveur.

2.5 Les outils pour développer

Comme nous l'avons vu précédemment, seul un éditeur de texte classique et un compilateur java sont nécessaires pour programmer une Servlet. Cependant ce type de programme peut bénéficier de certains outils prenant en compte les spécificités du processus de développement des servlets.

Tout d'abord, une Servlet est un programme déployé sur une machine distante (en général). Or, comme tout programme créé par un humain, une Servlet possède des bogues ou des erreurs de conception. Ces erreurs doivent pouvoir être détectées et corrigées depuis la machine du développeur sur la machine de déploiement (donc à distance). De plus, ces corrections ne doivent pas porter préjudice au fonctionnement des autres Servlets ou du serveur de Servlets (ou du serveur Web selon que vous optiez pour un moteur de Servlets embarqué ou externe) lui même car nous avons pu voir (voir 1.3.3 page 9) que les Servlets sont souvent développées dans le cadre d'applications critiques pour l'entreprise (et le temps passé au débogage représente autant d'argent perdu). Un outil prenant en compte ces exigences constitue donc un avantage indéniable.

Ensuite, outre la programmation de Servlets écrites en Java avec une syntaxe classique, nous avons vu qu'il était possible d'utiliser les JSP. Ce procédé de développement utilise une syntaxe différente qui doit être clairement présentée pour être comprise et faciliter le développement. La prise en compte de la syntaxe utilisée pour le développement des JSP est donc une caractéristique intéressante.

Après, nous avons vu que la diffusion des Servlets sur des serveurs d'application ou leur utilisation dans le cadre d'applications "métiers" amène le développeur de Servlets à utiliser de nombreux composants propres à la version entreprise du kit de développement Java. Ces composants, comme les EJB (Enterprise Java Beans) voient leur dévelopement considérablement facilité par l'utilisation d'outils spécifiques réduisant l'effort nécessaire de développement concernant le "protocole" de développement propre aux Beans (la génération du squelette pour le nommage des accesseurs en lecture et en écriture par exemple) ou le développement "visuel" des Beans. Ces composants étant par définition amenés à être utilisés par d'autres composants pour former un système, leur déploiement à distance doit aussi être facilité.

Enfin, pour pouvoir tester localement le développement des Servlets, il est nécessaire de disposer d'un moteur de Servlet et de la version entreprise du JDK. Ces deux composants, lorsqu'ils sont intégrés à l'outil de développement, permettent une facilité et une efficacité de programmation accrue.

Toutes ces caractéristiques et d'autres encore sont reprises par plusieurs outils que je vais présenter succintement ici, il ne tient qu'à vous de devenir un utilisateur averti de ces logiciels.

Le premier de ces outils est JBuilder de Borland. La version 4.0 a apporté de sensibles améliorations et cet IDE (pour Integrated Development Environmnent) propose quelques caractéristiques intéressantes comme l'intégration de la version 1.3 du JDK, le débogage à distance, le débogage et mise en valeur du code source pour JSP, le support de la version 2.2 de l'API Servlet et de la version 1.1 des JSP, un moteur de Servlets intégré (WebLogic de BEA, voir 1.4 page 9), la création visuelle de Java Beans et le déploiement de ceux-ci sans redémarrer le moteur de Servlets. Toutes ces caractéristiques sont disponibles avec la version "Entreprise" du logiciel, qui est la plus aboutie mais aussi la plus chère . Certaines sont disponibles avec des versions moins évoluées, pour en savoir plus reportez vous à l'URL `http://www.borland.com/jbuilder/jb4/feamatrix/`. Il est à noter que la version personnelle disponible gratuitement ne propose aucun de ces avantages.

Nous trouvons ensuite JRun Studio de Allaire, qui en est actuellement à sa version 3.1. C'est un outil de développement destiné à être couplé au serveur d'application JRun du même éditeur. Ce produit ne fonctionne que sous Windows et dispose de fonctionalités similaires à JBuilder mais il contient de nombreuses fonctionnalités destinées à la conception d'applications Web, à la manière d'un outil d'intégration HTML. Certains peuvent le percevoir comme un avantage (les intégrateurs de code HTML), d'autres comme un inconvénient (les programmeurs) car cela peut ajouter un peu de confusion.

Chapitre 3

Fonctionnement interne

Vous devriez maintenant disposer des connaissances suffisantes pour pouvoir débuter le développement de servlets et les faire fonctionner grâce aux outils de travail que vous avez pu mettre en place précédemment. Je vais maintenant tenter de vous décrire de manière complète le fonctionnement interne du moteur de servlets qui permettra de faire fonctionner vos futures servlets de manière efficace et cohérente. Grâce aux connaissances que vous allez acquérir dans ce chapitre, j'espère que vous comprendrez réellement le fonctionnement des servlets. Cela devrait vous permettre d'être relativement indépendant par rapport à la suite de votre apprentissage car vous pourrez comprendre n'importe quel document au sujet des servlets et vous pourrez aussi déduire des connaissances par vous même au prix d'un effort de réflexion logique. Il est important de noter que, même si les notions évoquées dans ce chapitre paraissent trop techniques et relever du détail, elles vous seront vraiment utiles dans votre développement de tous les jours. En effet, les mécanismes de fonctionnement de la plate-forme Java influeront directement sur le code que vous écrirez. Comme le dit si bien le proverbe, je vais tenter de vous apprendre à pêcher plutôt que de vous offrir le poisson à chaque fois que le besoin s'en fera sentir.

Les moteurs de Servlets ont besoin d'une machine virtuelle Java pour fonctionner. Cela signifie qu'en rentrant dans les détails de fonctionnement des Servlets, j'évoquerai à de nombreuses reprises des notions ayant trait au fonctionnement de toute machine virtuelle Java et à des notions encore plus universelles comme les threads, les processus, le chargement des classes au démarrage d'une application, le ramasse-miettes ou encore les références. Je considère tout ceci comme acquis et je ne donnerai pas d'explications sur ces notions à part si elles possèdent quelques spécificités liées à leur utilisation par les Servlets. Il est donc temps, si vous ne maîtrisez pas toutes ces notions, de vous documenter. Une bonne source pour se documenter sur de tels sujets est le bouquin "The Java Virtual Machine" disponible à l'URL `http://java.sun.com/docs/insidejava2vm/` qui vous apportera plus de connaissances qu'il n'en faut.

Il existe de nombreux fournisseurs de machines virtuelles Java différents. Chaque fournisseur se conforme à la norme définie par Sun et les bibliothèques de classes ainsi que les machines virtuelles fournies par ces organismes possèdent les mêmes "interfaces" mais pas forcément la même implémentation de ces interfaces. Un

chargeur de classes peut très bien utiliser une méthode différente pour la JVM de Microsoft pque pour la JVM de Sun. Ce n'est pas une chose rare. Les moteurs de Servlets eux aussi proviennent de nombreux éditeurs différents et sont donc programmés différemment par des développeurs différents. Ils se conforment aux aussi à une norme définie par Sun qui est représentée par les spécifications de l'API des Servlets qui ne décrit que les comportements que doivent suivre les moteurs de Servlets de façon générale. L'implémentation est libre et toujours différente entre deux moteurs de Servlets différents. Or nous avons vu que les moteurs de Servlets ne peuvent pas fonctionner sans une machine virtuelle Java, ce qui m'amène à cet avertissement : les comportements que je vais décrire dans le présent chapitre sont des comportements recommandés et rencontrés le plus souvent possible mais peuvent être différents selon la solution adoptée. Je vous conseille donc de garder un oeil critique sur ces descriptions, surtout celles qui évoquent les détails d'implémentation. Les comportements généraux sont quant à eux, en général, identiques pour chaque solution.

Je tenterai d'effectuer une comparaison avec les autres solutions une fois toutes les connaissances acquises à propos du fonctionnement interne.

3.1 Le modèle de fonctionnement

Nous savons que pour faire fonctionner des Servlets, il est nécessaire de disposer d'une machine virtuelle Java, d'un moteur de Servlets implémentant l'API des Servlets et éventuellement d'un serveur Web. Ces différents composants prenant part au système d'exécution des Servlets coopèrent d'une façon bien déterminée. J'ai rapidement présenté ce fonctionnement lors de la description des différentes solutions (voir 2.2.1 page 17) pour la mise en place d'une plate-forme exécutant des Servlets, mais il est nécessaire d'apporter des précisions.

La machine virtuelle Java peut être reliée de plusieurs façons au serveur Web qui gère les requêtes venant de l'utilisateur, ceci dépendant du type de solution choisi :

La machine virtuelle fait partie intégrante serveur, le serveur étant écrit en Java : c'est le cas du moteur de Servlets indépendant (comme Java Web Server ou Tomcat dans certains cas). Le moteur de Servlets est alors un thread de la JVM au même titre que les Servlets qu'il exécute, et que le serveur Web qui accepte les requêtes HTTP.

La machine virtuelle Java constitue un des Threads du serveur Web : c'est le cas du moteur de Servlets embarqué dans un serveur Web mono-processus et multi-threads. Dans ce cas précis l'intégration des Servlets dans le serveur Web est importante et il est facile, par exemple, de récupérer des informations concernant la requête provenant du client via les méthodes fournies par l'API des Servlets.

La machine virtuelle Java est un processus externe au serveur Web, qui est lui multi-processus. C'est le cas de Tomcat lorsqu'on l'utilise comme moteur de Servlets externe couplé au serveur Web Apache. Les processus lancés par le serveur Web et qui représente autant de clients effectuant des requêtes HTTP peuvent alors ce partager le processus correspondant à la JVM externe.

Suivant le choix de la solution, c'est aussi le coût en ressource processeur et mémoire qui va changer : plus le moteur de Servlets est externalisé, plus les changements de contexte sont lourds. En effet, les informations spécifiques à celle-ci pourront être partagées directement dans le cas des moteurs de Servlets indépendants (changement de contexte léger) ou devront être préalablement copiées en mémoire et nécessiter donc un traitement (changement de contexte lourd) dans le cas des moteurs de Servlets externes.

Cependant, quelle que soit la solution adoptée, chaque Servlet s'exécute dans un thread de la machine virtuelle Java à laquelle elle est associée. Cela implique que les Servlets peuvent communiquer entre elles facilement (changement de contexte léger) car elles fonctionnent dans le même espace d'adressage et les données sont donc accessibles en lecture et en écriture simplement. Le fait qu'elles constituent un thread exécuté dans la même machine virtuelle que le moteur de Servlets qui les exécute présente aussi quelques particularités gênantes. En effet, le moteur de Servlets et les Servlets partagent le même processus et une Servlet peut très bien entreprendre une action affectant le processus complet (et donc le moteur de Servlets). Un des exemples critiques est l'emploi de l'instruction **System.exit()** dans le code d'une de vos Servlets qui cause l'arrêt du moteur de Servlets jusqu'à ce que l'administrateur le redémarre. Bien entendu, il existe des moyens tout à fait classiques d'éviter ce type de problème (voir le chapitre traitant de la sécurité et plus particulièrement le Security Manager). Ce type de comportement illustre très bien un des aspects du fonctionnement des moteurs de Servlets.

La nécessité de disposer du traitement de plusieurs requêtes de manière simultanée provient de l'utilisation même des applications Web : plusieurs personnes peuvent demander le même service en même temps. Que se passe-t-il si plusieurs personnes demandent à accéder à la même Servlet en même temps ?

Il existe deux solutions possibles (comme il est précisé dans la normalisation des Servlets) :

- Une seule instance de chaque servlet est chargée en mémoire et chaque requête venant des clients Web sont autant de threads qui peuvent manipuler cette instance.
- A chaque requête HTTP correspond une instance différente pour chacune des servlets. Plus il y de requêtes demandant l'accés à une servlet, plus il y a d'instances de cette servlet chargées en mémoire.

La première solution constitue le comportement par défaut, ainsi aucune particularité dans la syntaxe de votre code ne doit être ajoutée. Lorsqu'un client accède à une Servlet, la méthode principale (**doGet()**, **doPost()** ou **service()** selon le choix de la méthode HTTP¨et de l'interface implémentée) est appelée et un nouveau thread est créé. C'est dans ce nouveau thread que le traitement correspondant au code de votre Servlet va être exécuté. Cependant, c'est toujours la même instance de la Servlet qui prend en charge la requête : les attributs de classes d'une même Servlet modifiés par un client sont donc modifiés pour tous. Cela implique l'utilisation des sémaphores via le mot clef **synchronized** afin de faire face aux éventuels problèmes de synchronisation. Il faut donc prendre garde à bien synchroniser la manipulation des attributs d'instance, mais aussi à ne pas tout encadrer dans un bloc synchronisé : le caractère simultané des requêtes serait alors perdu. Par exemple, vous ne devez pas marquer la méthode principale

(**doGet()**, **doPost()** ou **service()** en général) avec le mot clef synchronized, l'instruction accédant en lecture ou en écriture à l'attribut en question doit elle seule être encadrée. Voici un exemple de code d'une mauvaise synchronisation, puis d'une bonne synchronisation :

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
public class MauvaisThread extends HttpServlet {
    private int compteur = 0;
    public synchronized void doGet(HttpServletRequest req, HttpServletResponse
res) throws ServletException, IOException {
        PrintWriter out = res.getWriter();
        compteur++;
        out.println("Cette Servlet a été accédée " + compteur + " fois.");
    }
}
```

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
public class BonThread extends HttpServlet {
    private int compteur = 0;
    public void doGet(HttpServletRequest req, HttpServletResponse res)
                      throws ServletException, IOException {
        synchronized(this) {
        int tmpCompteur = ++compteur;
        }
        PrintWriter out = res.getWriter();

        out.println("Cette Servlet a été accédée " + tmpCompteur +
        fois.");
    }
}
```

La nécessité d'apporter une attention particulière à la synchronisation est une des principales charge de travail qu'impose l'attribution d'un thread séparé pour chaque requête vers la même Servlet. Par contre, l'amélioration au niveau de l'occupation mémoire et de la vitesse d'exécution est perceptible : un seul ensemble de classes est chargé en mémoire (les classes utilisées par les Servlets qui ont été demandées au moins une fois) et lors d'une nouvelle requête, tout l'environnement de la Servlet n'est pas dupliqué, c'est un thread qui est créé : cela prend moins de temps. C'est un modèle vraiment différent des applications utilisant la norme CGI car, dans ce cas là, à chaque requête correspond un processus, ce qui pose d'évidents problèmes d'occupation en mémoire et de vitesse d'exécution (surtout pour le lancement du script). Les autres modes de développement comme PHP (lorsqu'il est utilisé en temps que module d'un serveur

Web) ou ISAPI par exemple sont assez similaires au niveau du modèle d'exécution (une seule instance du programme, chaque requête est un thread différent). Bien entendu, la synchronisation n'est pas nécessaire si vous manipulez des variables locales seulement, qui sont propres à chaque thread, et donc à chaque requête. Ce modèle d'exécution permet aussi la persistance des Servlets : une Servlet peut très bien conserver en mémoire des informations acquises au fur et à mesure des nombreuses requêtes, cela introduit de nombreuses facilités de programmation pour traiter des cas faisant appel à des renseignements provenant des requêtes antérieures, comme le compteur précédent.

Une autre conséquence qui montre bien le mode de fonctionnement d'une Servlet utilisant un mode d'exécution classique est la persistance des threads nouvellement créés à l'intérieur du code d'une Servlet. En effet, lorsqu'un thread est créé et lancé lors du traitement d'une requête, celui-ci ne se termine pas lorsque la requête est terminée (à moins qu'on précise le contraire) mais lorsque l'instance de la Servlet est détruite (c'est à dire rarement dans des conditions normales d'utilisation : lors de l'arrêt du moteur de Servlets). Cela permet d'effectuer des traitements de type batch très intéressant tout en manipulant les données de l'instance de la Servlet. On peut imaginer une Servlet lançant toutes les nuits la production d'un rapport en arrière plan

L'autre modèle d'exécution est le modèle mono-thread. Ici, à chaque requête correspond une instance de Servlet. La totalité des instances créées pour une Servlet se nomme le *pool de Servlets*. Lors d'une nouvelle requête, le thread correspondant à cette requête puise dans ce pool de Servlets et communique avec l'une d'entre elles. Il est impossible, lors de l'utilisation de ce modèle d'exécution, que deux requêtes concurrentes soient liées à la même instance de Servlet. C'est important car c'est cette impossibilité qui rend ce modèle d'exécution "Thread Safe", c'est à dire sûr par rapport aux threads, sans modification dans le code, et sans devoir prêter attention à quoi que ce soit de particulier (comme la synchronisation).

La ou les Servlets désirant bénéficier de cette particularité doivent implémenter l'interface **SingleThreadModel**. Cela permet de bénéficier de ce comportement pour certaines Servlets de votre choix, et de conserver un comportement multi-threadé pour les autres. C'est la seule contrainte au niveau de l'écriture du code source d'une Servlet utilisant un tel modèle d'exécution.

Bien entendu, les possibilités et avantages évoqués lors de la description du modèle d'exécution traditionnel ont disparus, mais d'autres sont apparus. D'une part vous êtes vraiment sûr que plusieurs requêtes effectuées en même temps ne produiront pas de résultats erronés (ce que vous ne pouvez garantir à moins d'être un expert en programmation multi-thread). Ensuite cela vous permet d'exploiter plus efficacement le caractère multi-thread des applications utilisées par vos Servlets. C'est un peu paradoxal mais prenons l'exemple d'une Servlet liée à un SGBD : dans le cas d'une seule instance de Servlet, il n'est pas efficace de gérer les multiples connexions avec des comptes utilisateurs différents (correspondant aux utilisateurs effectuant la requête côté client) tout en synchronisant les connexions et les requêtes. C'est un peu le même problème que de réinventer la roue : le travail effectué en temps normal par le SGBD pour synchroniser les connexions multiples serait alors effectué par la Servlet. De plus, le traitement ne serait pas le plus efficace dans le cas du modèle d'exécution

traditionnel, alors qu'il le serait dans le cas d'un modèle mono-thread, tout en débarassant le programmeur de taches fastidieuses de synchronisation. En effet chaque instance de la Servlet utilise son environnement et les requêtes envoyées à la base n'ont pas besoin d'être synchronisées.

Nous avons vu que, dans tous les cas, chaque Servlet était matérialisée par un thread lancé par une JVM, et qu'il existe deux modèles d'exécution : le modèle multi-thread et le modèle mono-thread. Le premier correspond très bien à une gestion centralisée des ressources (comme pour un serveur de discussion en temps réel par exemple). En effet, la servlet agit alors comme un répartiteur de messages entre les différentes personnes connectées, les informations doivent donc être centralisées. Le deuxième s'accorde très bien avec l'utilisation de systèmes gérant déjà le multithreading et nécessitant des traitements ponctuels efficaces qui ne se basent pas sur les requêtes précédentes. Ainsi l'absence de gestion de synchronisation permet de servir chaque requête au plus vite, alors que la concurrence des requêtes gérées par le SGBD est assurée par ce dernier.

La description du fonctionnement interne de la gestion des Servlets au sein d'un moteur de Servlets étant terminée, je vais maintenant décrire les différentes étapes du cycle de vie d'une Servlet.

3.2 Le cycle de vie

De la compilation du code à l'arrêt du moteur de Servlets en passant par le traitement des requêtes provenant du client, les Servlets entament un chemin bien précis qui peut se décomposer (de manière très générale) en trois étapes :

- le chargement
- le traitement des requêtes, que l'on peut appeler exécution
- la destruction

3.2.1 Le chargement

Afin qu'une Servlet puisse accueillir les requêtes venant des clients, nous avons vu dans la section précédente qu'une instance de cette Servlet doit être présente. Le processus de chargement de la Servlet est effectué par le moteur de Servlet. Ce chargement peut être effectué au démarrage du moteur de Servlets ou bien juste au moment où le moteur détermine qu'il a besoin de la Servlet en question.

Tout d'abord, le moteur de Servlets recherche une classe du type de la Servlet à charger à l'endroit où se trouvent les classes des Servlets pour chaque contexte (voir 3.2.4 page 45). Si la Servlet n'est pas dejà chargée, le moteur de Servlets charge la Servlet en utilisant un chargeur de classe normal à partir du système de fichier local, ou de toutes autre ressource distante (cela dépend des contextes, pour plus de renseignements à propos des contextes, voir 3.2.4 page 45). Une fois que la Servlet est chargée, elle est instanciée et un objet du type de la Servlet chargée est donc présent en mémoire. Une Servlet peut être chargée une fois ou plus par le moteur de Servlets, cela dépend du modèle d'exécution choisi : si la Servlet implémente l'interface SingleThreadModel plusieurs instances peuvent

être créées en mesure du nombre de requêtes faites par l'utilisateur, si rien n'est précisé par le programmeur (la classe de la servlet n'implémente pas **SingleThreadModel**) une seule instance de la Servlet peut être présente. Plusieurs instances d'une même Servlet peuvent être créées également si plusieurs Servlets identiques possèdent des paramètres d'initialisation différents.

L'initialisation est la dernière étape du chargement d'une Servlet. Une Servlet doit être initialisée avant de pouvoir répondre aux requêtes provenant du client. En effet, il est possible que certaines variables d'instance (ou toutes) de la classe correspondant à la Servlet à charger requièrent des valeurs de départ, dépendant de paramètres connus seulement au moment où elle sont chargées ou enregistrées lors de leur dernière destruction.Les valeurs de ces paramètres d'initialisation peuvent être précisées par un outil de configuration pour certains moteurs de servlets (comme Java Web Server), par une balise spéciale lorsque l'on utilise SSI (voir 2.4.1 page 27) ou par programme. Le moteur de Servlet effectue cette initialisation en appelant la méthode **init(ServletConfig config)** de l'interface **Servlet**. un paramètre de type **ServletConfig** est passé lors de l'appel de cette méthode. Ainsi, il est possible d'accéder dans le code source de la Servlet aux paramètres d'initialisation employé pour cette dernière. L'objet de type **ServletConfig** passé en paramètre à cette méthode permet également d'accéder à un objet de type **ServletContext** (car il implémente l'interface **ServletContext**) qui décrit l'environnement d'exécution dans lequel tourne la Servlet. Il faut noter que si vous désirez personnaliser l'initialisation de votre Servlet en redéfinissant la méthode **init** de celle-ci, vous devez impérativement utiliser l'instruction

```
super.init(config);
```

en admettant que **config** soit l'identifiant de l'objet de type **ServletConfig** passé en paramètre à la méthode **init** que vous redéfinissez.

Je vous ai décrit le fonctionnement du chargement d'une Servlet dans le cas où tout se déroule comme prévu. Mais qu'en est-il lorsqu'une erreur survient, ou que le chargement de la Servlet n'est pas désiré? La méthode **init** peut générer une exception de type **ServletException** ou **UnavailableException**. Une fois l'exception générée, le processus de chargement est abandonné et l'instance de la Servlet est immédiatement détruite. La méthode **destroy** que nous allons découvrir d'ici peu n'est pas appelée dans ce cas. Une fonctionnalité intéressante est fournie par le constructeur de l'exception de type **UnavailableException**. En effet, il est possible, via un paramètre passé au constructeur de l'exception, d'imposer au moteur de Servlets de respecter un délai avant une nouvelle tentative de chargement de la Servlet dont le chargement vient d'échouer. L'instruction suivante :

```
throw new UnavailableException("Charge du serveur trop élevée", 120);
```

permet de respecter un délai de deux minutes avant la prochaine tentative de chargement à cause d'un taux d'occupation de la machine serveur trop élevé.

Le rechargement d'une servlet

Il existe une fonctionnalité mise à disposition des développeurs de servlets lorsqu'ils se trouvent dans une phase de développement soutenue. En effet, plutôt que d'effectuer des manipulations impliquant une interruption de service (le redémarrage du serveur) pour recharger une servlet afin de prendre en compte les modifications apportées au code, c'est le moteur de servlets qui recharge la servlet de façon automatique s'il pense que cela est nécessaire. Ainsi, il suffit de recompiler le code source d'une servlet pour qu'une nouvelle instance de celle-ci remplace l'ancienne instance. On peut également utiliser la commande **touch** sous Unix en lui passant comme paramètre le fichier de la classe à recharger pour effectuer la même chose (sans le coût d'une recompilation).

Pour déterminer s'il est nécessaire de recharger une classe, le moteur de servlets détermine si le fichier de classe a changé (en déterminant si la date de dernière modification du fichier de classe est bien égale à la date de modification du fichier de classe au moment de son dernier chargement par exemple). S'il a changé, le moteur de servlets arrête le chargeur de classe utilisé pour charger la classe précédente et effectue le chargement de la nouvelle classe avec un nouveau chargeur de classe. Il est nécessaire de passer par cette astuce car il est impossible de charger deux fois la même classe avec le même **ClassLoader**.

Cette fonctionnalité peut être fort intéressante pour accélérer la mise à jour du code et ne pas interrompre le service pour cette manoeuvre. Cependant, l'utilisation d'un **ClassLoader** différent de celui utilisé par les autres classes de la même application Web peut poser des problèmes liés au contexte et à la récupération d'éléments partagés car les références utilisées par les autres classes de l'application peuvent ne plus être valides. Si la notion de contexte est importante pour votre application, je vous conseille de désactiver la possibilité du rechargement de classe à la volée ou de ne pas l'utiliser.

3.2.2 L'exécution

L'exécution consiste, pour une Servlet, à accueuilir les connexions initiées par les clients et à traiter leur demande.

En ce qui concerne les Servlets, il est possible de programmer des applications destinées à servir exclusivement les requêtes utilisant le protocole HTTP, ou bien tous types de requêtes. Dans le premier cas, la Servlet doit hériter de la classe **HttpServlet**, dans l'autre, elle hérite de la classe **GenericServlet**, toutes deux appartenant au package j**avax.servlet** .

Pour répondre à une requête générique, le moteur de Servlets appelle la méthode **service** préalablement redéfinie dans le code de votre Servlet. Il est possible d'effectuer tous les traitements que vous désirez à partir de l'appel de cette méthode et de renvoyer la réponse. A cette fin, un objet de type **ServletRequest** possédant toutes les informations sur la requête et un objet de type **ServletResponse** permettant de transmettre les données produites au cours des traitements effectués par la Servlet au client sont passés en paramètres à cette méthode **service**.

Cependant, dans la plupart des cas vous utiliserez une classe héritant de la classe **HttpServlet** pour programmer vos Servlets.

Lorsqu'une requête est reçue par le moteur de Servlet, celui-ci examine d'abord de quel type de requête (on parle de méthode) il s'agit . Il en existe peu en ce qui concerne le protocole HTTP. Ce sont :

- GET : afin de demander au serveur de nous transmettre un flux d'information. Cela peut être un fichier statique ou des informations créées dynamiquement (c'est le cas des servlets ou tout autre solution de programmation web côté serveur).
- POST : même utilité que GET mais les paramètres de la requête sont passés directement au serveur au lieu d'être concaténés à l'URL qui identifie la ressource à laquelle on désire accéder.
- HEAD : afin de s'informer sur l'entête renvoyée par le serveur lors d'une requête donnée.
- OPTIONS : permet de connaître les services disponibles sur un serveur.
- PUT : permet de déposer un fichier dans l'arborescence du système de fichier localisé sur la machine faisant tourner le serveur Web.
- DELETE : permet de supprimer un fichier de l'arborescence du système de fichier localisé sur la machine faisant tourner le serveur Web.
- TRACE : permet de "pister" une requête afin de voir ce que voit le serveur.

Une fois que la méthode de la requête est identifiée, le moteur de Servlet appelle la méthode adéquate pour la Servlet à laquelle la requête tente d'accéder. Pour cela il appelle d'abord la méthode **service** de la classe qui se charge alors d'appeler la méthode adéquate. Voici la table de correspondance entre la méthode HTTP et la méthode appelée par le moteur de Servlets :

Méthode HTTP	Méthode appelée par le moteur de Servlets
GET	doGet()
POST	doPost()
HEAD	doHead()
OPTIONS	doOptions()
PUT	doPut()
DELETE	doDelete()
TRACE	doTrace()

Deux objets sont passés à ces méthodes lorsqu'elles sont appelées :

- un objet de type **HttpServletRequest** : il possède tous les renseignements sur les paramètres passés à la requête.
- un objet de type **HttpServletResponse** : il permet d'obtenir un flux de sortie pour communiquer la réponse au client.

Vous pouvez définir plusieurs de ces méthodes dans une servlet. Par exemple dans le cas où une servlet soit appelée par un formulaire qui peut utiliser soit la méthode POST soit la méthode GET pour transmettre sa requête, vous définirez le code de la méthode doGet et de la méthode doPost. Les méthodes présentes dans le tableau constituent l'équivalent de la méthode **main** pour une application classique ou la méthode **start** pour une applet. En fait la méthode main est la méthode **service** mais elle appelle automatiquement la méthode de

votre classe correspondant à la méthode HTTP utilisée par le client qui effectue la requête si vous ne la surchargez pas, ce qui est le cas le plus courant. On peut donc dire que du point de vue du programmeur, la méthode **service** est négligeable (dans le cas de servlets HTTP).

On remarque également que chacune de ces méthodes peut générer une exception, ce qui suppose que des erreurs peuvent survenir. Que se passe-t-il à ce moment là ? Tout dépend de l'exception qui est générée. Si c'est une exception du type **ServletException**, c'est le signe d'une erreur lors du traitement de la requête effectué par la méthode en question (**doXXX** ou **service**) et le moteur de servlets doit prendre les mesures appropriées à l'annulation de la requête (terminer le thread créé pour l'occasion, etc.). Si l'erreur produite nécessite une suspension du service fourni par la servlet, c'est une exception du type **UnavailableException** qu'il faut générer. Si l'on déclare que cette suspension est permanente, grâce au paramètre entier du constructeur de l'exception, le moteur de servlets supprime la servlet des servlets actives, appelle sa méthode **destroy** et libère l'espace mémoire alloué à l'instance de la servlet. Si le délai passé en paramètre au constructeur de l'exception de type **UnavailableException** est positif, alors le moteur de servlets peut choisir de ne plus router les requêtes dont la servlet est la cible durant la période passée en argument par le programmeur. Durant la période de l'indisponibilité, le moteur de servlets établit une réponse contenant le code HTTP 503 (service non disponible) pour toute requête dont la cible est la servlet qui a généré l'exception. Il est utile d'adopter ce type de comportement lorsque, par exemple, le SGBD utilisé par la servlet pour enregistrer les données et qui est situé sur une autre machine n'est plus disponible.

Lors de l'apparition d'une erreur, il peut être utile d'informer le serveur web et l'administrateur de celui-ci qu'une erreur s'est produite. Il est également pratique, en vue d'un débogage, de disposer de la trace précise des erreurs qui sont survenues. Les serveurs disposent en général d'un système de fichiers journaux pour cela, et les moteurs de servlets ne font pas exception à la règle. Pour utiliser ces fichiers journaux, il est recommandé d'utiliser la méthode **log(String message)** ou **log(Exception e, String message)** de l'interface **ServletContext**. La première écrit le message **message** dans le journal du serveur, la deuxième ajoute à ce message la trace de la pile (le résultat de **e.printStackTrace()**).

3.2.3 La destruction

Pour certaines raisons, le moteur de servlets peut décider s'il doit conserver ou non l'instance (ou les instances) de la Servlet au sein de la machine virtuelle. L'instance d'une Servlet peut être détruite quelques millisecondes après sa création ou après l'arrêt du moteur de Servlets. Rien ne peut contraindre le moteur de Servlets à ne pas détruire l'instance d'une Servlet.

Une fois que le moteur de Servlets a déterminé qu'il n'est plus nécessaire de conserver une instance de la Servlet (par exemple quand le serveur doit libérer de l'espace mémoire, ou lorsque l'exécution du moteur de Servlets est arrêtée), il doit permettre à la Servlet de libérer les ressources qu'elle a acquises au cours de son exécution (que ce soient les buffers utilisés pour l'écriture en sortie, les fichiers lus et écris ou encore les objets instanciés) et de sauvegarder ce qui doit

être persistant. C'est pour cela que le moteur de Servlets appelle la méthode **destroy** de l'interface **Servlet**, implémentée par toutes vos Servlets (voir 2.3 page 25). Avant cela, tous les threads qui ont été créés au sein de la méthode **service** de la Servlet doivent soit être terminés, soit dépasser un temps défini par le moteur de Servlets.

Une fois la méthode **destroy()** appelée, le moteur de Servlets ne doit plus router les requêtes provenant des utilisateurs vers la Servlet concernée. Dans le cas où le moteur de servlets pense que l'instance de la servlet puisse être utile à nouveau, il doit créer une nouvelle instance de celle-ci et poursuivre la destruction de la précédente. Une fois que la fin du corps de la méthode **destroy()** est atteinte, le moteur de servlets doit détruire l'instance de la servlet et rendre l'objet elligible par le ramasse-miettes de la machine virtuelle.

Vous disposez maintenant d'une vue d'ensemble suffisante pour comprendre les différentes étapes que devront emprunter les servlets que vous programmerez. Je vais maintenant aborder un autre concept important qui influe sur le fonctionnement des servlets que vous programmerez au sein du moteur de servlets que vous utilisez : les contextes.

3.2.4 Les contextes

Lorsque vous développez des Servlets, vous développez des applications différentes qui répondent à des exigences différentes et qui ont donc des besoins différents. Par exemple, vous pouvez mettre à disposition des visiteurs de votre site un forum de discussions, une application de configuration du serveur à distance (lorsque vous êtes en congé et que votre patron vous appelle en urgence pour redémarrer le serveur Web) et votre site Web principal. Toutes ces applications sont indépendantes l'une de l'autre car elles n'utilisent pas les mêmes ressources. Chaque classe correspondant à chaque composant de l'application doit rester visible seulement pour les autres classes de la même application. Par contre, au sein d'une même application, il doit être possible de partager des données facilement.Par exemple, dans l'application de commerce en ligne, une servlet permet aux utilisateurs de s'identifier au moment d'entrer sur le site. Lorsque l'un d'eux effectue un achat, c'est une autre servlet de l'application de commerce en ligne sui sera executée, et elle pourra puiser dans la liste des utilisateurs identifiés pour vérifier que l'acheteur s'est bien identifié. Pour toutes ces raisons, le concept de contexte a été créé.

Définition d'un contexte

Un contexte constitue pour chaque servlet d'une même application une vue sur le fonctionnement de cette application. Une application web peut être composée de :

- servlets
- JSP
- classes utilitaires
- documents statiques (pages html, images, sons, etc.)
- beans, applications clients

– méta informations décrivant la structure de l'application

Dans le code source d'une servlet, un contexte est représenté par un objet de type **ServletContext** qui peut être obtenu par l'intermédiaire d'un objet de type **ServletConfig**, par exemple de la façon suivante :

```
public void init(ServletConfig config) {
  ServletContext contexte = config.getServletContext();
  /* suite du code */
}
```

Grâce à ce contexte, il est possible d'accéder à chacune des ressources de l'application web correspondant au contexte. Il faut noter qu'à une application correspond un et un seul contexte, et qu'à un contexte correspond une et une seule application. On peut donc en déduire que chaque contexte est propre à une application et qu'il n'est pas possible de partager des ressources entre applications différentes. Par exemple la servlet **ServletLogin** de l'application de commerce en ligne ne peut pas accéder aux instances de servlets de l'application de configuration du serveur web. Les ressources qu'il est possible de partager sont :

– des documents statiques. Vous pouvez accéder à n'importe quel document statique d'un contexte grâce à la méthode **getRessource** ou à **getRessourceAsStream**.Le chemin passé en paramètre est interprété de façon relative à la racine de l'application correspondant au contexte en cours. Cela rejoint ce que j'ai dis précédemment : une servlet ne peut accéder qu'aux ressources de l'application à laquelle elle appartient quand elle utilise un contexte.
– des instances de servlets : en utilisant la méthode **getServlet**
– des paramètres d'initialisation pour toute l'application. En utilisant la méthode **getInitParameter**, il est possible de connaître la valeur d'un paramètre d'initialisation si on possède son nom. Pour connaître tous les paramètres d'initialisations définis, il faut utiliser la méthode **getInitParameterNames**.
– des attributs d'instance de servlets. Il est ainsi possible de partager des données au sein d'une même application. Pour cela, utilisez les méthodes **setAttribute** et **getAttribute**, en fournissant à chacune de ces méthodes le nom que devra avoir l'attribut au sein de l'application. Pour obtenir le nom de tous les attributs partagés, vous devez utiliser la méthode **getAttributeNames** et pour retirer un attribut, il suffit d'utiliser la méthode **removeAttribute**.

Vous devriez maintenant comprendre ce qu'est un contexte. C'est une abstraction supplémentaire qui permet de matérialiser les relations privilégiés que connaissent les modules d'une même application et le fait que chaque application est différente. Je vais maintenant décrire comment mettre en place plusieurs contextes différents avec le moteur de servlets Tomcat.

Configuration avec Tomcat

A chaque contexte correspond une arborescence dans le système de fichiers qui contient les ressources accédées lors des requêtes vers le moteur de servlets. Cette arborescence est identique pour chaque contexte. Voici comment se décompose la structure des répertoires :

- la racine : elle fait office de répertoire racine pour les ressources qui font partie du contexte. Par exemple l'URL http ://www.monserveur.fr/commerce/index.html fait référence au fichier index.html de ce répertoire racine. Vous pouvez créer les répertoires que vous désirez ou déposer les fichiers que vous voulez dans ce répertoire, comme par exemple un répertoire images/ qui sera accessible via l'URL http ://www.monserveur.fr/commerce/images/ .
- le répertoire WEB-INF : situé à la racine, il contient un fichier web.xml qui est le descripteur de déploiement du contexte. Il contient tous les paramètres de configuration utilisés par le contexte.
- le répertoire WEB-INF/classes/ : c'est le répertoire dans lequel vous déposez les classes de vos servlets et des classes personnalisées utilisées par celles-ci. Le chargeur de classe de l'application vient chercher les classes à charger dans ce répertoire.
- le répertoire WEB-INF/lib/ : il permet de déposer les archives au format jar (Java ARchive Files) qui contiennent des servlets, des beans ou des classes utilitaires utilisées par l'application. Le chargeur de classe de l'application recherche aussi dans ces archives pour trouver les classes dont il a besoin.

Toute cette arborescence peut être regroupée dans une archive (compressée ou non) de la même manière qu'une application Java classique, en utilisant l'utilitaire jar (pour Java ARchive tool). De cette façon il n'y a plus qu'un seul fichier à manipuler et votre application peut-être signée, afin de la rendre digne de confiance auprès des gens qui utiliseront votre application web. En général, avant de procéder à l'archivage de l'arborescence de tout un contexte, il faut créer à sa racine un répertoire META-INF et y placer un fichier MANIFEST.MF contenant les informations nécessaires à l'outil jar, afin qu'il puisse trouver tous les composants de l'application au moment voulu (lors d'une requête par exemple). Il est dit dans les spécifications des servlets que ce répertoire ne doit pas être mis à disposition du client par le serveur Web qui transmet les requêtes au moteur de servlets (ou par le moteur de servlets lui-même s'il est utilisé comme serveur web en même temps). Le contenu de ce répertoire ne sera donc pas visible de l'extérieur à moins que vous en décidiez autrement. Pour archiver une application, il suffit d'entrer la commande :

jar cfvm application.war META-INF/ MANIFEST.MF -C /usr/local/tomcat/webapps/appli/

ce qui produira le fichier application.war archivant l'application dont la racine est située dans le répertoire /usr/local/tomcat/webapps/appli/ et les informations sur l'archive contenues dans le fichier MANIFEST.MF situé dans le répertoire META-INF de la racine de cette application.

Vous savez maintenant que le fichier web.xml présent dans $APP/WEB-INF/ (où $APP correspond au répertoire de votre application web, par exemple /usr/local/tomcat/commerce) permet de spécifier les directives de configuration de l'application web. Voyons les concepts de base de la configuration d'une application web (et donc d'un contexte).

La configuration d'un contexte

Le fichier web.xml est écrit en XML (vous vous en doutiez) et sa structure obéit aux règles d'une DTD définie spécifiquement par Sun. La DTD définit

simplement quelles sont les balises qui sont utilisables à tel ou tel endroit du fichier de configuration. Vous devez préciser que vous utilisez cette DTD en haut du fichier web.xml de chaque application en ajoutant :

```
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
2.2//EN" "http ://java.sun.com/j2ee/dtds/web-app_2.2.dtd">
```

Les informations données par le fichier de configuration sont :

- les paramètres d'initialisation du contexte.
- la configuration de la session.
- les définitions des servlets et des JSPs.
- les correspondances entre servlets et entre JSPs.
- les correspondances entre types MIME.
- la liste des fichiers de bienvenue.
- les pages d'erreur.
- la sécurité.

Je ne vais pas aborder tous ces aspects de la configuration, mais sachez qu'il est possible de régler tous ces paramètres au sein d'un contexte. Je vous présente maintenant la configuration d'un contexte de base correspondant à notre application de configuration du serveur Web à distance.

```
<webapp>
  <!-- nom de l'application -->
  <display-name>
  Application de configuration d'un serveur Web à distance
  </display-name>
  <!-- paramètres disponibles dans le contexte de l'application -->
  <context-param>
    <param-name>Webmaster</param-name>
    <param-value>darktigrou@bigfoot.com</param-value>
  </context-param>
  <!-- définition de la servlet de login -->
  <servlet>
    <servlet-name>login</servlet-name>
    <servlet-class>ServletLogin.class</servlet-class>
    <!-- paramètres d'initialisation de la servlet -->
    <init-param>
      <pram-name>maxlogin</param-name>
      <param-value>1</param-value>
    </init-param>
  </servlet>
 <!-- fin de la définition de la servlet -->
 <!-- correspondance entre URLs et servlets appelées -->
 <servlet-mapping>
   <!-- toutes requête effectuée vers une URL comprenant "/catalog/"
```

```
    est redirigée vers la servlet login -->
    <servlet-name>login</servlet-name>
     <url-pattern>/login/*</url-pattern>
   </servlet-mapping>
   <session-config>
     <session-timeout>30</session-timeout>
   </session-config>
   <mime-mapping>
     <extension>pdf</extension>
     <mime-type>application/pdf</mime-type>
   </mime-mapping>
   <!-- définition des fichiers utilisés par défaut lors d'une requête concernant
   un URL ne spécifiant pas de fichier -->
   <welcome-file-list>
     <welcome-file>index.jsp</welcome-file>
     <welcome-file>index.html</welcome-file>
     <welcome-file>index.htm</welcome-file>
     <welcome-file-list>
   <!-- définition des pages d'erreur -->
   <error-page>
     <!-- pour une erreur de code HTTP 404, c'est le fichier 404.html présent à
     la racine du contexte qui est transmis au client -->
     <error-code>404</error-code>
     <location>/404.html</location>
   </error-page>
 <!-- fin de la configuration du contexte -->
 </web-app>
```

Les différentes choses à savoir à propos des contextes

Les contextes sont très intéressants pour classer et organiser correctement vos applications. Cependant, l'implémentation des contextes dans la majorité des moteurs de servlets ne permet pas d'être certain qu'ils adopteront le comportement que l'on attend d'eux dans certaines circonstances.

C'est le cas lorsqu'on utilise le rechargement des servlets, mais j'ai déjà évoqué le problème précédemment (voir 3.2.1 page 42).

C'est aussi le cas lorsque vous travaillez dans un environnement distribué qui utilise plusieurs machines virtuelles pour des besoins de répartition de charge ou de tolérance de panne. Dans ce cas là même si deux JVM exécutent la même application, deux contextes différents sont utilisés : il existe un seul contexte par application et par JVM.

Vous pouvez également utiliser la possibilité de "virtual hosting" de votre serveur Web. Ce procédé consiste à assigner plusieurs hôtes logiques à la même machine (par exemple http ://www.linux.com/ et http ://www.linux.net/ peuvent être

deux sites différents hébergés par le même serveur web). Lorsqu'un tel procédé est utilisé, un contexte différent est créé pour chaque hôte virtuel ou chaque application dans le même hôte virtuel (ou hôte logique) : un contexte ne peut pas être partagé entre deux ou plusieurs hôtes virtuels.

Chapitre 4

Les servlets HTTP

Vous devriez maintenant avoir acquis assez de connaissance en ce qui concerne le fonctionnement des servlets et de la couche logicielle en charge de leur fonctionnement : le moteur de servlets. Ces connaissances vont vous permettre de comprendre plus facilement certaines caractéristiques de ces dernières lorsque nous aborderons la programmation plus en détail.

Ce chapitre est le premier de ceux qui se consacrent à l'apprentissage de la programmation exclusivement. Les mécanismes sous-jacents seront abordés lorsque cela sera nécessaire mais la priorité est donnée à l'apprentissage de l'utilisation de la bibliothèque de classes mise à votre disposition pour créer vos applications Web.

Comme nous l'avons vu précédemment, la majorité des servlets sont destinées à étendre un serveur Web. Vous savez également que les applications Web utilisent, pour communiquer, un protocole nommé HTTP dont le fonctionnement se décompose en deux grandes phases : la requête et la réponse à cette requête. Nous allons donc étudier les moyens mis à notre disposition pour obtenir le plus d'informations utiles à propos d'une requête effectuée par un client (traditionnellement un navigateur Web) et ceux mis à notre disposition pour organiser notre réponse.

4.1 La gestion des requêtes

Lorsqu'un client envoie une requête à une des servlets que vous venez de développer, la servlet peut avoir besoin de connaître trois types d'informations :

- les informations sur le serveur.
- les informations sur le client qui a servi à effectuer la requête.
- les informations sur la requête elle même.

Les informations à propos du serveur

Les informations sur le serveur peuvent servir à connaître les bons chemins vers les ressources disponibles sur celui-ci, à connaître la version du serveur

utilisée ou encore son adresse IP. Cela peut permettre à vos servlets de s'adapter facilement dans le cas d'un déploiement sur des serveurs différents par exemple. Contrairement à des méthodes de développement comme CGI, ces informations sont accessibles via des méthodes appartenant à l'objet qui représente la requête (de type **HttpServletRequest** pour une servlet HTTP) ; Cela permet d'éviter les problèmes de typographie lors de l'obtention de ces informations, car le nom des méthodes est vérifié à la compilation et donnera une erreur si une des méthodes est mal orthographiée. Par exemple :

```
$port_serveur = $ENV{'SERVER_PRT'};
print "Le serveur écoute sur le port $port_serveur\n";
```

ne donnera pas la bonne information alors qu'apparemment tout est correct (même si vous avez remarqué l'erreur, ce dont je ne doute pas, faites semblant ou imaginez un code source de quelques milliers de lignes), et ceci sans aucun message d'erreur (à moins que l'on utilise des options comme le module strict, mais ce n'est pas le comportement par défaut). Par contre, dans une servlet :

```
int port_serveur = requete.getServerPrt();
out.println("Le serveur écoute sur le port " + port_serveur + "\n");
```

ce code générera une erreur de compilation, et vous corrigerez alors l'erreur sans aucun délai. Voici juste une illustration de ce que peut apporter le typage fort du langage Java par rapport à d'autres. A chaque variable d'environnement disponible avec CGI correspond une méthode associée à la requête pour obtenir des informations. Celles qui concernent le serveur sont :

- **getServerName()** : donne le nom du serveur, tel qu'il est donné dans la directive **ServerName** du fichier de configuration d'Apache (httpd.conf).
- **getServletContext().getServerInfo()** : retourne le nom du logiciel utilisé pour prendre en charge la requête , par exemple Tomcat/3.2.1.
- **getServletContext().getAttribute(String nom_attribut)** : permet de récupérer les différentes parties de l'information sur le logiciel serveur. Chaque moteur de servlets ou serveur Web possède ses propres noms d'attribut, les énumérer tous ici serait inutile. Je vous conseille donc de vous reportez à la documentation spécifique à votre moteur de servlets pour connaître les noms de ces attributs et leur signification.
- **getServerPort()** : retourne le port sur lequel écoute le serveur Web qui a pris en charge la requête.
- **getRealPath(String nom_fichier)** : retourne le chemin réel vers le fichier nom_fichier. En principe, ce chemin doit correspondre à **getRealPath("/") + nom_fichier**. C'est assez utile lorsque vous désirez effectuer des entrées/sorties vers des fichiers présents sur votre serveur.
- **getPathTranslated()** : cette méthode permet de connaître le chemin réel vers le fichier sur le système de fichier du serveur lorsqu'un fichier est passé à la servlet. Par exemple, pour une URL comme celle-ci : `http://www.monserveur.com/mon_application/servlet/MaServlet/index.html`, **getPathTranslated()** renverra la même chose que **getRealPath("/index.html")**.

Voici ci-dessous un programme classique affichant toutes les informations intéressantes d'un serveur et quelques autres informations utilisant les méthodes que nous venons de voir :

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
public class ServeurInfo extends HttpServlet {
  public void doGet(HttpServletRequest req, HttpServletResponse res) throws
ServletException, IOExcepption {
    res.setContentType("text/plain"); // on produit du texte ASCII
    PrintWriter out = res.getWriter();
    out.println("Nom du serveur : " + req.getServerName() + " .");
    out.println("Logiciel utilisé : " + req.getServletContext().getServerInfo() +
                " .");
    out.println("Port du serveur : " + req.getServerPort() + " .");
    out.println("Port du serveur : " + req.getServerPort() + " .");
    out.println("Chemin vers le fichier " + req.getPathInfo() + " : " +
                req.getPathTranslated(req.getPathInfo()) + " .");
  }
}
```

Une autre servlet se servant de ces méthodes pourrait servir à retourner n'importe quel fichier passé en paramètre de la requête :

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
public class FileServer extends HttpServlet {
 // on peut être amené à envoyer des fichiers binaires, donc on utilise
 // un ServletOutputStream au lieu d'un PrintWriter
 ServletOutputStream out = res.getOutputStream();
 // récupération d'une référence sur le fichier demandé
 File fichier = new File(req.getPathTranslated();
 if (fichier == null)
  // si le fichier est introuvable on envoie un code d'erreur 404
  res.sendError(HttpServletResponse.SC_NOT_FOUND);
 // sinon le type de contenu de la réponse correspond au type MIME
 // du fichier
 res.setContentType(getServletContext().getMimeType(fichier));
   try {
     // on utilise un tampon de 4 ko pour lire le fichier
     // ce qui est plus rapide qu'un lecture ligne par ligne
     char[] tampon = new char[4 * 1024];
     FileInputStream in = new FileInputStream(fichier);
     while (in.read(tampon) >= 0) {
       out.write(tampon);
     }
   } catch (IOEXception e) {
```

```
        // si une erreur se produit au milieu de la réponse
        // on envoie le code d'erreur HTTP adéquat
        res.sendError(HttpServletResponse.SC_PARTIAL_CONTENT);
      } finally {
        if (fichier!= null)
          fichier.close();
      }
    }
  }
```

Vous avez pris connaissance des principales méthodes permettant de connaître des informations relatives au serveur sur lequel tourne une servlet. Pour en savoir plus, je vous invite à consulter les méthodes de l'API de J2EE contenues dans l'interface **ServletResponse** pour en connaître d'autres.

Les informations à propos du client

Nous avons maintenant besoin d'informations sur le client qui a effectué la requête. En effet, vous pouvez penser qu'il est utile de connaître l'adresse IP de la machine qui a effectué la requête pour des besoins d'authentification ou bien que la version du navigateur utilisée influe sur les données à transmettre. Ces informations sont accessibles via l'interface **ServletRequest** comme pour les informations du serveur. Voici une liste des principales méthodes concernant la requête de cette interface :

- **getRemoteAddr()** : récupère l'adresse IP de l'hôte qui a initié la requête.
- **getRemoteHost()** : renvoie le nom d'hôte de la machine qui a initié la requête. Si le nom d'hôte ne peut pas être récupéré, la représentation de l'adresse IP du client est renvoyé sous forme de chaîne de caractères.

Ces requêtes permettent d'identifier une machine sur le réseau Internet et de disposer ainsi d'une méthode d'authentification au niveau de la machine. Cependant, ces méthodes restent basiques et ne permettent pas de disposer d'un système d'authentification robuste. Pour cela, il est nécessaire de mettre en place des systèmes plus sûrs et fiables basés par exemple sur le protocole HTTPS par exemple, et qui peuvent se servir des méthodes sus-citées.

Voici une servlet qui, en s'inspirant de la servlet qui sert les fichiers, met en place un mécanisme basique de contrôle des accès :

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
public class FileServer extends HttpServlet {
  private final static String fichierIP = "ip.txt";
  private final static String fichierHotes = "hotes.txt";

  public void doGet(Httpservletrequest req, Httpservletresponse res) throws
ServletException, IOException {
```

```
// on peut être amené à envoyer des fichiers binaires, donc on utilise
// un ServletOutputStream au lieu d'un PrintWriter
ServletOutputStream out = res.getOutputStream();
if (verifieIp(req.getRemoteAddr()) || verifieHote(req.getRemoteHost()) {
  // la machine cliente est autorisée à accéder à la servlet
  // récupération d'une référence sur le fichier demandé
  File fichier = new File(req.getPathTranslated();
  if (file == null)
  // si le fichier est introuvable on envoie un code d'erreur 404
  res.sendError(HttpServletResponse.SC_NOT_FOUND);
  // sinon le type de contenu de la réponse correspond au type MIME
  // du fichier
  res.setContentType(getServletContext().getMimeType(file));
    try {
      // on utilise un tampon de 4 ko pour lire le fichier
      // ce qui est plus rapide qu'un lecture ligne par ligne
      char[] tampon = new char[4 * 1024];
      FileInputStream in = new FileInputStream(file);
      while (in.read(tampon) >= 0) {
        out.write(tampon);
      }
    } catch (IOEXception e) {
      // si une erreur se produit au milieu de la réponse
      // on envoie le code d'erreur HTTP adéquat
      res.sendError(HttpServletResponse.SC_PARTIAL_CONTENT);
    } finally {
      if (file!= null)
        file.close();
    }

   } else {
   out.println("Vous n'êtes pas autorisé à accéder à cette servlet.");
 }

}

public boolean verifieIP(String ipRequete) {
/* cette méthode vérifie que l'adresse Ip de la machine
 * cliente qui veut récupérer le fichier est explicitement
 * autorisée (son adresse Ip est présente dans le fichier
 * fichierIp);
 */
  File f = new File(fichierIP);
  FileInputStream in = new FileInputStream(f);
```

```
      String s ;
      while ((s = in.readLine()) != null) {
        // l'ip du client est-elle autorisée ?
        if (s.equals(ipRequete))
          return true ;
        return false ;
      }
    }

    public boolean verifieHote(String hoteRequete) {
    /* cette méthode vérifie que le nom de la machine
     * cliente qui veut récupérer le fichier est explicitement
     * autorisée (son nom d'hôte est présent dans le fichier
     * fichierHotes) ;
     */
      File f = new File(fichierHotes) ;
      FileInputStream in = new fileInputStream(f) ;
      String s ;
      while ((s = in.readLine()) != null) {
        // l'hote est-il autorisé ?
        if (s.equals(hoteRequete))
          return true ;
        return false ;
      }
    }
```

Gardez bien en tête que cette Servlet ne constitue en aucun cas un mécanisme de sécurité valable. Elle illustre modestement l'emploi des méthodes relatives aux informations sur la machine cliente. En effet, toute vulnérabilité applicable au protocole IP est exploitable ici, comme le "spoofing".

Les informations sur la requête

Nous allons maintenant aborder l'analyse de la principale source d'information : la requête elle-même.

Plusieurs catégories d'informations sont importantes au niveau de la requête. Ces catégories correspondent aux différentes caractéristiques d'une requête HTTP. Nous pouvons avoir besoin d'informations sur :

- l'URL sur lequel porte la requête. Par extension on peut obtenir l'URI, qui n'est qu'un concept plus abstrait que celui d'URL[1].
- le schéma utilisé par la requête. La méthode HTTP (GET, POST, etc.), le protocole utilisé et sa version (HTTP/1.0, HTTP/1.1) et enfin l'entête d'URL (http ://, https ://, etc.) font partie du schéma.

[1] Reportez vous au document disponible à `http://www.ietf.org/rfc/rfc1630.txt` pour plus d'informations sur les différentes formes d'URI, ce qui inclue les URLs.

- les paramètres passés à la requête. Ce sont les données envoyées par le client avec la requête qui peuvent servir à préciser l'objet de la requête ou à envoyer des données au serveur (dans le cas de l'envoi de fichier par exemple).
- l'entête de la requête : le client peut apporter diverses précisions avec la requête, comme le format de données qu'il accepte, le logiciel du client (le nom du navigateur web par exemple) ou encore l'URL d'où provient la requête (si l'utilisateur a cliqué sur un lien pour effectuer la requête).

Pour obtenir l'URL de la requête, on utilise la méthode **getRequestURL()** de la classe **HttpUtils** du package **javax.servlet.http**. Cette méthode reconstruit l'URL complet de la requête, sous une forme telle que http ://www.monserveur.com/MaServlet par exemple. Cet URL est renvoyé sous forme d'objet de type **StringBuffer** et est donc modifiable. L'URL qui est retourné peut être différent de celui entré dans le navigateur web du client car tout y est : le nom du serveur, le port, l'entête (http ://), alors qu'un utilisateur ne précise pas le port sur lequel tourne le serveur si celui ci écoute sur le port 80 (ou sur d'autres ports selon la configuration des navigateurs). Les caractères spéciaux présents dans les paramètres de la requête (après le " ?" par exemple pour une requête utilisant la méthode GET) peuvent aussi être différents par rapport à ceux présents dans le champ de saisie de l'URL du navigateur : les "%20" seront par exemple remplacés par des espacements. Parfois cependant, il n'est pas nécessaire d'avoir des informations sur l'URL entier, mais seulement sur la partie concernant la ressource demandée. Pour cela vous pouvez utiliser la méthode **getRequestURI()** de l'interface **HttpServletRequest**. Cette méthode retourne l'Uniform Ressource Identifier de la requête. Pour des requêtes portant sur des ressources statiques (pages HTML par exemple), l'URI correspond exactement à l'URL sans l'entête (http :// par exemple), le nom du serveur et le numéro de port. Par exemple, pour l'URL suivant : http ://www.monserveur.com/index.html, l'URI correspondant est /index.html. Pour des URLs référençant un contenu dynamique (des servlets par exemple), le procédé de traduction entre URL et URI est parfois plus subtil. Voici une liste d'exemples qui vous permettra de mieux cerner ces subtilités :

- l'URL http ://www.monserveur.com/MaServlet correspond à l'URI /MaServlet.
- l'URL http ://www.monserveur.com/MaServlet ?param1=valeur1 correspond à l'URI /MaServlet.
- l'URL http ://www.monserveur.com/MaServlet/info_chemin ?param1=valeur1 correspond à l'URI /MaServlet/info_chemin.
- l'URL http ://www.monserveur.com/index.html correspond à l'URI /index.html.

Pour être encore plus précis, il est possible de connaître le chemin de la servlet correspondant à un URL donné. Il suffit d'utiliser la méthode **getServletPath()** de l'interface **HttpServletRequest**. Cette méthode retourne sous la forme d'un objet de type **String** la partie de l'URL qui appelle la Servlet, ou la valeur null si l'URL ne fait pas référence à une servlet. Lorsqu'une servlet fait partie d'une chaîne de servlets, la valeur renvoyée par **getServletPath()** correspond à celui qui aurait été renvoyé par la première servlet de la chaîne (étant donné que ce n'est pas l'URL qui décide quelles servlets doivent faire partie de la chaîne, et que cette URL ne connaît donc que le premier maillon de cette chaîne). Voici une liste d'exemples qui illustre ce que je viens de dire :

- Si l'URL est http ://www.monserveur.com/MaServlet, la méthode **getServletPath()** renvoie "/servlet/MaServlet".
- Si l'URL est http ://www.monserveur.com/MaServlet ?param=valeur, la méthode **getServletPath()** renvoie "/servlet/MaServlet".
- Si l'URL est http ://www.monserveur.com/MaServlet/info_chemin ?param=valeur, la méthode **getServletPath()** renvoie "/servlet/MaServlet".
- Si l'URL est http ://www.monserveur.com/index.html, la méthode **getServletPath()** renvoie null.
- Si l'URL est http ://www.monserveur.com/alias_servlet.html, la méthode **getServletPath()** /alias_servlet.html. Ici alias_servlet.html est un alias sur une servlet. C'est à dire que l'on précise au serveur web que toute requête demandant le document alias_servlet.html de la racine de son arborescence doit être traduite en une requête vers la servlet associée.

Pour une Servlet invoquée via SSI (voir 2.4.1 page 27), **getServletPath()** renvoie null si la servlet a été incluse dans un document statique, ou le chemin vers la première servlet d'une chaîne si elle fait partie de cette chaîne de servlets.

En ce qui concerne le schéma utilisé par la requête, il existe de nombreuses méthodes permettant d'obtenir des informations. La méthode **getSheme()** permet de récupérer l'entête de la requête : par exemple "https" si le client a demandé une connexion sécurisée. La méthode **getProtocol()** permet de récupérer le protocole utilisé par la requête (par exemple "HTTP/1.0"). La méthode **getMethod()** permet de connaître la méthode HTTP utilisée. La valeur renvoyée peut être soit "GET", "POST" ou toute autre méthode HTTP (voir 2.1.2 page 15).

Les paramètres d'une requête sont compris dans celle-ci. Chaque paramètre possède un nom et une valeur. Les paramètres sont passés différemment au serveur selon que la requête utilise la méthode HTTP GET ou POST. Dans le cas de l'utilisation de la méthode POST, les paramètres sont directement envoyés au serveur comme des données classiques, et le serveur les récupère sur son entrée standard. Dans le cas où la requête utilise la méthode HTTP GET, les paramètres ainsi que leur valeur sont passés au serveur en ajoutant ces informations à l'URL. Par exemple, pour passer un paramètre "nom " de valeur "gilli" à une servlet nommée MaServlet, l'URL correspondante sera : http ://www.monserveur.com/servlet/MaServlet ?nom=gilli . Chaque paire nom=valeur est séparée par un "&" de la paire suivante. Pour passer deux paramètres nom et prénom, l'URL prendra la forme suivante (toujours si la requête utilise la méthode HTTP GET) :

http ://www.monserveur.com/servlet/MaServlet ?nom=gilli&prenom=julien

Les caractères comme l'espace ou le signe "%" sont remplacés par leur code hexadécimal correspondant, ou "+" pour l'espace. Il faut alors décoder ces caractères spéciaux pour recueillir des informations correctes.

Ces paramètres peuvent être récupérés de plusieurs façons. Tout dépend de la connaissance que l'on peut avoir des paramètres passés. Dans une requête utilisant la méthode HTTP POST, l'utilisateur ne peut entrer "à la main" des paramètres dont on ne pourrait connaître à l'avance le nom et la valeur. Dans ce cas il n'est pas utile de pouvoir récupérer le nom des paramètres passés à la requête. Au contraire, pour une requête utilisant la méthode HTTP GET, les paramètres sont modifiables par le client puisque ajoutés à l'URL qui est visible

dans le champ de saisie de l'adresse au niveau du navigateur web. Il peut être alors utile de récupérer le nom de ces paramètres. Ceci est effectué par la méthode **getParameterNames()** de l'interface **ServletRequest**. Cette méthode renvoie un objet de type **Enumeration** que vous devriez savoir manipuler si vous possédez les connaissances requises (voir page 5). Pour obtenir les valeurs de tous les paramètres, vous devez utiliser la méthode **getParameterValues()** qui renvoie un objet de type String[] contenant les valeurs de chaque paramètre. Enfin, la méthode **getParameter(String nomParametre)** permet de connaître la valeur du paramètre **nomParametre** représentée par un objet de type String. Ci-dessous se trouve une servlet permettant d'afficher toutes les informations sur tous les paramètres de la requête :

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;
public class ServeurInfo extends HttpServlet {
  public void doGet(HttpServletRequest req, HttpServletResponse res) throws
ServletException, IOExcepption {
    res.setContentType(“text/plain”); // on produit du texte ASCII
    PrintWriter out = res.getWriter();
    Enumeration parametres = req.getParameterNames();
    out.println(“Affichage des informations sur les paramètres de la
                requête”);
    while (parametres.hasMoreElements()) {
    String nomParametre = (String) parametres.nextElement();
    out.println(“Le paramètre “ + nomParametre + “ a la valeur : “ +
                getParameter(nomParametre) + “ .”);
   }
  }
}
```

Il faut noter qu'il existe une méthode permettant de récupérer tous les paramètres passés à la requête à la fois, c'est à dire toutes les paires paramètre=valeur sans décodage des caractères spéciaux préalable. Ceci n'est pas d'une utilité renversante à mon goût, et je ne pense pas que vous l'utiliserez beaucoup. Sachez cependant que c'est la méthode **getQueryString()** de l'interface **HttpServletRequest** qui peut vous rendre ce service. Toutes les autres méthodes vues au dessus décodent les paramètres pour vous avant de les renvoyer.

Les entêtes de la requête précisent diverses choses comme :

- le logiciel utilisé par le client.
- les types MIME des formats de données acceptés.
- le type d'authentification utilisé dans le cas d'une connexion sécurisée.
- la date de dernière requête du client pour la ressource demandée.
- le jeu de caractères utilisé par le navigateur.
- la valeur du cookie associé au domaine auquel appartient la servlet demandée.
- le langage utilisé par le client.

Comme pour la récupération des paramètres associés à la requête, il existe plusieurs façons de récupérer la valeur et le nom des en-têtes de requête. A chaque en-tête correspond une méthode distincte qui permet de récupérer la valeur de cette entête en particulier. Ces méthodes sont mentionnées dans la documentation officielle de l'API des servlets[2], dans la section concernant l'interface **HttpservletRequest**. Ensuite il est possible de récupérer une liste des noms des entêtes passés par le client lors de la requête avec la méthode **getHeaderNames()** de l'interface **HttpServletRequest**. Cette méthode renvoie un objet de type **Enumeration**. Une fois le nom de tous les en-têtes récupérés, il suffit de récupérer la valeur de chaque entête avec la méthode **getHeader(String nomEntete)** de la même interface pour obtenir toutes les en-têtes correspond à une requête. La servlet suivante affiche toutes les en-têtes passées au serveur avec la requête :

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;
public class ServeurInfo extends HttpServlet {
  public void doGet(HttpServletRequest req, HttpServletResponse res) throws
ServletException, IOExcepption {
    res.setContentType(“text/plain”); // on produit du texte ASCII
    PrintWriter out = res.getWriter();
    out.println(“Affichage des informations sur les en-têtes de la requête”);
    Enumeration entetes = req.getHeaderNames();
    while (entetes.hasMoreElements()) {
      String nomEntete = (String) entetes.nextElement();
      out.println(“a l’entête “ + nomEntete + “ correspond la valeur “ +
                  getHeader(nomEntete) + “.”);
    }
  }
}
```

Le résultat est différent pour la même requête avec des navigateurs différents, car de nombreuses en-têtes dépendent du navigateur, comme “User-Agent” ou “Accept”. Cette dernière définit le type de fichiers acceptés par un navigateur.

Certaines des valeurs d'en-têtes peuvent être mieux traitées en étant récupérées comme un entier ou un objet de type **Date**. Les méthodes **getIntHeader()** et **getDateHeader()** servent respectivement à cela. Si le format de l'en-tête ne correspond pas au type retourné par l'une de ces méthodes, une **IllegalNumberException** sera générée pour **getIntHeader** et une **IllegalArgumentException** sera générée pour la méthode **getDateHeader()**

Le langage utilisé par le client (correspondant à l'en-tête “Accept-Language”) est disponible via la méthode **getLocale()** de l'interface **ServletRequest**. La méthode **getLocales()** de la même interface permet de récupérer dans l'ordre

[2] Disponible à `http://java.sun.com/j2ee/j2sdkee/techdocs/api/index.html`.

décroissant de préférence les langages acceptés. Ces informations peuvent être utilisées si vous gérez des servlets internationalisées. Pour plus d'informations à ce sujet, consultez la section "internationalisation" du tutoriel Java à l'URL suivant : `http://java.sun.com/docs/books/tutorial/i18n/index.html`.

En ce qui concerne les connexions dites "sécurisées", il est possible d'en connaître plus grâce à la méthode **isSecure()** de l'interface **ServletRequest** qui renvoie un booléen précisant si la requête a été faîte en utilisant un canal sécurisé ou non (comme le protocole HTTPS qui utilise la méthode de cryptage SSL, qui signifie Secure Socket Layer). La méthode utilisée pour établir une connexion sécurisée est obtenue par la méthode **getAuthType()** de la même interface. Cette méthode retourne un objet de type String qui peut prendre des valeurs comme "BASIC" lorsque les fonctionnalités d'identification du serveur web sont utilisées ou "SSL" par exemple. Pour plus de renseignements à propos de la sécurisation des accès HTTP au niveau utilisateur, vous pouvez consulter le document disponible à l'URL `http://www.apacheweek.com/features/userauth`. En ce qui concerne la mise en place du protocole SSL, je vous conseille de consulter le document accessible à l'URL `http://www.apacheweek.com/features/ssl`.

Il existe également des attributs spéciaux qui peuvent être mis en place par le moteur de servlets lui même en plus des attributs dont vous pouvez récupérer le nom avec la méthode **getHeaderNames()** vue juste au-dessus. Chaque moteur de servlets peut décider de mettre en place des attributs spéciaux, mais ce n'est pas obligatoire. Ils sont accessibles via la méthode **getAttributeName(String nomParametre)** de l'interface **ServletRequest**, qui ressemble à la méthode du même nom de l'interface **ServletContext**. Le serveur Java Web Server définit par exemple trois attributs supplémentaires qui ont pour nom **javax.net.ssl.cipher_suite**, **javax.net.ssl.peer-certificates** et **javax.net.ssl.session**. Ils permettent de récupérer des informations sur une requête sécurisée utilisant le protocole HTTPS (et donc SSL). Cette méthode permet également de partager des données au sein du même contexte (voir 3.2.4 page 45). Une seule valeur peut être associée à un seul nom d'attribut. Il est recommandé que les noms d'attributs suivent une convention similaire aux packages du JDK (voir la spécification du langage Java à l'URL `http://java.sun.com/docs/books/jls` pour plus de renseignements).

Enfin, nous terminerons cette section concernant la requête en décrivant comment traiter celles qui permettent d'envoyer des données brutes au serveur. Je vous ai dit précédemment que les requêtes de type HTTP POST étaient passées au serveur web par son flux d'entrée standard (plus communément appelé STDIN, pour STanDard INput). Ce flux d'entrée peut être utile pour d'autres choses : chaîner plusieurs servlets en associant le flux de sortie standard d'une servlet au flux d'entrée standard d'une autre, passer à une servlet non HTTP des données brutes. Par "données brutes", il faut comprendre données binaires comme des fichiers graphiques au format jpeg par exemple. Pour traiter ce flux il est nécessaire de connaître le type de contenu transmis et la longueur du flux. Pour connaître le type MIME du flux disponible sur l'entrée standard, il faut utiliser la méthode **getContentType()** de l'interface **ServletRequest**. Cette méthode renvoie un objet de type String qui correspond à un type MIME, comme par exemple "text/html" ou "image/png". Pour connaître la longueur des données passées à l'entrée standard, il suffit d'employer la méthode **getCon-**

tentLength() de la même interface, qui retourne un entier correspondant à la longueur des données en entrée.

Vous devriez maintenant être en mesure de traiter n'importe quel type de requête. Il faut maintenant réagir à cette requête en envoyant une réponse. Nous étudierons tout d'abord les moyens mis à notre disposition pour émettre une réponse conforme aux caractéristiques d'une réponse HTTP, et nous verrons ensuite les différentes façons de produire du contenu HTML et multimédia (images, animations).

4.2 La réponse

Comme nous l'avons vu en 2.1.2 page 15, la réponse est organisée en plusieurs sections. Les méthodes de l'interface **ServletResponse** (et donc de **HttpServletResponse** puisqu'elle dérive de la précédente) permettent de contrôler chacune des ces sections.

En ce qui concerne le code d'état, l'interface **HttpservleResponse** possède un nombre assez impressionnant d'attribut de classes (statiques) de type entier représentant chacun un code d'état. Afin de transmettre la bonne entête, on utilise la méthode **setStatus(int etat)** en passant comme paramètre un des attributs statique de l'interface. Par exemple, pour notre servlet qui permet de servir des fichiers, nous aurions pu utiliser l'instruction **setStatus(HttpServletRequest.SC_NOT_FOUND)** dans le cas d'un fichier introuvable. Le code d'état **HttpServletRequest.SC_NOT_FOUND** correspond au code d'état HTTP 404. Vous pourrez trouver tout l'inventaire des attributs de l'interface **HttpServletRequest** avec leur code d'état HTTP correspondant dans la documentation de l'API des servlets disponible à `http://java.sun.com/j2ee/j2sdkee/techdocs/api/index.html`.

Les en-têtes HTTP

Une fois le code d'état envoyé par la réponse, un ensemble de paramètres d'entête doivent être donnés. Ces en-têtes permettent au client d'adapter son comportement afin de recevoir correctement les données. Ces en-têtes ne sont pas classables en diverses catégories, et remplissent chacune une fonction bien particulière. Je préfère donc vous donner une énumération de celles-ci en vous signalant leur utilité et comment les manipuler avec les servlets. Notez également qu'il n'est pas nécessaire de préciser la valeur de toutes ces en-têtes, mais seulement de celles qui possèdent une valeur particulière pour votre servlet.

Tout d'abord, l'en-tête "Content-Type" définit le type MIME de contenu que la servlet s'apprête à fournir. La méthode **setContentType(String typeMime)** de l'interface **ServletResponse** permet de définir ce type. Par exemple, l'instruction

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

permet de signaler au client web que le contenu produit est du texte au format HTML.

Un autre exemple d'en-tête est "Location", qui permet de rediriger l'utilisateur. Il est possible de préciser cette en-tête grâce à la méthode **setHeader(String name, String valeurEntete)** de l'interface **HttpServletResponse**. Cette méthode est le pendant de l'accesseur en lecture des en-têtes de requête venant du client : la méthode **getHeader(String name)**. Il est possible de spécifier chaque en-tête HTTP via cette méthode, il suffit juste de connaître le nom de l'entête. Pour cela, je vous invite à consulter DOCUMENT REFERENCE HTTP. Comme pour la méthode **getHeader(String name)**, il est possible de spécifier la valeur d'une en-tête avec un autre type de données que la classe **String**. Pour cela il suffit d'utiliser les paramètres de type **long** ou entier des autres méthodes **setHeader()**. Pour connaître toutes les versions de **la méthode setHeader** possibles, reportez vous à la documentation de référence de l'API des servlets, disponible à `http://java.sun.com/j2ee/j2sdkee/techdocs/api/index.html`.

Afin de ne pas devoir se rappeler de la signification des codes d'états et de leurs en-têtes correspondantes, quelques méthodes sont disponibles et permettent l'envoi de ces deux informations à la fois. Par exemple la méthode **sendRedirect(String nouvelleUrl)** utilisée comme ceci :

```
res.sendRedirect("http ://www.monserveur.com/autreUrl.html")
```

permet d'effectuer le même traitement que les deux instructions :

```
res.setStatus(HttpServletResponse.SC_MOVED_TEMPORARILY);
res.setHeader("Location", "http ://www.monserveur.com/autreUrl.html");
```

Je vous invite à consulter la documentation de référence sur l'API des servlets disponible ici `http://java.sun.com/j2ee/j2sdkee/techdocs/api/index.html` pour connaître les autres méthodes de ce type (comme **sendError**).

Quelques conventions sont à respecter pour utiliser les en-têtes correctement. Tout d'abord, vous ne pouvez pas avoir deux valeurs différentes pour la même entête. C'est pour cela qu'un autre appel à la méthode **setHeader** concernant le même nom d'en-tête écrasera l'ancienne valeur. Ensuite vous devez absolument envoyer les en-têtes avant le corps de la réponse, c'est à dire que l'instruction

```
out.println("envoi de données");
```

suivie de l'instruction

```
res.setHeader("Location", "http ://www.monserveur.com/bidule.html");
```

n'est pas correcte. Il n'est pas certain que cette incorrection génère une erreur. Cela dépend du moteur de servlets qui fait tourner votre servlet. Par exemple le Java Web Server possède un tampon qui stocke une petite quantité de données (quatre kilo octets) avant de les envoyer. Cela vous laisse un peu de temps pour envoyer les en-têtes. Enfin, il vaut mieux définir le type de contenu à envoyer avant d'obtenir le flux de sortie (dont je parle un peu plus tard), ceci

afin d'obtenir un flux de sortie correctement configuré en fonction du type de contenu à envoyer, sans avoir à effectuer aucune manipulation.

Vous pouvez maintenant produire une en-tête de réponse conforme au protocole HTTP de façon à ce que tous les navigateurs comprennent les données que vous fournissez. Voyons maintenant le corps de la réponse.

Envoyer les données

Le corps de la réponse sont les données elles-mêmes. Si une requête porte sur le fichier index.html d'un répertoire donné, le corps de la réponse sera le contenu du fichier index.html. Pour pouvoir écrire des données sur le flux de sortie, il est nécessaire d'obtenir un flux (au sens "Java" du terme). Vous pouvez choisir entre deux types de flux : le **PrintWriter** obtenu par la méthode **getWriter()** de l'interface **HttpServletResponse** ou le **ServletOutputStream** obtenu grâce à la méthode **getOutputStream()** de l'interface **ServletResponse**. Le premier type de flux permet d'écrire efficacement un flux de caractères, destiné à être lu et supportant l'internationalisation. Le deuxième type est conçu pour écrire des données au format binaire. Si vous appelez tentez d'obtenir un flux de type **ServletOutputStream** après avoir obtenu un flux de type **PrintWriter** ou inversement, une exception de type **IllegalStateException** sera générée.

Pour envoyer du contenu au client, il suffit alors d'utiliser les méthodes disponibles selon le flux choisi, par exemple la méthode **println(String contenu)** pour un flux de type **PrintWriter** ou **ServletOutputStream**.

Il faut savoir que la sortie peut être gérée de deux manières : tamponnée ou directe. Une sortie tamponnée permet d'accroître les performances d'écriture sur le flux de sortie. Par contre, les besoins en mémoire vive peuvent s'accroître car une plus grosse quantité de données est gardée en mémoire plus longtemps, et les servlets ne fournissant qu'une ressource (par exemple un seul fichier HTML sans image) très grande (la connexion dure longtemps) ne bénéficierait pas de cette méthode.

La méthode directe quant à elle envoie les données au fur et à mesure qu'elles sont générées (à chaque retour à la ligne).

Aucun moteur de servlets n'est tenu de fournir un mécanisme de tampon, vous devrez donc consulter la documentation du moteur de servlets que vous utilisez pour savoir s'il est possible d'utiliser un tel mécanisme. Pour obtenir des informations sur la sortie tamponnée, plusieurs méthodes existent :

- **getBufferSize()** : fournit la taille du tampon utilisé actuellement pour l'écriture.
- **setBufferSize(int tailleTampon)** : ajuste la taille du tampon à **tailleTampon**. Le moteur de servlets définit alors un tampon au moins égal à la taille donnée. Le fait de disposer d'un peu de liberté pour définir la taille réelle du tampon permet au moteur de servlets de réutiliser un éventuel tampon dont la taille serait supérieure ou égale à la taille demandée. Outre l'accélération des performances en écriture, une taille de tampon supérieure à zéro permet la mise en place des en-têtes un peu après l'envoi des premières données. Cette méthode doit être appelée avant l'envoi de toute information au client (les en-têtes et le code d'état font partie des données ici).

- **isComitted()** : permet de savoir si des données ont déjà été envoyée au client. Vous pouvez encore spécifier le code d'état et les en-têtes de la réponse si cette méthode renvoie faux.
- **reset()** : efface toutes les données présentes dans le tampon si aucun élément de la réponse n'a été encore envoyé. Les en-têtes seront aussi effacées, vous devrez donc en spécifier de nouvelles (mais qui peuvent être identique). Si vous appelez cette méthode alors que des éléments de la réponse ont été envoyé, une exception du type **IllegalStateException** est générée**.**
- **flushBuffer()** : permet de forcer l'envoi des données lorsque le tampon n'est pas encore plein. Après l'appel de cette méthode, le code d'état et l'en-tête HTTP ont été communiquées au client et un appel à la méthode **isCommitted()** renverra la valeur "vrai".

Comme tout tampon utilisé en programmation, lorsqu'il est plein, les données présentes dans ce dernier sont envoyées sur la sortie, et la méthode **isComitted()** renverra la valeur "vrai" à partir de ce moment.

Les données sont envoyées au travers d'une connexion établie entre le client (navigateur web) et le serveur. Or vous savez que tout navigateur est doté d'un bouton "stop" qui arrête le chargement de la page web demandée. Que se passe-t-il à ce moment là ?

La connexion est alors coupée et le flux ouvert en sortie par la servlet vers le naviagetur n'est plus valide. Deux cas sont possible :

- Vous utilisez un flux de type **PrintWriter**.
- Vous utilisez un flux de type **ServletOutputStream**.

Dans le premier cas, l'écriture sur un flux non valide ne génère aucune exception. Pour savoir si le flux est toujours valide, vous devrez utiliser la méthode **checkError()** de la classe **PrintWriter**. Cette méthode retourne le booléen **true** si le flux de sortie n'est plus valable.

Dans le deuxième cas, l'écriture sur un flux non valide génère une exception de type **IOException**. Il suffit donc d'utiliser un bloc **try { } catch (IOException e) { }** pour gérer le problème.

Il est important d'utiliser ces mécanismes de contrôle lorsque des tratitements coûteux en temps processeur peuvent être lancés par votre servlet. Un test avant le début d'un tel traitement permet d'éviter un gaspillage dans le cas où l'utilisateur aurait pressé le bouton "stop".

Persistance de la connexion

Une autre particularité inhérente à HTTP et qu'il est intéressant de souligner est l'absence de persistance de connexion par défaut. C'est à dire qu'une ressource regroupant plusieurs types de données, par exemple une page web contenant du texte et trois images, nécessitera généralement plusieurs connexions : une pour chaque ressource contenue dans le document accédé (donc quatre connexions dans l'exemple utilisé).

La raison pour laquelle ceci se passe est qu'il est impossible pour le client web (le navigateur), en utilisant ce que nous avons vu du protocole HTTP, de connaître

les différentes étapes de l'envoi des données sans utiliser une connexion pour chaque ressource. Pour utiliser la même connexion pour toutes les ressources d'un document, le serveur doit spécifier explicitement où débute l'envoi des données et où il se termine (en donnant la taille des données à envoyer). Toutes les ressources peuvent donc être envoyées sur la même connexion

Si la taille donnée par le serveur pour l'ensemble des ressources est inférieure à la taille de celles-ci, le client attendra les octets imaginaires manquants. Si la taille donnée est supérieure à la taille réelle de la ressource à envoyer, l'écriture dépassera la fin du flot et une exception sera probablement générée. Les détails concernant la gestion de ces erreurs dépendent du moteur de servlets utilisé, il se peut que vous ne rencontriez pas un comportement strictement identique.

Pour spécifier la taille des données à envoyer, il faut utiliser la méthode **setContentLength(int taille)** de l'interface **ServletResponse**. Le paramètre **taille** correspond à la taille en octet de la ressource qui va être envoyée. L'utilisation de cette méthode permet au navigateur, s'il le souhaite, de gérer la récupération du document avec une connexion persistante. La seule contrainte ici est d'appeller cette méthode avant d'envoyer quoi que ce soit sur le flux de sortie. Pour résoudre ce problème, il suffit d'écrire le contenu à envoyer dans un tampon dont vous pourrez connaître la taille. Une fois cette taille connue, il suffit d'utiliser la méthode **setContentLength(int taille)** en passant la taille du tampon en paramètre de la méthode, puis d'écrire le tampon sur la sortie. Voici un exemple de servlet utilisant ce mécanisme :

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

public class ServletPersistante extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res) throws
ServletException, IOException {
    res.setContentType(“text/html”);

    // création du tampon contenant les données à envoyer
    // ce tampon s’étend au fur et à mesure que des données y
    // sont écrites
    BytearrayOutputStream tampon = new BytearrayOutputStream();
    // création du flux d’écrire de texte sans vidage automatique
    // du tampon utilisé
    PrintWriter out = new PrintWriter(tampon);

    out.println(“<HTML>”);
    out.println(“<HEAD><TITLE>Servlet persistante</TITLE></HEAD>”);
    out.println(“<BODY>”);
    out.println(“Ceci est une servlet utilisant une connexion persistante.”);
    out.println(“</BODY>”);
```

```
        out.println("</HTML>");

        // récupération de la taille du tampon
        // afin de déterminer la valeur de l'en-tête
        // Content-length
        res.setContentLength(tampon.size());
        // envoi des données
        tampon.writeTo(res.getOutputStream());
      }
    }
```

Vous connaissez désormais les mécanismes fondamentaux qui gèrent la diffusion du contenu vers un client utilisant le protocole HTTP. Nous n'avons cependant pas considéré la forme du contenu. Étant donné que des pages web peuvent vite devenir conséquentes et que leur esthétique est importante, nous allons voir deux choses différentes. Tout d'abord vous étudierez les différentes façons de produire du code HTML, et ensuite les possibilités d'intégration de contenu multimédia (images, sons, animations).

La production de pages web écrites en HTML

Il existe plusieurs manières d'envoyer du code HTML vers un client utilisant le protocole HTTP. Nous en verrons trois d'entre elles, qui me semblent couvrir les bases reprises par d'éventuelles méthodes différentes. Comme vous le verrez, le choix de ces méthodes dépend essentiellement de la quantité de contenu à fournir, de la structure de votre document et du service fourni par celui-ci. En effet, si tous les documents d'un site possèdent la même forme, il sera intéressant de "factoriser" le code HTML en utilisant des modèles.Par contre si le site web contient un code HTML très simple il sera plus simple et rapide d'utiliser des fonctionalités basiques.

Nous avons dejà vu, tout au long du document, la création d'un contenu HTML classique. Il suffit pour cela d'utiliser un flux de type **PrintWriter** et de spécifier l'en-tête "Content-type" adéquate comme ceci :

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

Les données au format HTML sont alors passées à la méthode **println** de classe **PrintWriter** comme des chaînes de caractères classiques de la manière suivante :

```
out.println("<B>Voici une phrase en police grasse.<B>");
```

Il n'apparaît aucune limitation particulière sur cet exemple précis, mais il faut songer à l'utilisation de cette méthode pour produire un contenu complexe. Cela devient très rapidement pénible pour plusieurs raisons :

– le code source de la servlet devient très vite énorme.

– la majeure partie de ce contenu est statique (pas généré par un algorithme).
– les erreurs sont fréquentes (oubli de fermeture d'une balise par exemple).
– on n'utilise pas la programmation orientée objet.

Pour remédier à cela, nous allons étudier deux alternatives : la génération HTML orientée objet et le système de templates (ou modèles). Ces deux techniques ne font pas partie de l'API des servlets et ne sont donc pas fournies par l'ensemble de classes que vous avez téléchargé lors de l'installation de Tomcat (voir 2.2.3 page 22). Outre le fait que vous devrez les télécharger, il vous sera sûrement nécessaire de vous re-habituer à un mode de fonctionnement particulier à cet ensemble de classes et à sa documentation. Étant donné que je ne ferais qu'aborder le sujet il vous sera probablement utile de fournir quelques efforts supplémentaires pour vous familiariser avec ces techniques, en lisant la documentation par exemple.

La première technique que nous alons étudier est la génération HTML orientée objet. Les classes que j'utilise dans les exemples qui suivent sont téléchargeables à l'URL `http://www.weblogic.com/`.Une fois ces classes téléchargées, il suffit d'ajouter leur chemin dans la variable environnement CLASSPATH comme ceci :

export CLASSPATH=$CLASSPATH :/chemin/vers/les/classes/leJar.jar

Cette instruction permet de pouvoir compiler et lancer les exemples. Appuyons nous sur un exemple pour démarrer :

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

public class BonjourMondeObjet extends HttpServlet {
  public void doGet(HttpServletRequest req, HttpServletResponse res) throws
                    ServletException, IOException {
    res.setContentType("text/html");
    ServletPage page = new ServletPage();
    page.getHead().addElement(new TitleElement("Bonjour, monde !"));
    page.getBody().addElement(new BigElement("Bonjour monde !"));

    page.output(res.getOuputStream());
  }
}
```

Compilez et effectuez une requête sur cette servlet et un message célèbre apparaîtra. Vous pouvez déjà vous apercevoir que plus aucune balise HTML n'est utilisée. Cela évite déjà les nombreuses erreurs d'écritures qui peuvent être rencontrées lorsque l'on manipule directement ces balises. Les seules erreurs qu'il est possible de commettre au niveau de la structure HTML du document seront maintenant signalées à la compilation (nom de méthode inexistant par exemple), ce qui évite des séances fastidieuses de recherche d'erreur. Ensuite, on voit clairement que nous utilisons la programmation orientée objet. En effet, un objet de type **ServletPage** est créé et la production du code HTML se fait, en l'occurrence, par les deux instructions suivantes :

```
page.getHead().addElement(new TitleElement("Bonjour, monde !"));
    page.getBody().addElement(new BigElement("Bonjour monde !"));
```

La méthode **getHead()** de la classe **ServletPage** retourne une référence sur un objet décrivant l'en-tête de la page HTML. Il est possible d'appeler d'autres méthodes sur cet objet, comme **addElement**. Cette méthode, comme vous pouviez vous en douter, ajoute un élément à l'en-tête. Ici c'est un élément décrivant le titre qui est ajouté, mais on aurait pu ajouter d'autres éléments appartenant à d'autres types. L'utilisation d'une génération orientée objet permet de changer les propriétés de tout le code HTML généré par toutes vos pages en changeant seulement des informations au niveau de l'objet utilisé. Ainsi, il suffit de modifier la classe **TitleElement** pour en modifier la mise en forme. C'est un peu le même principe que les feuilles de styles[3] qui permettent de séparer le contenu de la forme. Les servlets produisant beaucoup de code HTML et utilisant cette méthode sont donc plus lisibles et faciles à maintenir. Il est également possible de rajouter ses propres types en créant des sous classes ou de nouvelles classes. Rien ne vous empêche de créer vos propres classes de base utilisées par chacun de vos sites, puis pour chacun d'eux, ou créer des sous classes spécialisées. Je vous laisse découvrir par vous même cette méthode dont les modalités d'utilisation peuvent être différente selon les besoins. Cela dit vous ne devriez éprouver aucune difficulté, en sachant que vous devez bien connaître la programmation orientée objet. Vous la connaissez n'est-ce pas ?

Malgré l'utilisation de cette méthode intéressante, comment peut on éviter de voir proliférer le contenu statique au sein d'une servlet, et comment permettre aux graphistes de votre site web de travailler en toute indépendance des programmeurs ? Une solution envisageable est l'utilisation de templates. Le mot "template" signifie "modèle". En effet, de nombreux sites web utilisent un contenu dynamique qui suit des règles précises. En utilisant un système de modèles, il suffit simplement de préciser quelle partie du code HTML est dynamique, et de générer le contenu correspondant à ces petites parties par vos servlets. Tout le reste de la page est statique et extrait par le moteur de modèles à partir du fichier HTML. Un moteur de modèles est un ensemble de classes qui prend en charge la liaison entre les marqueurs définis dans les pages HTML pour distinguer le contenu dynamique du contenu statique et votre servlet. Puisque l'on parle de moteur de modèles, il faut savoir qu'il en existe plusieurs. Chaque moteur propose ses propres possibilités et sa manière de les implémenter. C'est à vous de choisir celui qui vous correspond le mieux. Les principaux moteurs de modèles sont :

- Enhydra : disponible à `http://www.enhydra.org/`.
- Webmacro : disponible à `http://www.webmacro.org/`.
- Freemarker : disponible à `http://freemarker.sourceforge.net/`.

tous ces moteurs sont des logiciels libres, vous pouvez donc vous les procurer gratuitement, bénéficier de nombreuses améliorations rapidement et participer au développement. Voici ci-dessous un exemple simple d'une servlet utilisant un modèle pour produire le contenu. Cet exemple est programmé avec le moteur de modèles Enhydra :

[3] Pour plus d'informations sur les feuilles de styles, consultez le site web du Wolrd Wibe Web Consortium (w3c) à l'URL `http://www.w3c.org/`.

```
<html>
<body>
<span id="MonTitre">Categorie Vide</span>
<p>
<span i="MaLigne">   <a id="MonUrl" href="#">Contenu Categorie</
</p>
</body>
</html>
```

La servlet permettant d'utiliser le modèle ci-dessus :

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
public class MaServlet extends HttpServlet {
public void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
  // instancier la page
  MaPageHTML p = new MaPageHTML();
  // remplir le titre
  // l'objet "cat" est une instance d'un type personalisé qui et défini
  // par une classe utilisateur quelconque (ici cette classe
  // est une sous-classe d'ArrayList).
  p.setTextMonTitre(cat.isEmpty()?"" :cat.getName());
  // retrouver les elements
  Element ligne = p.getElementMaLigne();
  Element paragraph = ligne.getParentNode();
  Element url = p.getElementMonUrl();
  // retirer la ligne template et ajouter les lignes reelles
  paragraph.removeChild(ligne);
  for (int i=0; i < cat.size(); i++) {
    // changer le lien
    url.setHRef(encodeURL("/servlet/JTus?p=1&ca="+cat.refCateg));
    // changer le texte
    url.removeChild(url.getFirstChild());
    url.appendChild(p.createTextNode(cat.Nom));
    // dupliquer et ajouter la ligne
    Node clonedNode = ligne.cloneNode(true);
    paragraph.appendChild(ligne);
  }
  // retourner la page au client
  response.setContentType("text/html");
  java.io.PrintWriter writer = response.getWriter();
  writer.print(p.toDocument());
  writer.flush();
}
}
```

On peut voir également que pour une gestion des modèles de base, un moteur de modèles peut être trop lourd à utiliser.

Pensez simplement que vous pouvez très bien programmer vos classes représentant par exemple un pied de page standard. Cette classe contiendrait une méthode **ecrireContenu(PrintWriter sortie)** qui écrirait son contenu sur le flux de sortie de la servlet. Vous pourriez alors créer autant de pied de pages spécifiques en dérivant cette classe de base, tout en conservant la même forme pour tous les pieds de page du mêm"e type utilisés dans vos pages web. En utilisant des conventions de nommage (pour vos méthodes et attributs) simples et en les respectant scrupuleusement, vous pourriez arriver à disposer d'un système de modèles simple, utile et moins lourd que les ténors du genre.

Maintenant que nous savons générer du contenu HTML, voyons ce qui est mis à notre disposition pour générer un contenu multimédia.

La génération de contenu multimédia

Les deux types de contenu multimédia qu'il est possible de générer avec les servlets sont les images et les animations. Les images sont parfois très utiles car elles permettent de représenter une information de manière synthétique en très peu de place. Dans certains cas (représentation de l'évolution du cours d'une action par exemple), la même information écrite avec du texte prendrait beaucoup de place et serait moins compréhensible. Les graphiques sont une illustration du caractère irremplaçable des images au sein d'un système d'information. Quant aux animations, elles servent parfois l'information mais ne sont pas facilement générées depuis un serveur. En effet, le protocole HTTP n'étant pas orienté connexion comme nous avons pu le voir (voir 2.1.2 page 15), il est seulement possible de produire des animations lentes, grâce à des types d'en-têtes particuliers. Il en est de même pour le son. Un fichier sonore ne peut pas être transmis en flux continu du fait du fonctionnement du protocole HTTP lui même. Il est cependant tout à fait possible de gérer des animations graphiques ou un flux sonore mais pas dans le cadre d'une servlet fournissant un service à un client HTTP comme un navigateur web. Il faut utiliser la connexion par sockets pour cela, et ce n'est pas l'objet de ce chapitre. Nous allons maintenant aborder la conception de graphique côté serveur en utilisant les servlets et le protocole HTTP.

La génération d'images

Une image est un fichier au même titre qu'un document HTML ou au format ASCII, mis à part qu'il utilise le format binaire (pas de notions utiles dans des documents textuels comme le retour à la ligne). Notre servlet d'exemple servant n'importe quel type de fichier (voir 4.1 page 53) peut donc envoyer directement le fichier au client qui, grâce aux en-têtes décrivant le format de l'image ("image/jpeg" par exemple), pourra la décoder et l'afficher sur l'écran de la machine cliente. Mais comment faire lorsque nous voulons générer dynamiquement une image et l'envoyer au visiteur de notre site web ?

Vous savez qu'il est possible de créer des images dynamiquement dans une application Java classique ou une applet. Un ensemble de classes est mis à disposition pour créer, manipuler et afficher des images. Elles font partie du package **java.awt** et disposent de toutes les fonctions utiles, dont certaines destinées à coder l'image brute créée en un format standard étudié pour l'échange de données graphiques sur internet, comme le format Jpeg. Grâce à l'exemple suivant, vous allez vous apercevoir que la génération d'une image par une servlet ne diffère pas vraiment de la méthode utilisée par une application classique :

```
import com.sun.image.codec.jpeg.*;
import javax.servlet.http.*;
import javax.servlet.*;
import java.awt.*;
import java.awt.image.*;
import java.io.*;
public class ImageServlet extends HttpServlet {
  public void doGet(HttpServletRequest req, HttpServletResponse res) throws
                    ServletException, IOException {
   // le contenu produit est une image au format jpeg
   res.setContentType("image/jpeg");
   ServletOutputStream out = res.getOutputStream();
   // l'objet enc va encoder les données et les envoyer sur
   // le flux de sortie de type ServletOutputStream
   JPEGImageEncoder enc = JPEGCodec.createJPEGEncoder(out);
  // création d'une nouvelle image dune réolution de 1024 par
  // 768
  BufferedImage image = new
  BufferedImage(1024,768,BufferedImage.TYPE_BYTE_INDEXED);
 // récupération du contexte graphique lié à l'image
  Graphics2D g = image.createGraphics();
  // la prochaine opération s'effectuera avec la couleur rouge
  g.setColor(Color.red);
  // affichage de la célèbre phrase
  g.drawString("Bonjour monde!", 400, 500);
  // transformation des données au format jpeg et envoi
  // de celles-ci sur le flux standard de sortie (le navigateur)
  enc.encode(image);
  }
}
```

Faites cependant attention à un problème relativement difficile à déceler. Une machine sous Unix, lorsqu'elle fournit des services au sein d'un réseau (comme un moteur de servlet par exemple) ne lance pas un environement graphique. Or, lorsque nous décidons de créer une image, des classes du paquetage **java.awt** sont chargées et des objets de type **Image** décrivant les images à afficher sont créés. C'est pour cela que vous rencontrerez une erreur si la machine sur laquelle

tourne le moteur de servlets ne dispose pas d'un environement graphique et n'ajuste pas sa variable d'environement DISPLAY. Vous devez également vous assurer que l'utilisateur sous lequel tourne le processus de la JVM est autorisé à utiliser le "display". Lisez la documentation sur la commande **xhost** en entrant

man xhost

pour savoir comment autoriser des utilisateurs à utiliser le display.

Chapitre 5

Le mot de la fin

Voilà, c'est dejà terminé pour cette version du document. Vous êtes bien sûr encouragé à le critiquer, à contribuer à son développement et à me faire part de toutes sortes de réactions. Pour favoriser cela et débuter un projet destiné à créer un ensemble de documents a usujet des servlets, le projet *jwebadmin* a été créé. Vous pouvez y participer en me contactant à *darktigrou@bigfoot.com* ou en vistant le site web `http://www.sourceforge.net/project/jwebadmin/`.

L'ensemble de documents sera constitué d'une documentation écrite (qui sera une version plus complète de cette documentation) et d'une application d'administration de serveur web à distance.

Les prochaines versions de ce document aborderont les thèmes suivants :

- La communication entre les applets et les servlets.
- La gestion des sessions.
- La sécurité.

Les thèmes abordés actuellement seront étendus et corrigés et plus d'exemples seront mis à disposition. Il serait aussi agréable de disposer d'un système d'annotation, pour permettre aux lecteurs de compléter les informations données par ce document. Je vous remercie d'avoir été attentif jusqu'au bout de cet apprentissage.