MASTER EN SCIENCES INFORMATIQUES

Internet et les bases de données
approche comparative de solutions technologiques

Davreux, Olivier

Award date:
1999

Mémoire de fin d'étude

Promoteurs: J-L. Hainaut & Cl. Lobet-Maris

INTERNET

ET

LES BASES DE DONNEES

APPROCHE COMPARATIVE DE SOLUTIONS TECHNOLOGIQUES

O. DAVREUX

Olivier Davreux

3ème Maîtrise Informatique - F.U.N.D.P. Namur

1998-1999

RÉSUMÉ.

L'Internet permet l'accès à l'information partout dans le monde. Cette information doit bien évidemment être mise à disposition sur des serveurs permettant ainsi à quiconque ayant un accès Internet de s'y connecter et d'en prendre connaissance.

C'est essentiellement cette mise à disposition de l'information qui va retenir notre attention tout au long de ce mémoire.

A cette fin, nous détaillerons les différentes technologies qui permettent la gestion de l'information via l'Internet, en faisant particulièrement attention aux techniques permettant la connexion de bases de données à partir des pages Web. Ces technologies seront alors évaluées à partir d'un ensemble de critères que nous auront définis auparavant.

Une fois le panorama des technologies détaillé, nous proposerons alors une méthodologie d'aide à la décision permettant à tout concepteur de sites Web d'être guidé vers une technologie particulière en fonction des exigences requises pour son futur projet. Ces exigences seront également évaluées à l'aide des critères précités.

ABSTRACT.

Internet allows the access to information from all over the world. This information must be placed at disposal on servers allowing anyone having an Internet access to connect it and to make inquiries.

In this dissertation we shall essentially focus on this information disposal.

To this end, we will detail the different technologies which allow the information management by way of Internet, taking special care at technologies permitting the databases connections from Web pages. These technologies will then be evaluated use of a criteria set that we will have defined before.

Once this technology panorama will be detailed, we will propose a decision help methodology allowing every designer of Web sites to be guided to a particular technology according to the required demands for his future project. These demands will also be evaluated via the use of the abovementioned criteria.

TABLE DES MATIÈRES.

Partie I. Construction d'une grille d'évaluation.

Partie II. Analyse des technologies.

TABLEAUX, FIGURES ET EXEMPLES.

Tableaux.

Figures.

Exemples.

PARTIE I. CONSTRUCTION D'UNE GRILLE D'ÉVALUATION.

Chapitre 1. INTRODUCTION GÉNÉRALE AU MÉMOIRE.

Le présent chapitre est destiné à l'analyse de la situation qui sera abordée dans ce mémoire. A cette fin, nous détaillerons les concepts sous-tendus par les termes INTERNET et BASES DE DONNEES qui constituent le titre du document. Nous les préciserons d'abord séparément pour ensuite en détailler les liens qui peuvent les unir.

1.1. L'Internet.

L'Internet permet la circulation d'informations à travers le monde. Ainsi, quiconque possédant un ordinateur et une connexion Internet peut accéder à des serveurs d'informations (appelés "serveurs Web") et consulter des documents à distance. Dans ce mémoire, nous allons nous intéresser aux possibilités de mise à disposition des informations via le réseau mondial, c'est-à-dire au niveau de la conception des sites Internet. Notons au niveau du vocabulaire que les termes Internet, Net, Web,, World Wide Web (*WWW*) sont des synonymes et seront donc employés de manière aléatoire dans ce document.

A propos de la conception des sites Web, un grand nombre de méthodes sont offertes, de la simple page *HTML* statique à la création de cette même page au moment de la connexion, en tenant compte des informations actuelles sur le sujet demandé. On peut penser dans ce dernier cas à la compagnie aérienne qui désire mettre à disposition de ses agences le nombre de places restant sur chacun de ses vols: afin d'avoir l'information la plus exacte au moment de l'appel, il est nécessaire de consulter la base de données de la société avant l'envoi de la page. Ces différentes possibilités seront donc examinées tout au long de notre analyse.

1.2. Les bases de données.

Tout document de texte peut être mis à disposition sur le Web. Toutefois, la structuration des données permet généralement d'en faciliter le contrôle. Ainsi, on peut parler des bases de données et de leur mise à disposition via le Web. En effet, les bases de données offrent un certains nombre d'avantages indéniables:

- quant à la mise à jour des informations;
- quant à leur structuration;
- quant aux outils permettant de gérer ces informations;
- quant à la vitesse d'accès à l'information.

On comprend bien l'intérêt d'une base de données quand on utilise des logiciels de réservation de billet de train, quand on recherche des numéros de téléphone ou un livre, etc.

On distinguera dans ce document deux types de bases de données (*BD*): les "relationnelles" et les "semi-structurées". Il existe actuellement d'autres types de *BD* (*BD* "orienté objets", *BD* "hiérarchiques", ...) mais elles n'interviennent pas dans les solutions technologiques qui seront proposées par la suite. Nous ne nous y attarderons donc pas ici.

1.2.1. BD "relationnelles".

Comme J-L. Hainaut le souligne dans "*Bases de données et modèles de calcul - Outils et méthodes pour l'utilisateur*" ([2]), "Une base de données relationnelle apparaît comme une collection de tables de données, ou fichiers *plats*. Il s'agit d'une structure extrêmement simple et intuitive qui, pour l'utilisateur du moins, ne s'encombre d'aucun détail technique concernant les mécanismes de stockage sur disque et d'accès aux données.".

De manière conceptuelle, on peut donc représenter une *BD* "relationnelle" comme un ensemble de tables dans lesquelles chaque ligne peut être mise en relation avec des lignes d'autres tables. Ainsi, par exemple, une *BD* reprenant des informations sur des films pourrait être illustrée comme suit:

Table **FILM**:

Titre	Realisateur	Producteur	Annee
La grande vadrouille	G. Oury	L. Besson	1985
La courbure de l'espace-temps	P.-Y. Schobbens	J. Fichefet	2038
Le coup de la pince à cheveux	H. Lotti	C. Goya	1938
Y a-t-il un pilote dans la Droli-Mobile ?	H. Droli	H. Droli	1999
Catch à trois	S. Mihy	R. Bournonville	1998
...	...	...	...

Table **ACTEUR**:

Nom	Prenom	Date de Naiss.
Frimout	Dirk	21/08/1969
Droli	Helmut	24/01/1975
de Funès	Louis	20/05/1928
Richard	Pierre	08/09/1952
Bourvil		29/12/1932
...	...	...

Exemple 1: Tables d'une base de données "relationnelle".

On remarque de suite qu'il serait intéressant de pouvoir lier ces deux tables de manière à savoir quels acteurs ont joué dans tel ou tel film. Par exemple, il faudrait un lien entre le *film* dont le *titre* est "La grande vadrouille" et les *acteurs* dont le *nom* est "de Funès" et "Bourvil". Dans les *BD* "relationnelles", de tels liens existent grâce à des "clés" et à des "index". Ces technologies permettent donc de retrouver efficacement les lignes d'une table qui correspondent à une ligne particulière d'une autre table. On peut alors retrouver toutes les informations liées entre elles.

La gestion de ces *BD* se fait à l'aide de "Système de Gestion de Base de Données" où *SGBD*. Il en existe un grand nombre sur le marché et le plus connu d'entre eux est "Oracle"

qui tourne sur un grand nombre de plates-formes dont "Unix" et "Windows NT", ensuite vient "Sybase" également présent sur les deux plates-formes. On commence à parler de "SQL Server" de "Microsoft", qui est un produit dérivé de "Sybase" et qui n'existe pas sur plate-forme "Unix". On peut encore citer "Informix", "DB2", "Ingres" (sur "Linux"), "MySQL" (sur "Linux" également) ou "Illustra".

"Toutes les manipulations s'effectuent au moyen d'un unique langage, *SQL* (*Structured Query Language*). Ce langage permet à l'utilisateur de demander au *SGBD* de créer des tables, de leur ajouter des colonnes, d'y ranger des données et de les modifier, de consulter les données, de définir les autorisations d'accès. Les instructions de consultation des données sont essentiellement de nature prédicative. On y décrit les données qu'on recherche, notamment en spécifiant une condition de sélection, mais on ne spécifie pas le moyen de les obtenir, décision qui est laissée à l'initiative du *SGBD*." ([2]).

Grâce à ce langage simple qu'est *SQL*, il est aisé d'exprimer des requêtes simples. Par exemple, rechercher les *Titres* des *Films*, et leur *producteur* respectif, qui ont été produit après 1990 s'écrit simplement comme suit:

```
SELECT Titre, Producteur
FROM FILM
WHERE Annee >= 1990
```

Exemple 2: Requête *SQL*

Bien entendu, l'extraction de données provenant de plusieurs tables et satisfaisant des conditions complexes nécessitera des requêtes plus élaborées demandant plus d'expérience en *SQL* de la part de l'utilisateur.

1.2.2. BD "semi-structurées".

On parlera de *BD* "semi-structurées" pour des fichiers contenant des informations sans liens technologiques entre eux. Les données ont une certaine structure mais il n'existe pas d'artifice qui relie les champs entre eux. Si on reprend l'exemple ci-dessus, à chaque fichier correspond une table mais pour retrouver des informations liées, il faut parcourir toute la table à la recherche de chaînes de caractères: aucun support n'est fournit pour les recherches.

L'exemple suivant reprend la table *FILM* décrite ci-dessus en mode "semi-structuré" avec un choix de délimitateurs particuliers: **{}** pour délimiter les lignes et le **;** pour séparer les champs. La première ligne permet de donner un nom à chaque champ.

```
{TITRE;REALISATEUR;PRODUCTEUR;ANNEE}
{La grande vadrouille;G. Oury;L. Besson;1985}
{La courbure de l'espace-temps;P.-Y. Schobbens;J. Fichefet;2038}
{Le coup de la pince à cheveux;H. Lotti;C. Goya;1938}
{Y a-t-il un pilote dans la Droli-Mobile ?;H. Droli;H. Droli;1999}
{Catch à trois;S. Mihy;R. Bournonville;1998}
...
```

Exemple 3: Base de données "semi-structurée".

La création de tels fichiers ne posera aucun problème aux plus novices en informatique. Toutefois, ce genre de base de données présente certaines limitations. Premièrement, les erreurs de frappe, et les informations en général, seront plus difficilement visibles. Ensuite, les recherches seront plus compliquées puisqu'aucun outil n'est fourni directement. Des fonctions et procédures personnelles devront donc être écrites pour pouvoir effectuer des

requêtes sur les données (on perd donc l'avantage du *SQL* en *BD* "relationnelles" qui permet d'utiliser un langage commun quel que soit la base de données sous-jacente). Cela signifie donc un découpage du fichier par une recherche des caractères spéciaux utilisés comme délimitateurs (ici {} et ;). De plus, si l'on désire effectuer des requêtes "croisées" entre plusieurs fichiers, de nouvelles procédures devront être créées. On notera finalement que, de manière générale, toute recherche sur une *BD* "semi-structurée" sera donc moins performante que son équivalente en *BD* "relationnelle", que se soit au niveau de la lecture/écriture ou de la recherche. Ceci est principalement dû au fait que les technologies utilisées par les *SGBD* sont plus adaptées qu'une recherche par caractère dans des fichiers séparés.

1.3. Caractères statique et dynamique des pages.

Comme on le verra par la suite, la base principale de l'Internet, au niveau du contenu des documents, est le fichier *HTML*. La plupart du temps, ces fichiers sont écrits une fois pour toute et mis à jour tous les x mois, lorsque le concepteur en a le temps. De telles pages restent acceptables tant que l'information qu'elles font circuler ne change pas trop souvent: on parlera alors de pages *statiques*. A partir du moment où ce caractère statique ne convient plus, on peut imaginer bien des solutions de mise à jour. Nous pouvons citer:

- La mise à jour manuelle périodique: une personne est chargée de mettre les dernières informations dans les pages *HTML*, tous les jours, toutes les semaines voire tous les mois;
- La mise à jour automatique périodique: une application est chargée de consulter la base de données pour changer les pages qui ne sont plus à jour;
- La mise à jour automatique par le *SGBD*: lors de la modification de la *BD*, une procédure pourrait s'exécuter, ce qui aurait comme conséquence une mise à jour des fichiers *HTML* (c'est ce qu'on appelle les *triggers*);
- La création de la page au moment de l'appel: lors de la connexion, la page est créée en fonction des informations contenues dans la base à ce moment précis ainsi qu'éventuellement des informations fournies par l'utilisateur;
- La consultation continue: on peut penser ici à une application de la Bourse où les informations seraient automatiquement mises à jour toutes les 10 secondes par exemple.

Du caractère *statique* de l'information, on passe à un caractère de plus en plus *dynamique* en fonction du but recherché.

1.4. But du mémoire et démarche.

Comme nous venons de le montrer, il existe une multitude de façon d'utiliser le Web. Ce que nous allons voir par la suite, c'est qu'il existe également une multitude de solutions permettant de créer des sites Web. En effet, le concepteur d'un site Internet se trouve confronté à un ensemble de technologiques applicables.

Afin de l'aider dans sa mission, ce mémoire va dans un premier temps lui permettre de situer son projet par rapport à un ensemble de critères définis dans le deuxième chapitre de ce document (Chapitre 2: Critères de conception.). La deuxième partie (Partie II: Analyse des technologies.) décrira alors les architectures offertes pour la mise à disposition d'information via le Net, avec pour chacune d'entre elles une évaluation face aux critères définis précédemment. Finalement, après avoir fait une synthèse de toutes les technologies, nous proposerons une aide à la décision vers un choix technologique en fonction des combinaisons de valeurs pour chaque critères (Partie III: Synthèse et choix technologiques.).

Le concepteur d'un site désirant être guidé vers des choix technologiques pourra donc suivre la procédure suivante:

- définir les valeurs des critères pour son propre projet (Chapitre 2);
- découvrir l'ensemble des technologies qui sont offertes par l'Internet pour se faire une idée globale de la situation (Partie II);
- suivre l'aide à la décision qui lui est proposée pour découvrir les choix technologiques appropriés à son propre projet. En fonction des choix qui lui seront proposés, les chapitres correspondants dans la Partie II pourront être relus plus en profondeur.

1.5. Public visé.

Ce mémoire s'adresse aussi bien aux concepteurs de sites professionnels qu'aux novices n'ayant que très peu de connaissances informatiques. Pour les premiers, ce document représente une synthèse des technologies mises à disposition par le marché. Cette synthèse pourra ainsi apporter des informations sur des technologies qui leur seraient encore inconnues ou leur donner une évaluation des techniques qu'ils utilisent par rapport aux autres techniques possibles. Pour les novices en la matière, ce mémoire constitue une méthode de travail vers un choix technologique pour un cas particulier: ainsi, tout administrateur d'une société désirant créer son site Web se verra guider vers un choix technologique qu'il pourra conseiller à ses concepteurs. Il est à noter qu'une connaissance minime dans le domaine informatique est tout de même requise.

1.6. Bibliographie.

[1] George Reese, "*Database Programming with JDBC and JAVA*", O'Reilly & Associates, United States of America, June 1997.

[2] Jean-Luc Hainaut, "*Bases de données et modèles de calcul - Outils et méthodes pour l'utilisateur*", InterEditions, Paris, 1994.

Chapitre 2. CRITÈRES DE CONCEPTION.

2.1. Introduction.

Avant de détailler les technologies liées au Web et aux bases de données, il paraît important de pouvoir "classer" le type de site que l'on désire concevoir par rapport à un ensemble de critères. Le but de cette partie sera donc de proposer un certain nombre de critères que le concepteur pourra évaluer par rapport à son futur projet. Pour chaque critère, des valeurs seront proposées. Le concepteur pourra donc choisir la valeur qui correspond le mieux à son projet par rapport au critère correspondant et ainsi définir une liste de valeurs.

Les technologies qui seront décrites dans la suite du mémoire (Partie II) reprendront une première évaluation des critères en fonction des possibilités offertes par la technique concernée. Ensuite, une synthèse sera faites et des choix technologiques seront proposés toujours en fonction de ces critères (Partie III).

Signalons encore que l'ensemble des critères décrit ici relève d'une réflexion personnelle de l'auteur sur les points qui lui paraissent importants lors de la conception de sites Internet et constitue en ce sens une partie originale du mémoire.

2.2. Liste des critères.

Afin de proposer des choix technologiques pour un projet particulier, le concepteur peut prendre note des valeurs qui correspondent le mieux à chaque critère pour son futur site. Il constituera ainsi une liste pouvant être utilisée par la suite. Il lui suffit pour cela de noter la lettre entre parenthèses devant la valeur correspondante.

2.2.1. Critères informationnels.

Cette première série de critères concerne les informations à mettre à disposition via l'Internet.

2.2.1.1. Structure des données.

La *structure des données* exprime le fait que les informations peuvent être ou non décomposées suivant une structure déterminée. La structure va de simples fichiers multimédia à la décomposition de l'information en champs et sous-champs simples (hiérarchie récursive) comme c'est le cas en bases de données (voir 1.2).

Echelle de valeurs:

(I) Inexistante → aucune structure: les informations sont de simples documents multimédia (textes, images, vidéo, etc.);

(H) Hiérarchique → documents décomposés hiérarchiquement en titres, sous-titres, paragraphes, etc.;

(E) Elevée → décomposition des informations suivant une structure de champs simples (bases de données "relationnelles" E(rel) ou "semi-structurées" E(ss)).

2.2.1.2. Volume.

Le *volume* représente la quantité d'informations mise à disposition à un moment donné.

Le volume peut aller de quelques fichiers de quelques Ko à une base de données de plusieurs Gb.

Echelle de valeurs:

(P) Petit → quelques dizaines de fichiers de quelques Ko;

(G) Gros → plus d'un Mégabyte, voire des Gigabytes.

2.2.1.3. Fréquence des changements.

La *fréquence des changements* détermine la période pendant laquelle les informations restent "acceptables". C'est le nombre de changements à effectuer par unité de temps.

Les valeurs pour un tel critère vont de la seconde (cas de la Bourse) au mois voire à l'année. Plus une information aura une fréquence de changement élevée et plus elle sera *dynamique*, tandis qu'une information de fréquence de changement petite sera dites *statique*.

Echelle de valeurs:

(F) Faible → changements tous les mois, tous les ans;

(M) Moyenne → changements toutes les semaines;

(E) Elevée → changements toutes les heures, toutes les minutes, toutes les secondes.

2.2.1.4. Sécurité.

La *sécurité* d'un système définit sa capacité de protection des informations qu'il met en circulation. Plus les informations seront confidentielles et plus le système devra être sécurisé.

On peut valorisé ce critère en déterminant le type de protection recherché :

- Protection en écriture ;
- Protection en lecture ;
- Protection en mise à jour ;
- Protection combinée: accès différents en fonction de la personne connectée.

Echelle de valeurs:

(L) Accès libre → pas de limitation d'accès ou accès limités à certaines personnes mais identiques pour toutes celles-ci;

(E) Elevée → protection combinée: en fonction de la personne connectée, accès en écriture et/ou en lecture et/ou en mise à jour.

2.2.2. Architecture.

Lorsqu'on pense à la création de sites Internet, la première idée qui apparaît à l'esprit est celle d'une architecture Client/Serveur illustrée par la figure suivante:

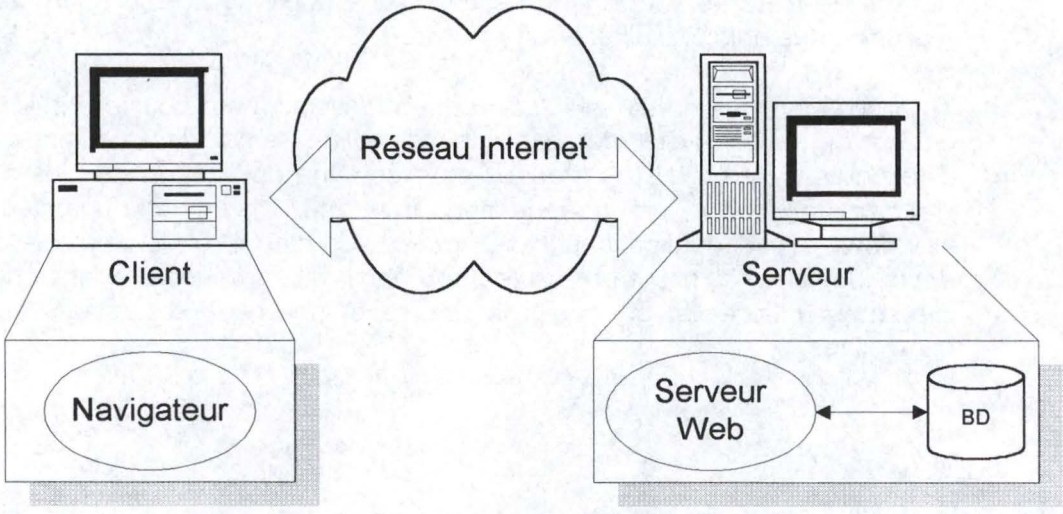

Figure 1: Architecture Client/Serveur.

La machine cliente se connecte au serveur via son navigateur. Sur le serveur, le serveur Web reçoit la requête, l'analyse, accède si nécessaire à une ou plusieurs bases de données locales et renvoie une réponse au client.

Toutefois, d'autres architectures sont possibles et sont utilisées via le Web:

- Sur le client, on peut trouver d'autres programmes en plus du navigateur. Par exemple, on peut lui associer une application qui se connecterait directement à une base de données sur un système distant. Plus simplement, le processeur client pourrait être utilisé pour des vérifications ou des calculs quelconques. Ce type d'architecture sera donc à envisager quand on désirera diminuer la durée

d'utilisation du processeur Serveur pour chaque requête. On favorisera ainsi le nombre d'accès possible au serveur par unité de temps;

- A l'opposé, le serveur peut quant à lui être aussi plus complexe. En effet, le serveur peut par exemple se connecter à des bases de données externes, tournants sur d'autres systèmes. La connexion se fait alors à l'aide d'un protocole spécifique ou par le Net.

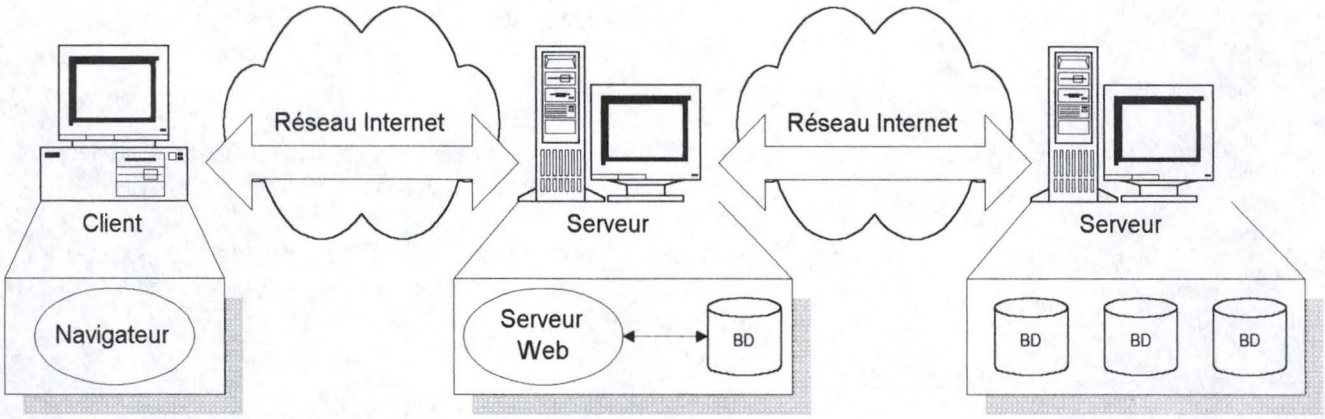

Figure 2: Architecture à bases de données distribuées.

Tout en restant central, le serveur va chercher ses informations sur d'autres systèmes distants. Cette architecture sera donc utilisée lorsque les données seront à des endroits différents;

- Toujours à propos du serveur, celui-ci peut devenir à son tour client. En effet, si le but est de pouvoir relier des applications tournants sur des machines (serveurs) différentes, il peut être avantageux de créer une application de plus haut niveau ("MiddleWare") sur un serveur spécifique fournissant une interface d'accès standard à ces applications hétérogènes. Ce genre d'architecture conviendra donc idéalement lorsqu'un certain nombre d'applications existeront sur des systèmes différents et que l'on désirera pouvoir les utiliser de manière centralisée.

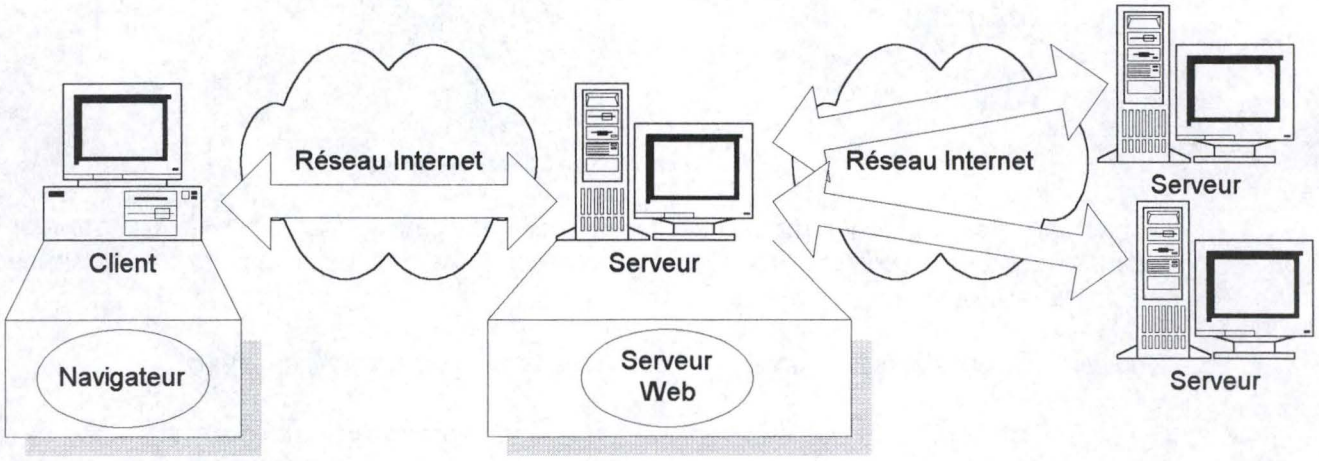

Figure 3: Architecture avec "MiddleWare".

Les différentes architectures présentées ci-dessus ne sont pas exhaustives mais elles permettent de mieux imaginer les possibilités offertes par l'Internet. Actuellement, beaucoup d'efforts sont fait pour essayer de développer de nouvelles architectures permettant la mise à disposition d'informations hétérogènes (c'est-à-dire provenant de systèmes différents et de natures différentes). On peut par exemple imaginer qu'il n'existe plus un seul serveur unique mais plusieurs serveurs, chacun pouvant fournir une partie des services offerts à l'utilisateur final. Si la gestion de telles architectures est certainement plus complexe que la simple architecture client/serveur, nous pouvons considérer dans le cadre de ce document qu'elles représentent des cas particuliers des architectures décrites ici.

Le choix d'une architecture spécifique dépend de plusieurs paramètres. C'est ce que nous allons montrer au travers des critères qui suivent.

2.2.2.1. Système client.

Ici, on s'intéresse à la partie cliente de l'application.

2.2.2.1.1. Configuration cliente.

La *configuration cliente* est l'installation de logiciels sur la machine "cliente".

Une configuration *légère* sera uniquement constituée d'un navigateur (browser) tandis qu'une configuration plus *lourde* nécessitera l'installation d'applications externes sur la station cliente. On peut également penser à d'autres choses comme des "plug-ins" venant s'intégrer au navigateur utilisé (les "plug-ins" sont des petits programmes annexes dédiés à l'interprétation d'objets non supportés d'origine par le navigateur - voir 3.2.1).

Echelle de valeurs:

(L) Légère	→	simple navigateur (browser);
(M) Moyenne	→	installation de plug-ins (petits programmes se greffant au navigateur) ou de pilotes (drivers);
(Lo) Lourde	→	installation complète ou partielle d'une application assez lourde.

2.2.2.1.2. Interface graphique.

L'*interface graphique* exprime la complexité graphique à mettre en place pour l'application désirée.

Au plus une interface nécessitera des objets graphiques, au plus elle sera caractérisée de *complexe*.

Echelle de valeurs:

(S) Simple	→	textes et images;

(M) Moyenne → quelques champs de texte éditable et des boutons;

(C) Complexe → interface genre application "Windows" complète: listes de sélection, menus déroulants, etc.

2.2.2.1.3. Interactivité.

L'*interactivité* concerne le taux de participation de l'utilisateur dans le processus de recherche de l'information. Si sa seule possibilité est l'activation d'hyperliens, on dira que l'interactivité est faible. A l'opposé, il se peut que le site propose différents formulaires, lesquels mènent vers de nouvelles pages en fonction des informations et ensuite fournissent un mode de paiement pour des articles sélectionnés. On le voit dans ce cas de commerce électronique, l'interactivité est plus complexe.

Au niveau de l'interactivité, on peut également citer la gestion de la "trace" du surfeur, c'est-à-dire l'historique des liens activés: un système qui peut générer des documents en fonction du parcours antérieur au travers du site sera plus interactif qu'un site qui propose toujours les mêmes pages quel que soit l'historique.

Echelle de valeurs:

(F) Faible → pas d'interactivité ou seulement de l'hypertexte;

(M) Moyenne → formulaires prédéfinis sans interactions directes (commande de documents par exemple). les informations sont donc envoyées vers le serveur pour être stockées mais aucun traitement spécifique n'y est associer. L'utilisateur peut par exemple envoyer ses coordonnées;

(E) Elevée → envoi de requêtes spécifiques, consultation de bases de données, commande on-line, paiement direct, etc.

2.2.2.2. Système serveur.

Sur le serveur devra se trouver un serveur Web pour répondre aux requêtes provenant d'un navigateur. Mais, suivant l'architecture désirée, d'autres outils devront être présents. C'est ce qui nous décriront via les critères suivants:

2.2.2.2.1. Configuration serveur(s).

La gestion des documents Internet nécessite un serveur Web capable de gérer les requêtes provenant des navigateurs. Pour les fichiers *HTML*, ce serveur Web suffit mais il se peut que d'autres technologies doivent lui être associées. De plus, tous les serveurs Web n'ont pas les mêmes particularités. La configuration serveur(s) exprime donc les caractéristiques des logiciels devant se trouver sur le (ou les) serveur(s).

Echelle de valeurs:

(L) Légère → serveur Web;

(M) Moyenne → installation de petits logiciels annexes au serveur Web;

(Lo) Lourde → installation complète ou partielle d'une application assez lourde.

2.2.2.2.2. Distribution des applications.

Comme on l'a montré plus haut, il existe plusieurs types d'architecture possibles. Notamment, les applications de gestion des informations peuvent prendre place à divers endroits, c'est ce qu'on appellera la *distribution des applications*. Le cas le plus simple consiste à associer toute application avec le serveur Web, sur une même machine. A l'opposé, on peut très bien imaginer des applications sur des systèmes distants et une application centrale s'occupant de connecter le serveur Web avec ces autres machines (voir Figure 3).

Echelle de valeurs:

(C) Client → applications sur le client (100% "Client-Side");

(S) Serveur → applications sur le serveur (100% "Server-Side").

(CS) Client/Serveur → applications partagées entre le client et le serveur;

(M) Multiple serveurs → applications distribuées sur plusieurs machines différentes.

2.2.2.2.3. Distribution des BD.

De même que pour les applications, les *BD* peuvent se situer à différents endroits en fonction de l'architecture choisie. L'architecture Client/Serveur (Figure 1) reprend les bases de données avec le serveur Web et les applications correspondantes sur une même machine. Par contre, si les *BD* sont situées sur des systèmes distants, il faudra fournir au serveur Web le moyen d'entrer en communication avec ces informations. On parlera alors de *distribution des BD*. C'est le cas de l'architecture décrite par la Figure 2: les applications tournent en annexe au serveur Web et se connectent directement à des bases de données distantes.

Echelle de valeurs:

(I) Inexistante → les *BD* se trouvent sur le même système que le serveur Web;

(E) Existante → *BD* sur systèmes distants;

2.2.3. Critère de conception.

Ce critère est relatif à la conception même du site .

2.2.3.1. Compétences locales.

Les *compétences locales* représentent les connaissances des concepteurs, qu'il s'agisse de connaissances d'utilisation de logiciels, de programmation "Orienté Objets" (*O.O.*), *etc.*

Echelle de valeurs:

(F) Faible → pas de connaissances en programmation;

(M) Moyenne → connaissances de langages simples comme le *HTML;*

(E) Elevée → programmeurs;

2.3. Tableau récapitulatif.

Critères	Valeurs
Critères informationnels.	
Structure des données.	(I) Inexistante - (H) Hiérarchique - (E) Elevée { E(rel) pour "relationnelles" ou E(ss) pour "semi-structurées" }
Volume.	(P) Petit - (G) Gros
Fréquence des changements.	(F) Faible - (M) Moyenne - (E) Elevée
Sécurité.	(L) Accès libre - (E) Elevée
Système client.	
Configuration cliente.	(L) Légère - (M) Moyenne - (Lo) Lourde
Interface graphique.	(S) Simple - (M) Moyenne - (C) Complexe
Interactivité.	(F) Faible - (M) Moyenne - (E) Elevée
Système serveur.	
Configuration serveur(s).	(L) Légère - (M) Moyenne - (Lo) Lourde
Distribution des applications.	(C) Client - (S) Serveur - (CS) Client/Serveur - (M) Multiple serveurs
Distribution des BD.	(I) Inexistante - (E) Existante
Critère de conception.	
Compétences locales.	(F) Faible - (M) Moyenne - (E) Elevée

Tableau 1: Valeurs pour chaque critère.

PARTIE II. ANALYSE DES TECHNOLOGIES.

Chapitre 3. TECHNOLOGIES GÉNÉRALES.

Le présent chapitre est destiné à l'explication générale des technologies utilisées pour le Web. Il est donc dédié aux utilisateurs ayant peu de connaissances globales en matière d'Internet.

3.1. Architecture du Net.

Le protocole utilisé pour la navigation sur le Web s'appelle "HyperText Transfer Protocole" (*HTTP*) et il permet, de manière générale, la communication de fichiers entre un "serveur" (appelé serveur Web ou serveur Internet) et une station cliente (appelée également "client"). Comme nous le verrons dans le chapitre lui étant consacré (voir Chapitre 4), le *HTML* est un langage permettant de créer des documents constitués principalement de texte et d'images. La grande particularité du *HTML* réside dans les "hyperliens" qui sont des liens vers d'autres fichiers situés sur un serveur quelconque. Ce sont donc ces fichiers *HTML* qui sont transférés du serveur vers votre station lorsque vous indiquez une adresse *URL* à votre navigateur ou que vous activez un hyperlien. Une *URL* ou "Uniform Ressource Locator" est une adresse identifiant un fichier sur un serveur particulier du réseau Internet. Elle se présente sous la forme:

http://www.nom_de_domaine.extension/fichier.html

exemples *d'extension*:	be	→	BElgique
	fr	→	FRance
	com	→	COMmercial
	ac	→	Académique

cette extension situe le serveur Web par rapport au réseau global

Exemple 4: Adresse *URL*

Imaginons par exemple que nous désirons consulter le fichier "mémoire.html" qui se situe sur le serveur "droli.com". La figure suivante illustre brièvement la procédure suivie pour avoir finalement le fichier demandé sur son navigateur:

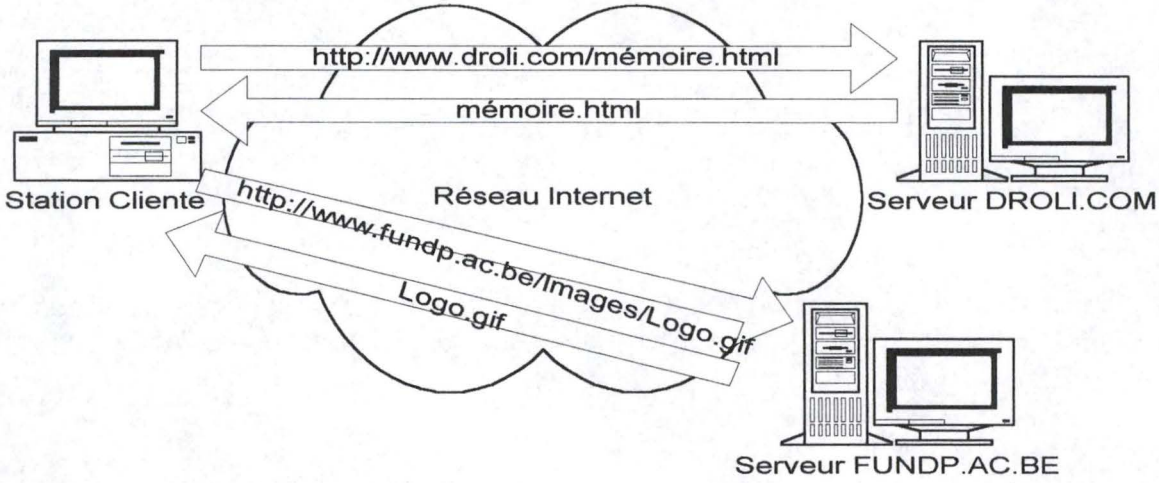

Figure 4: Transfert de fichiers via Internet.

Sur la Figure 4, la station cliente demande une connexion via l'adresse du serveur "droli.com" pour accéder au fichier "mémoire.html". Le serveur envoie alors le fichier demandé vers la station, sur laquelle le navigateur va interpréter le *HTML* pour afficher le texte à l'écran. Dans notre exemple, une image est insérée dans le document reçu, c'est-à-dire qu'il y a un lien vers une image. Ce lien n'est rien d'autre qu'une adresse *URL*, ce qui veut dire que le navigateur va de nouveau faire une demande vers le serveur identifié par l'*URL*. Ici, l'image se trouve sur le serveur des Facultés, dans le répertoire "Images" et se nomme "Logo.gif". La procédure est donc identique au premier appel.

3.1.1. Protocole HTTP.

Le protocole *HTTP* est très simple. Un client, via un navigateur, effectue une requête, le serveur Web lui répond et la transaction est terminée. Il n'existe aucun lien entre plusieurs transactions.

3.1.1.1. La requête.

Voici à quoi ressemble une requête de connexion envoyée par le client au serveur:

```
GET /Site/Homepage.html HTTP/1.0
User-Agent: Mozilla/4.0 (compatible; MSIE 4.0; Windows 95)
Accept: www/source
Accept: text/html
Accept: image/gif
...

D'autres informations peuvent être ajoutées à la suite de ce bloc.
```

Exemple 5: Requête de connexion passée au serveur Web (protocole *HTTP*).

Lorsque le client envoie sa requête, la première chose qu'il spécifie est une commande *HTTP*, appelée une "méthode", spécifiant au serveur le type d'action qu'il désire. Cette première ligne spécifie également l'adresse *URL* du document et la version du protocole *HTTP* a utiliser. Dans l'exemple précédent, le client réclame donc la page nommée *Homepage.html* située dans le répertoire */Site/*, suivant la méthode "GET" et en utilisant le protocole *HTTP/1.0*.

```
GET /Site/Homepage.html HTTP/1.0
```

Après l'envoi de la requête, le client peut ajouter des informations optionnelles. Ce bloc constitue alors "l'en-tête" ("header") de la requête et il spécifie au serveur des informations comme par exemple le type de software utilisé par le client et le type de fichiers qu'il sait interpréter. Ces types sont définis par leur spécification MIME (Multipurpose Internet Mail Extension) utilisée pour l'identification des données envoyées sur le Web.

```
User-Agent: Mozilla/4.0 (compatible; MSIE 4.0; Windows 95)
Accept: www/source
Accept: text/html
Accept: image/gif
```

Après le "header", le client envoie une ligne blanche pour indiquer la fin de cette section d'en-tête. En fonction de la méthode utilisée, le client peut ensuite insérer un ensemble de données destinées à être traitées par le serveur avant l'envoi d'une réponse. Nous y reviendrons au chapitre consacré aux formulaires *HTML* (voir 4.4).

3.1.1.2. La réponse.

Après réception d'une requête par un serveur Web, celui-ci l'analyse et renvoie une réponse au client du genre de l'exemple qui suit.

```
HTTP/1.0 200 OK
Date: Saturday, 23-May-98 03:25:12 GMT
Server: JavaWebServer/1.1.1
MIME-version: 1.0
Content-type: 1029
Last-modified: Thursday, 7-May-98 12:15:35 GMT
...
```

Exemple 6: Réponse du serveur au client (protocole *HTTP*).

La première ligne de la réponse est un "ligne d'état" ("status line") spécifiant la version du protocole *HTTP* utilisée par le serveur, un code d'état ("status code"), et une description pour ce code:

```
HTTP/1.0 200 OK
```

Dans l'exemple, le code 200 signale que la requête est un succès, d'où le *OK* en description de code. Un autre code fréquent est le 404 avec la description "Not Found", qui, faut-il le préciser, signifie que le document demandé n'a pas été trouvé.

Après la ligne de "status", le serveur envoie un "en-tête" de réponse précisant au client certaines choses comme le type de software utilisé par le serveur et le type de contenu envoyé en réponse.

```
Server: JavaWebServer/1.1.1
...
Content-type: 1029
```

Le serveur envoie ensuite une ligne blanche pour conclure la section d'en-tête. Si la réponse est un succès, les données demandées sont envoyées à la suite. Sinon, la réponse correspond à une information spécifiant le pourquoi de l'échec.

3.2. Les navigateurs (browsers).

Le navigateur, ou "browser" en anglais, est l'outil de navigation sur l'Internet. Cette interface utilisateur permet la connexion aux sites Web désirés et affiche le contenu des pages en interprétant le code *HTML* (voir Chapitre 4). On trouve bon nombre de navigateurs différents à l'heure actuelle. "Netscape Communicator/Navigator" et "Microsoft Internet Explorer" sont les plus utilisés mais on peut citer également "Mosaic", "Lynx", "Quaterdeck Mosaic", "Emissary", "Mariner", "MacWeb", etc. Chacun de ces outils possèdent ses caractéristiques propres et une page affichée avec un navigateur peut apparaître légèrement différente sur un autre navigateur. La structure générale reste cependant la même (heureusement !).

3.2.1. Les "plug-ins".

En plus des fonctions de bases intégrées dans le navigateur (lecture du code *HTML*, gestion de l'historique des pages visitées, etc.) la plupart de ceux-ci permet d'y accoler des "plug-ins". Les "plug-ins" sont des logiciels qui viennent se greffer sur le "browser" pour ajouter des fonctionnalités non supportées d'origine par ce dernier. Aujourd'hui, on trouve plus de cent "plug-ins" sur le marché (souvent gratuit) mais peu d'internautes les utilisent pour plusieurs raisons. Principalement, les "plug-ins" posent pas mal de problèmes de sécurité puisqu'ils sont capables d'exécuter n'importe quelle commande sur le système client. Si la source n'est pas connue, il est donc préférable de ne pas se risquer au chargement de tels software. D'autre part, les applications avec "plug-ins" ralentissent généralement fortement le chargement des pages Web, ce qui est souvent très mal vu par les utilisateurs d'Internet. Toutefois, certains "plug-ins" sont très pratiques et très utilisés comme "RealPlayer" permettant d'écouter de la musique "on-line", transformant l'ordinateur en véritable station radio ou en juke-box. On peut également citer "Macromedia Flash" qui permet de créer des clips graphiques prenant peu de place et apparaissant à l'écran à allure plus que raisonnable. Citons encore "QuickTime" pour la vision de vidéo sur Internet (résolution assez limitée) et "CoolTalk" pour la téléphonie.

On distingue trois types de "plug-ins", les cachés ("hidden"), ceux qui remplissent toute la page du navigateur ("full-page") et ceux qui prennent une portion de la page ("embed"). Par exemple, un "plug-in" de musique peut-être caché s'il ne nécessite pas d'opération de la part de l'utilisateur, ou partiel ("embed") s'il prévoit un panneau de contrôle.

3.3. Les serveurs Web.

Le serveur Web est la partie logicielle se trouvant sur le système serveur et destiné à transmettre les documents aux machines clientes qui en font la requête. On trouve actuellement une multitude de serveurs Web dont les principaux sont: "Apache", "Netscape Enterprise Server", "Netscape FastTrack Server", "Windows NT Internet Information Server (*IIS*)", "Windows NT Workstation Personal Web Server", "Windows 95 Personal Web Server", "OmniHttpd", etc.

La fonctionnalité principale d'un serveur Web consiste donc à analyser la requête d'un client, à chercher le fichier demandé et à faire parvenir ce document via le protocole décrit plus haut (3.1.1). Tous ces fichiers doivent se trouver sur le système serveur où est installé le serveur Web. De plus, seuls certains répertoires sont accessibles par ce serveur afin de protéger les informations et les applications non destinées au Web. Toujours à propos de

cette sécurité, il est important que le serveur Web travaille en étroite collaboration avec le système d'exploitation puisque c'est ce dernier qui détermine les droit d'accès pour chaque personne. Ainsi, lorsqu'un internaute désire accéder à un fichier, le serveur Web vérifie d'abord les droits d'accès sur celui-ci. Si les droits sont au minimum, tout le monde peut y accéder et le transfert à donc lieu. Dans le cas contraire, un login et un mot de passe sont nécessaires. Le serveur Web envoie donc un message de refus de connexion avec la possibilité de fournir une authentification. C'est alors au tour du navigateur de jouer par l'ouverture d'une boite de dialogue avec l'utilisateur lui demandant d'entrer un login et un mot de passe. La requête part à nouveau vers le serveur mais avec l'authentification cette fois. Le serveur Web vérifie la validité de la requête et transmet alors le fichier demandé si l'accès est permis. Si l'accès est de nouveau refusé, le processus est réitéré jusqu'à l'abandon de la requête ou jusqu'à ce que l'authentification soit valable.

Si la requête demandée correspond à l'exécution d'une application, le principe sera le même que celui expliqué ci-dessus mais la prudence est encore plus de rigueur puisque le programme peut exécuter toute commande sur le système serveur. On veillera donc à ce que ces programmes soient bien protégés de telle manière que l'exécution de commandes "dangereuses" soient évitées. Imaginons par exemple qu'un utilisateur puisse lancer une commande de formatage du disque à partir d'une application Internet. Les dégâts pourraient bien entendu être très conséquents.

3.4. Technologies "Server-side" et "Client-side".

La présente discussion provient d'une réflexion sur la différence entre le "Server-Side JavaScript " et le "Client-Side JavaScript" que l'on peut trouver au chapitre 4 du site de "Nestcape" sur le "Server-Side JavaScript" {2}.

L'environnement client (le navigateur) représente la face visible d'une application Internet. Dans cet environnement, des pages *HTML* sont affichées dans des fenêtres et un historique des pages visitées est géré par le navigateur. Tout objet technologique dans cet environnement doit donc être capable, entre autres, de manipuler des pages, des fenêtres et des historiques. On parle dans ce cas de technologies "Client-Side", c'est-à-dire de technologies gérant les ressources du côté du client, sur la station de travail.

Par contraste, dans l'environnement du serveur, il faut pouvoir utiliser les ressources de celui-ci. Par exemple, on désirera connecter une base de données "relationnelle", partager de l'information entre utilisateurs d'une application, ou manipuler les fichiers sur le serveur. Tout objet technologique dans cet environnement doit donc être capable entre autre de manipuler des *BD* "relationnelles" et le système de fichier du serveur. On parle dans ce cas de technologies "Server-Side", c'est-à-dire de technologies gérant les ressources du côté du serveur.

Lors du développement d'une application pour l'Internet, le choix des technologies devra se faire en fonction des exigences tout en gardant à l'esprit les grandes différences entre une station de travail et un serveur, exprimées dans le tableau suivant:

SERVEURS	CLIENTS
Les serveurs sont généralement des stations de travail à grandes performances avec un processeur rapide et une large	Les stations clientes sont souvent des systèmes à processeur moins puissant et à capacités plus limitées.

capacité de stockage.	
Les serveurs peuvent être saturés lorsque les accès deviennent trop nombreux (centaine de clients connectés pour des serveurs moyens).	Les machines clientes sont souvent utilisées par une seule personne. Il peut donc être avantageux d'y charger une partie du programme pour y effectuer certaines opérations qui libéreront ainsi le serveur.
	L'analyse des données du côté client peuvent réduire la largeur de bande exigée pour le transfert si l'application décompose ces données.

La meilleure solution sera bien évidemment celle qui profitera au mieux de ces deux types de technologies. En effet, il y a généralement plusieurs façons pour partager une application entre le client et le serveur. Certaines tâches ne peuvent être exécutée que sur le client et d'autres uniquement sur le serveur. Les autres tâches peuvent être exécutées sur les deux. Bien qu'il n'existe pas de règles définitives pour savoir où faire quoi, on peut tout de même citer les grandes lignes suivantes:

On utilisera généralement une technologie "Client-Side" pour les tâches suivantes:

- valider une entrée utilisateur, c'est-à-dire vérifier qu'une valeur dans un formulaire est valide;
- demander confirmation à l'utilisateur, afficher une erreur ou une information;
- calculer des sommes ou des moyennes à partir d'information fournies par l'utilisateur ou provenant du serveur;
- exécuter toutes fonctions n'exigeant pas d'information sur le serveur.

Par contre, on utilisera généralement une technologie "Server-Side" pour les tâches suivantes:

- maintenir des informations à propos d'un ensemble d'accès client;
- maintenir le partage de données entre plusieurs clients ou plusieurs applications;
- accéder à une base de données ou à des fichiers sur le serveur;
- utiliser une application située sur le serveur;
- cacher le code de l'application et ainsi envoyer uniquement les informations nécessaires (sécurité + gain de temps de transfert);
- créer dynamiquement des pages Web.

A propos de la génération dynamique de pages Web, on distingue deux façons de faire. La première consiste à créer la page de A à Z au moment de l'appel. La deuxième techniques est ce qu'on appelle le "Server-Side Include" ou *SSI*. Dans ce dernier cas, une page type est déjà créée mais elle contient toutefois certaines données inconnues au moment de la création. Ce n'est qu'au moment de l'appel, avant de l'envoyer au client, que

la page sera complétée. Les *SSI* requièrent donc une analyse du contenu par le serveur Web au moment de l'appel. Le serveur, reconnaissant le type de document demandé, va décider d'analyser les données, va exécuter les parties de programmes insérées et va générer ainsi une page lisible par un navigateur.

3.5. Bibliographie.

HTTP.

[3] Jason Hunter et William Crawford, "*JAVA Servlet Programming*", O'Reilly & Associates, United States of America, Octobre 1998.

Sécurité.

[4] Paul Enfield, "*Implementing a secure site with ASP*", Microsoft Corporation, 17/07/1998

Sites "plug-ins".

{1} "Un Gouveau Guide Internet - UNGI" – Chap. 49: Quelques "plug-ins" Netscape: http://ungi.cge.net/cyber/

Sites technologies "Serveur-Side" et "Client-Side".

{2} "Server-Side JavaScript" - Writing Server-Side JavaScript Applications: http://developer.netscape.com/docs/manuals/enterprise/wrijsap/index.html

Chapitre 4. HTML.

4.1. Introduction.

Le *HTML* ("HyperText Markup Language") est le langage actuellement le plus utilisé pour le développement des pages Web. Son principe est très simple et ne requiert aucune connaissance approfondie en informatique: un fichier *HTML* est un fichier texte (un simple éditeur de texte suffit donc) où tout élément particulier doit s'écrire à l'aide de "balises" (ou "marqueur", ou encore "tag" en anglais) prédéfinies par le langage. Ces marqueurs servent à indiquer au navigateur comment il doit afficher les données à l'écran. Par bonheur, presque tous les navigateurs affichent la plupart des codes *HTML* de la même manière. Ainsi, si cela fonctionne avec un navigateur, vous pouvez être presque sûr que cela fonctionnera avec tous les autres navigateurs. Attention toutefois: certains navigateurs n'ont jamais dépassé le premier stade de leur développement et ils n'affichent pas du tout ce que vous attendez. D'autres sont plus anciens et ne bénéficient donc pas des derniers progrès du langage *HTML*. Il est donc très utile de savoir que les navigateurs les plus utilisés sont "Netscape Navigator" et "Microsoft Internet Explorer", ces deux outils ayant également quelques particularités différentes. La prudence veut donc qu'on utilise le plus possible les parties du langage *HTML* qu'ils ont en commun, et qu'on spécifie toujours pour quel type de navigateur le site a été conçu et testé.

4.2. Le principe des marqueurs / balises.

Le "marqueur", la "balise" ou le "tag" est l'unité de base du codage dans le système *HTML*. Tout dépend des marqueurs dans *HTML*.

Les balises sont délimitées par les signes "inférieur à" et "supérieur à" (**<** et **>**). Par exemple, la balise de paragraphe est **<P>**, la balise pour une ligne horizontale est **<HR>** et celle pour un retour à la ligne est **
**.

La deuxième chose importante à savoir à propos des marqueurs, est la constitution de "blocs". En effet, la majeure partie de la programmation en *HTML* consiste à placer des éléments dans des blocs qui possèdent un marqueur d'ouverture et un marqueur de fermeture. Les marqueurs de fermeture sont identiques aux marqueurs d'ouverture à la différence qu'ils sont précédés d'une barre oblique (/). Par exemple, le marqueur d'ouverture pour l'écriture italique est **<I>** alors que le marqueur de fermeture est **</I>**. Tout texte se trouvant dans le "bloc" italique (entre les balises d'ouverture et de fermeture) sera affiché par le navigateur en italique.

Voici un exemple de document *HTML*:

```
<HTML>
```

```
<HEAD>
<TITLE>Texte apparaissant dans la barre de titre du navigateur</TITLE>
</HEAD>

<BODY TEXT="#000000" LINK="#0000FF" VLINK="#ffff00"
     BGCOLOR="#FFFFFF">
Ceci est un simple texte
<BR>
<A HREF="http://www.info.fundp.ac.be">Ceci est un hyperlien vers
     le site des FUNDP</A>
<BR>
Et ci-après vous devriez voir une image:
<BR>
<IMG SRC="LogoFUNDP.gif">
<HR><HR>
</BODY>

</HTML>
```

Exemple 7: document *HTML*

On peut distinguer 3 blocs principaux dans cet exemple: le bloc délimitant le document *HTML*, délimité par les balises **<HTML>** et **</HTML>**, le bloc de tête, **<HEAD>** et **</HEAD>**, ainsi que le corps du document, **<BODY>** et **</BODY>**. A propos de ce dernier bloc, on remarque qu'une balise d'ouverture peut contenir des "arguments" (certains étant facultatifs, d'autres obligatoires). Ici, ces arguments concernent les couleurs (définie en hexadécimales) du documents: couleur du texte (**TEXT=**), couleur des "hyperliens" (**LINK=**), couleur des "hyperliens" déjà visités (**VLINK=**) et couleur de fond (**BGCOLOR=**). Ceci ne constitue évidemment qu'un échantillon des balises et des arguments que l'on rencontre en *HTML*.

Voici à quoi ressemble ce document *HTML* une fois chargé avec "Netscape Navigator":

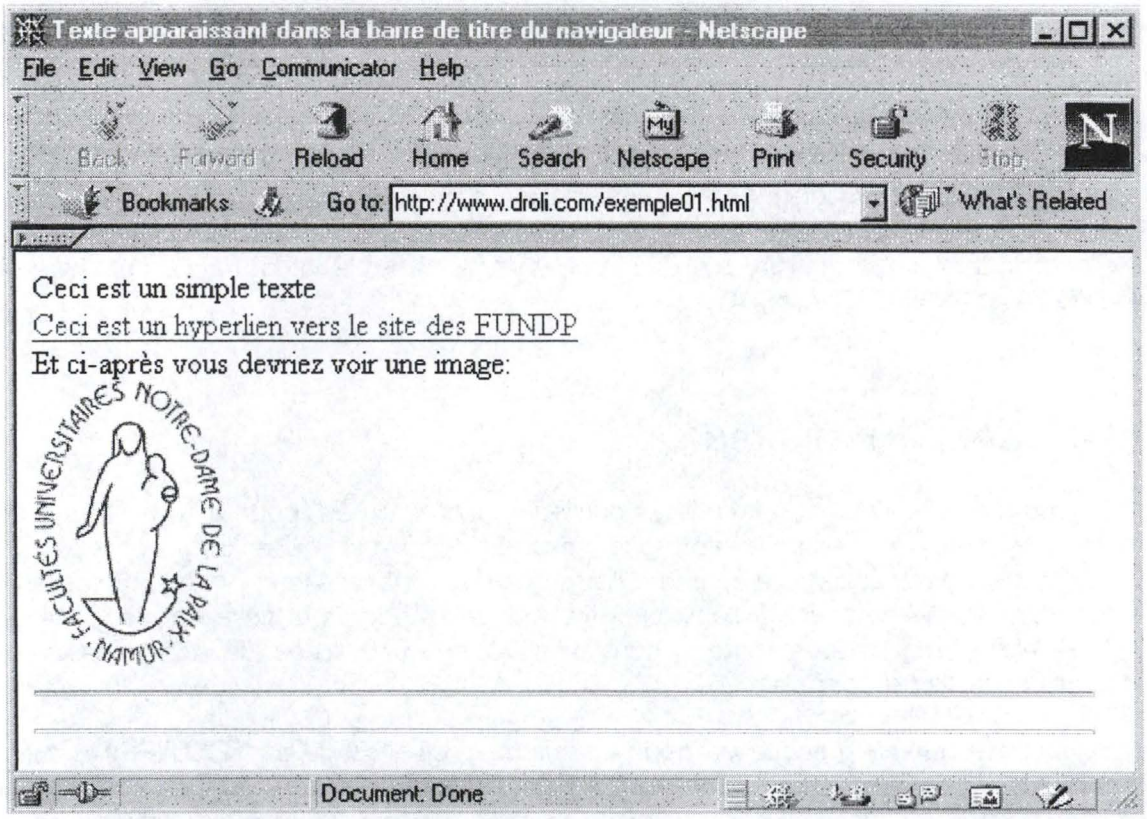

Figure 5: Illustration du *HTML* sur Netscape.

4.3. Les "hyperliens".

Dans l'exemple qui précède, on peut observer une ligne écrite en bleu et soulignée. Dans le code *HTML*, cette ligne correspond au code suivant:

```
<A HREF="http://www.info.fundp.ac.be">Ceci est un hyperlien vers
    le site des FUNDP</A>
```

Exemple 8: Insertion d'un hyperlien en *HTML*

C'est un "hyperlien", la technologie principale à la base du *HTML*.

La couleur de ce texte est caractéristique du site (définie par l'argument **LINK** de la balise **<BODY>**) et permet généralement de distinguer les hyperliens sur une page *HTML*.

Lorsque le "surfeur" clique sur de tels liens, il donne en fait l'ordre au navigateur de charger le document qui se trouve à l'*URL* décrite dans l'argument **HREF=** de la balise **<A>** (A pour ANCHOR, ou ANCRAGE). Un lien peut être "interne" (vers un document annexe appartenant au même site) ou "externe" (vers un autre site internet).

Pour rester dans les liens, il est utile de savoir que toute information autre que du texte constitue également un lien en *HTML*. Dans notre exemple, l'insertion d'une image se fait par la balise **<IMG>** avec l'argument **SRC=** (SRC pour SOURCE)désignant le fichier à charger à cet endroit:

```
<IMG SRC="LogoFUNDP.gif">
```

Exemple 9: Insertion d'une image en *HTML*

La source est un nom de fichier précédé éventuellement par un ensemble de répertoire. Si l'image ne se trouve pas sur le même serveur que la page *HTML*, on peut également la localiser par son adresse *URL* complète.

Citons également les "images réactives", très fréquentes sur le Net. Ce sont en fait des hyperliens déguisés en images: au lieu d'avoir un texte en surbrillance, l'utilisateur peut cliquer sur cette image avec le même effet de chargement d'une nouvelle page. Techniquement, il suffit de définir une image dans le bloc dédié aux hyperliens: **<A HREF=...><IMG SRC=...></A>**.

4.4. Les formulaires.

Les formulaires sont très utilisés sur le Web, notamment pour la consultation des bases de données. Ces formulaires sont destinés à envoyer des informations concernant l'utilisateur au propriétaire du site, c'est-à-dire du client vers le serveur. On trouve beaucoup de formulaires pour les "livres d'or", souvent utiles pour mettre en évidence la popularité d'un site Web: chaque visiteur peut ainsi laisser une trace de son passage avec ses appréciations personnelles.

Il existe quatre marqueurs pour les formulaires: **<FORM>**, **<TEXTAREA>**, **<SELECT>** et **<INPUT>**. Le premier sert à délimiter le formulaire et prend entre autre comme paramètre **(ACTION=)** l'adresse *URL* du programme à exécuter sur le formulaire. De tels programmes seront possibles via les technologies "Server-Side" que nous détaillerons plus loin comme "Java" (Chapitre 7), les "Scripts *CGI*" (Chapitre 8), le "JavaScript" (Chapitre 9), le *ASP*, etc.

Les autres balises liées aux formulaires sont illustrées dans l'exemple qui suit:

```
<FORM ACTION=URL_action>

Ceci est un "TEXTAREA":
<TEXTAREA ROWS="3" COLS="30">Tapez vos commentaires ici</TEXTAREA>

<BR><BR>

Ici, vous pouvez faire un choix, c'est un "SELECT":
<SELECT>
      <OPTION>choix 1
      <OPTION>choix 2
      <OPTION>choix 3
</SELECT>

<BR><BR>

Et ici, vous pouvez voir les différents champs d'"INPUT":<BR>
-Insertion de texte: <INPUT TYPE="TEXT" NAME="prénom" SIZE="15"
      MAXLENGTH="13"><BR>
-Insertion d'un mot de passe (le mot s'écrit avec des astérisques "*"): <INPUT
      TYPE="PASSWORD" NAME="mot_de_passe" SIZE="8"
      MAXLENGTH="8"><BR>
-Les cases à cocher (plusieurs choix permis parmi plusieurs):
      <INPUT TYPE="CHECKBOX" NAME="choix 1">
      <INPUT TYPE="CHECKBOX" NAME="choix 2" CHECKED >
      <INPUT TYPE="CHECKBOX" NAME="choix 3" CHECKED><BR>
-Les boutons radio (un seul choix permis parmi plusieurs):
```

```
        <INPUT TYPE="RADIO" NAME="critère 1">
        <INPUT TYPE="RADIO" NAME=" critère 2" CHECKED>
        <INPUT TYPE="RADIO" NAME=" critère 3"><BR>
-Le bouton de restauration: <INPUT TYPE="RESET" VALUE=""Cliquez pour
        réinitialiser"><BR>
-Le bouton de soumission: <INPUT TYPE="SUBMIT" VALUE="Cliquez pour soumettre
        vos données"><BR>
</FORM>
```

Exemple 10: Formulaire *HTML*

Chaque balise supporte des paramètres qui ne sont pas tous représentés ici. Notamment, l'argument **NAME=** peut être inséré dans la plupart des marqueurs pour pouvoir les nommer et les gérer localement via du "JavaScript" par exemple (voir 4.6) ou sur le serveur via une technologie "Server-Side". La figure suivante nous montre ce que donne le code *HTML* de l'exemple précédent sous Netscape:

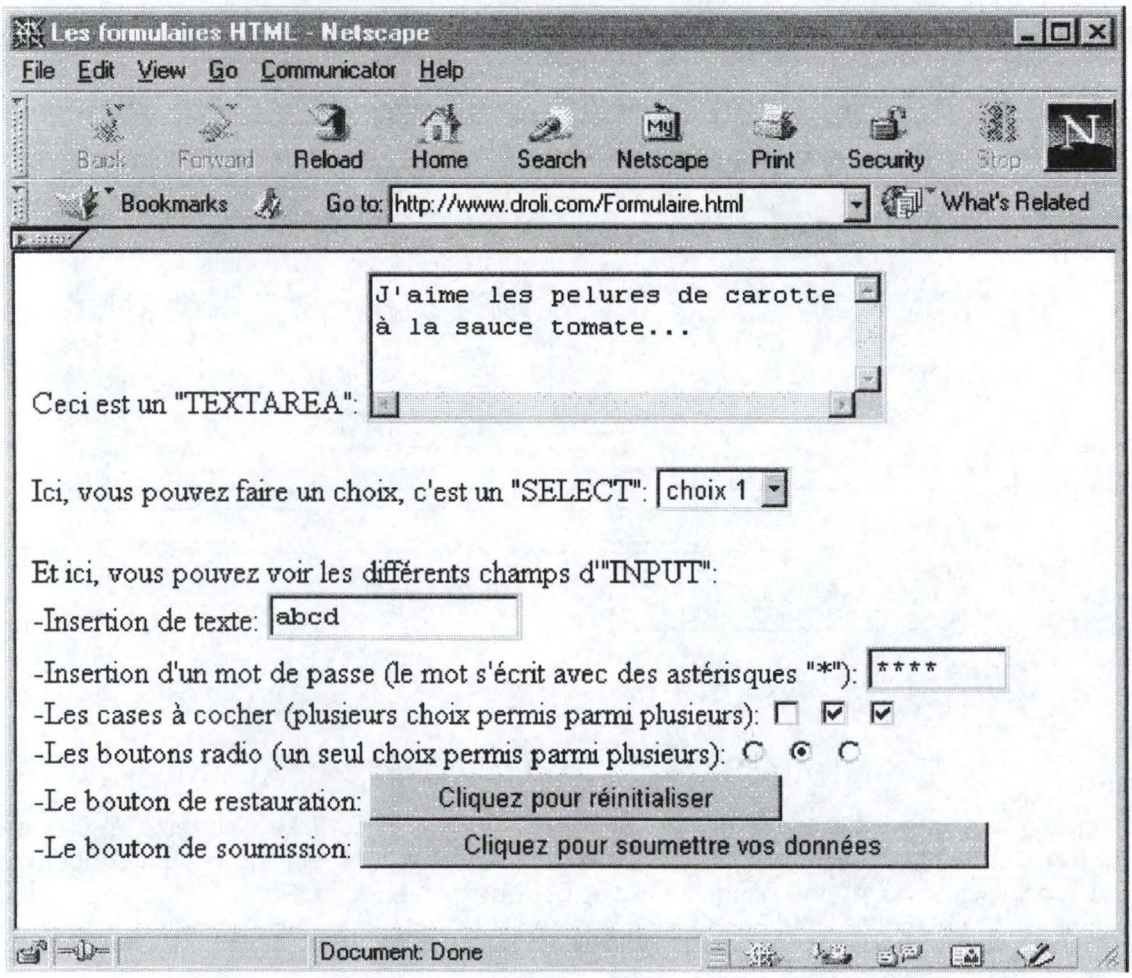

Figure 6: Illustration des formulaires *HTML*

On notera enfin que ces balises permettent une gestion de l'interface pour des formulaires mais ne constituent pas une solution de gestion des informations qui y seront introduites. Pour cela, il faut se référer à d'autres technologies.

4.4.1. Les méthodes d'envoi de données.

Lors de l'envoi d'une requête vers un serveur Web, le client doit déterminer la méthode utilisée (voir protocole *HTTP* - 3.1.1). Les deux méthodes principales sont GET et POST et elles peuvent être utilisées lors de l'envoi de données via un formulaire *HTML*. On signale la méthode désirée via l'attribut **METHOD=** de la balise <FORM>.

4.4.1.1. Méthode GET.

Un formulaire *HTML* utilisant la méthode GET ressemble donc à ceci:

```
<FORM ACTION="/traitement.exe" METHOD="GET">
Entrez votre nom: <INPUT TYPE="text" NAME="nom" SIZE="40">
<BR>
Entrez un mot: <INPUT TYPE="text" NAME="mot" SIZE="40">
<BR>
<INPUT TYPE="submit" VALUE="Cliquez pour envoyer">
</FORM>
```

Exemple 11: Scripts *CGI* - Méthode GET.

Si on ouvre ce document sous Netscape, on obtient le formulaire suivant:

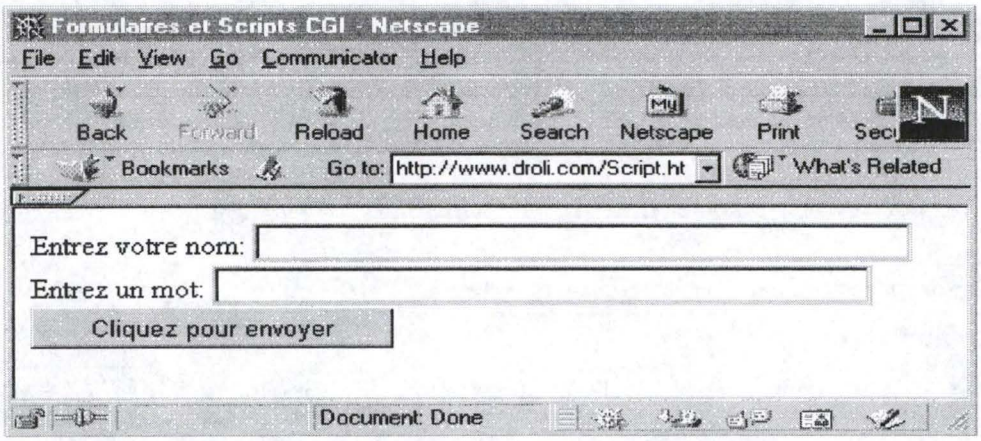

Figure 7: Illustration d'un formulaire *HTML*

Imaginons que l'utilisateur entre le nom "Olivier", le mot "Internet" et qu'il clique sur le bouton d'envoi. Le navigateur va alors se connecter au serveur de manière équivalente à une connexion normale lorsqu'on spécifie une adresse *URL* ou lorsqu'on active un "hyperlien". La seule différence réside dans le passage de données provenant du formulaire. Si on analyse le document *HTML* décrivant le formulaire, on s'aperçoit que les deux champs de texte (<INPUT TYPE="texte" ...>) possèdent un attribut **NAME**. C'est ce paramètre qui va jouer le rôle d'identifiant. On a donc un premier champ appelé "nom" auquel correspond l'entrée "Olivier" et un deuxième nommé "mot" lié à l'entrée "Internet". L'activation du bouton de soumission va alors provoquer une connexion correspondant à l'adresse suivante:

```
http://www.droli.com/traitement.exe?nom=Olivier&mot=Internet

où   http://www.droli.com est l'adresse du serveur où réside la page courante
     traitement.exe est le programme indiqué dans la balise FORM (attribut ACTION=)
     qui est censé analyser les données fournies et renvoyer une réponse au client.
```

Exemple 12: Adresse *URL* correspondant à un appel *CGI* en méthode GET.

La partie qui succède au point d'interrogation est ce qu'on appelle un "string de requête" ou "query string". Cette adresse est tout à fait utilisable telle quelle et aura le même effet qu'un remplissage des champ du formulaire.

Une fois une telle adresse passée au serveur, celui-ci décide de lancer l'application associée (ici, "traitement.exe") après avoir transféré le "query string" dans une variable d'environnement ('QUERY-STRING' sous UNIX). Ce string est alors à disposition pour le programme qui doit découper l'information en blocs utilisables: en analysant la chaîne de caractères, il peut récupérer les valeur "Olivier" et "Internet" pour les mettre dans des variables et les utiliser à sa guise.

4.4.1.2. Méthode POST.

Un formulaire *HTML* utilisant la méthode POST ressemble à ceci:

```
<FORM ACTION="traitement.exe" METHOD="POST">
Entrez votre nom: <INPUT TYPE="text" NAME="nom" SIZE="40">
<BR>
Entrez un mot: <INPUT TYPE="text" NAME="mot" SIZE="40">
<BR>
<INPUT TYPE="submit" VALUE="Cliquez pour envoyer">
</FORM>
```

Exemple 13: Scripts *CGI* - Méthode POST.

Et l'affichage graphique est inchangé (voir Figure 7).

La seule différence avec la méthode GET au niveau du document *HTML* se trouve donc dans la valeur de l'attribut METHOD de la balise FORM. Par contre, il existe une distinction au niveau de l'envoi des données. En effet, la méthode POST envoie ces informations comme un flux de données: la connexion est équivalente à une connexion habituelle vers l'adresse du programme mais les données du formulaire sont insérées à la fin du bloc de requête de connexion (voir 3.1.1).

De son côté, le serveur va lancer l'application en lui passant les informations comme flux de données, c'est-à-dire dans l'entrée standard ("STDIN" sous "Unix"). Ces données sont contenues dans un string comme c'était le cas pour la méthode GET (même format). L'analyse est donc identique au cas précédent.

4.4.1.3. GET ou POST ?

Les deux méthodes ont leurs avantages et leurs inconvénients. On peut formuler principalement les remarques suivantes:

- La méthode GET place les informations dans une variable d'environnement. Or, certains systèmes limitent la longueur de ces variables à un certain nombre de caractères. Si l'information est trop longue, le script ne marchera pas. Par contre, la méthode POST utilise un flux de données, ce qui signifie qu'il n'y a pas de limite pratique à ces données (l'administrateur doit fixer une limite mais elle est souvent très élevée);

- La méthode GET est équivalente à une connexion via l'*URL* à laquelle on ajoute de l'information. Si l'on désire effectuer plusieurs fois la même recherche (après un certain temps, par exemple) on peut mémoriser l'adresse dans un "bookmark" (signet ou "marques-ta-page"), ce qui permet de ne pas devoir remplir le formulaire à chaque fois. La méthode POST ne permet pas une telle solution.

4.5. Premières remarques.

Les possibilités offertes par le *HTML* sont assez limitées. En effet, on peut dire que les documents créés sont de simples feuilles de textes et d'images (ou autre document multimédia) mais avec le gros avantage de pouvoir faire des liens entres les différentes pages et vers d'autres sites Web grâce aux hyperliens (et aux images réactives).

Au niveau de la programmation *HTML*, il existe bon nombre d'éditeurs qui peuvent faire le travail pour vous: il vous libère de la corvée de la saisie des marqueurs et vous permettent de travailler comme dans un traitement de texte. D'ailleurs, Microsoft incorpore une fonction de transformation de ses fichiers "Word" en fichiers *HTML* depuis la version 97. On peut citer également "Live Markup for Windows", "WebForms" (idéal pour les formulaires), "HotDog Professional" ou encore "Netscape Navigator Gold" qui permet de modifier et de créer des codes *HTML* à partir du navigateur "Netscape".

4.6. "JavaScript".

4.6.1. "JavaScript" et "Java".

Le "JavaScript" est un langage de programmation dans la même lignée que le "Java" (que nous analyserons plus loin - Chapitre 7) mais avec toutefois de grandes différences. Pour être bien clair: "JavaScript" n'est pas une version simplifiée de "Java". La similarité des noms n'est que pur marketing. D'ailleurs "JavaScript" devait s'appeler à l'origine "LiveScript" mais son nom a été changé (malencontreusement ?) à la dernière minute. Pour une comparaison plus approfondie sur ces deux langages, on se référera au point 7.2 au chapitre concernant "Java".

"JavaScript" est un langage "orienté objets" (*O.O.*). Un programme "JavaScript" est non compilé et contenu dans le corps de la page *HTML*. Ce n'est qu'au moment du chargement de la page, et donc du code "JavaScript", par le browser que le programme sera interprété et exécuté.

La syntaxe du "JavaScript" est semblable à celle de langages tels "C", "C++" et "Java", mais de manière nettement simplifiée. Cette particularité en fait un langage facile à apprendre et à utiliser par les auteurs de pages *HTML*.

4.6.2. "JavaScript", "Jscript" et "ECMAScript".

Le "JavaScript" est supporté par la majorité des navigateurs ("Netscape 3.0+", "Internet Explorer 3.0+", mais aussi "Opera 3" par exemple). Toutefois, "JavaScript" est un produit de

"Netscape" mais, logiquement, "Microsoft" propose également le même genre de langage sous le nom de "JScript". Bien que ces deux langages soient plus ou moins compatibles, il existe quelques différences notamment au niveau des fonctions destinées au *DHTML* (voir Chapitre 5).

"JavaScript" a été standardisé par l'*ECMA* ("European Computer Manufacturers Association") et est sur la voie d'une standardisation par l'*ISO* ("International Standards Organization). Ces standards définissent un langage connu officiellement sous le nom de "ECMAScript" et qui est l'équivalent approximatif du "JavaScript 1.1". Notons que ce langage est un mélange du "JavaScript" et du "JScript", et que donc il n'en favorise aucun des deux en particulier.

4.6.3. Utilisation du JavaScript.

"JavaScript" est donc un langage de programmation destiné à des applications simples, soit pour accéder a des ressources systèmes de la machine cliente, soit pour rendre une page plus réactive:

4.6.3.1. Accès aux ressources systèmes.

Dans cette catégorie de programme, on retrouve un ensemble de fonctions permettant l'accès à l'heure ou à la date de la machine cliente ou encore des possibilités de chargement de fichiers depuis le client vers le serveur.

4.6.3.2. Réactivité des pages.

L'utilisation la plus fréquente du "JavaScript" concerne la réactivité au niveau du client. En effet, bon nombre d'applications sont destinées à améliorer l'interface graphique du site en présentant par exemple un menu constitué d'images: en fonction de la page affichée, l'image correspondant à cette page apparaît différemment dans le menu.

Une autre grande utilité du "JavaScript" réside dans une première vérification de formulaires. En effet, étant donné qu'une connexion avec le serveur coûte cher en temps et en argent, il est souvent très intéressant de faire quelques vérifications localement, sur la machine cliente. Par exemple, si un des champs du formulaire est obligatoire, un simple test "JavaScript" permettra de faire une remarque à l'utilisateur avant l'envoi des données au serveur. De même, si une adresse électronique doit être inscrite dans un des champs, le programme local pourra vérifier que le caractère "@" fait bien partie de la chaîne de caractères.

Une remarque doit être faite ici en ce qui concerne la sécurité. En effet, on serait tenté de faire le maximum de vérification sur les données en local. Cependant, il y a généralement moyen de contourner le formulaire et d'envoyer les informations directement via l'adresse *URL*: comme on l'a montré plus haut (4.4.1), il est en effet possible d'exécuter le script correspondant au formulaire en se connectant à une adresse du genre:

```
http://nom_du_serveur/programme?champ1=valeur1&champ2=valeur2
```

où champ1 et champ2 sont les noms des champs du formulaire (pour les connaître, il suffit de parcourir le fichier *HTML* sous format de texte: affichage du fichier source est une fonction d'origine sur les navigateurs)
valeur1 et valeur2 sont les valeurs respectives pour champ1 et champ2

Exemple 14: Connexion *CGI* via l'*URL*

Prudence, donc !!! La vérification locale ne doit pas empêcher une vérification au niveau du serveur.

4.6.4. Sources JavaScript.

Bien que ce langage soit plus complexe que le simple *HTML*, il est tout de même plus facile d'accès que le "Java". Toutefois, des connaissances en programmation "orienté objets" sont nécessaires.

D'autre part, un grand nombre de sites propose un ensemble de fonctions prêtes à l'utilisation. Les non-programmeurs peuvent donc se tourner vers ces applications facilement intégrables à leur page Web. On peut par exemple citer les sites suivants:

- http://www.developer.com;
- http://www.essex1.com/people/timothy/js-index.htm;
- http://www.dynamicdrive.com.

4.6.5. JavaScript et HTML.

"JavaScript" peut être implanté dans une page *HTML* de trois manières:

- par la balise **<SCRIPT>**;
- en utilisant les événements;
- en mettant le code dans un fichier séparé (Netscape 3.0+)

4.6.5.1. Balise <SCRIPT>.

```
<SCRIPT LANGUAGE="JavaScript">
code
</SCRIPT>
```

Exemple 15: Balise <SCRIPT>.

On le voit, le choix du langage ne se limite pas au "JavaScript" et reste ouvert à d'autres langages comme "Visual Basic" supporté uniquement par "Microsoft Internet Explorer" ou "JScript" décrit ci-dessus.

Ce bloc de script est à placer de préférence dans la partie entête (**<HEAD>**) de la page *HTML*. Le code placé à l'intérieur du couple de balises **<SCRIPT>** et **</SCRIPT>** est évalué après le chargement de la page.

L'argument **LANGUAGE=** peut également contenir la version utilisée pour le code. Par exemple, LANGUAGE="JavaScript1.2" signifie que le code qui suit utilise des fonctions de la version 1.2 de "JavaScript". Les navigateurs ne supportant pas cette version n'exécuteront donc pas le script.

Notons que le couple de balise **<NOSCRIPT>** et **</NOSCRIPT>** permet d'encadrer le texte qui sera affiché si "JavaScript" n'est pas compris par le navigateur.

4.6.5.2. Les événements.

Les événements sont les résultats d'une action de l'utilisateur, comme par exemple un clic sur l'un des boutons de la souris. La syntaxe de ces événements est la suivante:

```
<balise eventHandler="code JavaScript"
        où    "balise" est le nom d'une balise
              "eventHandler" est le nom d'un événement
```

Exemple 16: Syntaxe des événements JavaScript.

Par exemple, pour utiliser la fonction "HelloWorld()" quand un utilisateur appuie sur un bouton, l'instruction suivante pourrait être utilisée:

```
<INPUT TYPE="button" onClick="HelloWorld();"
```

Exemple 17: Evénement JavaScript "onClick".

La fonction "HelloWorld()" doit alors être décrite quelque part dans le document *HTML*, de préférence dans le bloc de balise **<HEAD>**. Toutefois, rien n'empêche d'insérer directement le code à la place de l'appel de la fonction, dans la balise même.

A titre d'information, voici quelques événements supportés par "JavaScript", chaque événement étant applicable pour certaines balises particulières:

onClick	click de souris
onLoad	fin de chargement de la page
onMouseOver	passage de la souris sur l'objet correspondant
onMouseOut	sortie de la souris de l'objet correspondant
onSubmit	click de souris sur un bouton de soumission ("submit")
onUnload	l'utilisateur quitte la page en cours

Exemple 18: Quelques événements JavaScript.

4.7. Les "cookies".

4.7.1. Définition.

Un "cookie" est une petite quantité de données stockées par le navigateur sur le client et correspondant à une page ou un site particulier. Les "cookies" jouent le rôle de mémoire pour le browser, de telle sorte que celui-ci peut se remémorer certaines préférences de l'utilisateur ou d'autres variables quelconques lorsque l'utilisateur quitte une page puis y revient. On peut penser par exemple à une adresse email introduite dans un formulaire sur une page et réutilisée par une autre page plus loin dans la navigation. Plutôt que de

demander à l'utilisateur de la ré-encoder, il est plus judicieux de la conserver et de la réutiliser automatiquement.

Cette technologie était destinée à l'origine aux "Scripts *CGI*" (voir Chapitre 8) et est en fait implémentée en tant qu'extension du protocole *HTTP*. Les données contenues dans les "cookies" sont automatiquement transmis entre le navigateur et le serveur Web. Ainsi, les "Scripts *CGI*" sur le serveurs peuvent lire et écrire des données dans les "cookies" stockés sur la machine cliente. Comme nous allons le voir, "JavaScript" peut aussi manipuler les "cookies" en utilisant la propriété *cookie* de l'objet *document*.

4.7.2. Description.

Indépendamment du "JavaScript" qui permet leur gestion, chaque "cookie" possède un nom, une valeur et quatre attributs optionnels. Le couple nom/valeur correspond à la donnée à conserver.

Les quatre attributs sont les suivants:

- **expires=***date*, définissant la date d'expiration du "cookie". Si aucune date n'est spécifiée, il sera détruit lors de la fermeture du navigateur. Dans le cas contraire, le "cookie" sera inséré dans un fichier local prévu par le browser et ne sera détruit que lorsque la date sera dépassée. La prochaine connexion au site transmettra donc automatiquement ce "cookie" au serveur.

- **path=***répertoire*, spécifiant l'ensemble de pages Web concernées par le "cookie". Par défaut, un "cookie" est associé et accessible par la page qui l'a créé ainsi que par les pages contenues dans le même répertoire (directory) ou dans un de ses sous-répertoires. Pour agrandir la zone d'accès du "cookie", il suffit donc de spécifier le répertoire concerné dans l'attribut "path";

- **domain=***domaine*, se situe un niveau au dessus de "path". Par défaut, les "cookies" ont accès uniquement aux pages situées sur le même serveur Web d'où proviennent les pages qui les ont créés. Si l'on veut par exemple que le serveur sur "order.amazon.com" puissent lire les "cookies" créés à partir de "catalog.amazon.com", l'attribut "domain" doit être spécifié en temps que "amazon.com". Notons tout de même que le domaine ne peut pas être extérieur au domaine du serveur.

- **secure**, un booléen spécifiant comment les "cookies" sont transmis sur le réseau. Par défaut, les "cookies" ne sont pas sécurisés, c'est-à-dire qu'ils sont transmis par une connection *HTTP* normale, non sécurisée. Si le "cookie" est marqué "secure", il sera transmis de manière sécurisée si toutefois le browser et le serveur sont connectés via *HTTPS* ou un autre protocole sécurisé.

4.7.3. Gestion par "JavaScript".

Rappelons que les quatre attributs décrits ci-dessus ne sont pas des propriétés d'objets "JavaScript". Les "cookies" sont d'abord des objets externes au "JavaScript", faisant partie du protocole *HTTP*.

Un "cookie" a la forme suivante:

"Nom=*valeur*; expires=*date*; path=*directory*; domain=*domaine*; secure"
où -*valeur* est un string (sans cractères spéciaux comme des virgules, des espaces, etc.) représentant la donnée à conserver
-expires, path, domain et secure sont des attributs facultatifs (voir ci-dessus)

Exemple 19: Format du "cookie".

Afin de pouvoir enregistrer plusieurs données dans un seul "cookie", on peut utiliser le format suivant, par exemple:

"Nom_du_Cookie=*valeur*; expires=*date*; path=*directory*; domain=*domaine*; secure"
où -*valeur* est un string de la forme "*var_1*:*val_1*& *var_2*:*val_2*& *var_3*:*val_3*" (ou d'autres séparateurs spécifiques à la place de ":" et de "&")
-expires, path, domain et secure sont des attributs facultatifs (voir ci-dessus)

Exemple 20: "Cookie" à valeur multiple.

La gestion de ces "cookies" par programmation "JavaScript" peut alors se faire par analyse de la chaîne de caractère contenue dans la propriété *cookie* de l'objet *document*. En fait, *document.cookie* est un string contenant tous les "cookies" liés à la page Web courante. Il a donc la forme suivante:

"Nom_du_Cookie_1=*valeur_1*; expires=*date*; path=*directory*; domain=*domaine*; secure;
Nom_du_Cookie_2=*valeur_2*; expires=*date*; path=*directory*; domain=*domaine*; secure; Nom_du_Cookie_3=*valeur_3*; expires=*date*; path=*directory*; domain=*domaine*; secure;"

Exemple 21: String contenu dans *document.cookie*.

Une simple analyse des caractères permettra de décomposer ce string en une liste de couples "cookie/valeur" puis de décomposer chaque "valeur" en couples "variable/valeur" dans le cas de "cookies" à valeurs multiples (voir Exemple 20).

4.7.4. Limitations.

Les "cookies" sont dédiés à un stockage peu fréquent d'une petite quantité de données. Ils ne doivent donc pas être utilisés comme moyen général de communication ou comme mécanisme de transfert de données. Il convient donc de les utiliser avec modération. D'ailleurs, les navigateurs ne peuvent pas enregistrer plus de 300 "cookies" au total, ni plus de 20 par serveur Web, ni plus que 4 kilobytes de données par "cookie". La contrainte des 20 "cookies" par serveur est la plus restrictive, et il est donc conseillé de ne pas séparer les données à mémoriser en autant de "cookies" mais plutôt d'utiliser le format à valeurs multiples (voir Exemple 20).

4.8. Première évaluation.

La création de page Web risque de passer encore pour un moment par l'utilisation du *HTML*. Ce dernier constitue actuellement une solution très pratique pour la mise à disposition de documents sur l'Internet. Nous allons maintenant décrire les particularités du *HTML* par rapport aux critères définis précédemment (voir Chapitre 2: Critères de conception.). Une évaluation de ces critères en fonction des valeurs proposées dans cette même partie sera donnée en fonction des possibilités offertes par cette technologie.

4.8.1. Critères informationnels.

4.8.1.1. Structure des données.

Le *HTML* permet de gérer des fichiers sans structure particulière. Il est donc idéal pour tout document multimédia avec du texte et des images voire du son, de la vidéo, etc.

Cette technologie n'offre aucune possibilité de gestion de structures hiérarchiques ni de bases de données.

Valeur idéale: Inexistante (I).

4.8.1.2. Volume.

Un documents *HTML* contient uniquement du code *ASCII*, c'est-à-dire uniquement des caractères sans fioritures de mise en page comme dans un fichier "Word" par exemple. Dès lors, ces fichiers prennent peu de place et leur volume est donc directement proportionnel à la quantité d'informations qu'ils contiennent (moins les balises). Toutefois, le but étant de les transférer via Internet, il est important que ces documents ne soient pas trop lourds à télécharger. Il ne faut donc pas dépasser les 100 Ko, et plutôt préférer une décomposition en plusieurs fichiers différents (avec utilisation des hyperliens). Le chargement étant le centre du problème, mieux vaut guider l'utilisateur vers ce qui l'intéresse plutôt que d'afficher toutes les informations sur une seule page. En effet, celle-ci contiendra généralement une quantité appréciable d'informations n'intéressant pas notre "surfeur". Il faut bien se dire qu'un site "qui n'avance pas" sera forcément "zappé". Attention également à l'utilisation d'images (ou autre objet multimédia) dans les documents *HTML*. Ce sont des liens externes au texte mais leur affichage dépendra également de leur taille. On privilégiera donc les petites images peu gourmandes en place mémoire (bien choisir son format: "gif" ou "jpg" en général).

Le *HTML* permettra donc la gestion d'une grande quantité d'informations à condition de la diviser en petits fichiers liés entre eux.

Valeur idéale: Petit (P) .

4.8.1.3. Fréquence des changements.

Etant donné que les fichiers *HTML* sont de "simples" documents de texte, leur mise à jour représente encore un travail assez pénible. Toutefois, si le but de ces fichiers consiste uniquement en des ajouts de données ainsi que de la consultation et non pas des suppressions et des mises à jour, l'utilisation du *HTML* peut constituer une solution très intéressante.

Valeur idéale: Faible (F).

4.8.1.4. Sécurité.

Les protections possibles pour des fichiers *HTML* sont assez limitées et dépendent du serveur Web. En effet, les accès sur un fichier seront soit permis pour tout le monde, soit

refusés pour tout le monde. Dès lors, à partir du moment où ce fichier se trouve sur le serveur Web, il est mis à disposition de tous les surfeurs, à moins que ce serveur ne joue le rôle de filtre en demandant une identification de la part de l'internaute. C'est donc du côté du serveur Web qu'il faut se tourner pour d'éventuelles protection d'accès.

Valeur idéale: Accès libre (L).

4.8.2. Système client.

4.8.2.1. Configuration cliente.

La configuration exigée du côté du client est très légère. La seule nécessité réside dans l'utilisation d'un navigateur comme "Netscape Navigator", "Microsoft Internet Explorer", etc.

Valeur idéale: Légère (L).

4.8.2.2. Interface graphique.

Le *HTML* offre des objets graphiques allant des champs de textes aux boutons, en passant par les menus déroulants et autres boutons radio. C'est donc une interface graphique moyenne (voir les formulaires au point 4.4).

Valeurs idéales: Simple (S) ou Moyenne (M).

4.8.2.3. Interactivité.

Le *HTML* permet l'interactivité uniquement au travers des "hyperliens" qui laisse à l'utilisateur le choix de sa navigation. Les formulaires seuls ne constituent pas une possibilité d'interactivité puisque la liaison avec le serveur doit se faire par une technologie "Server-Side" ou "Client-Side" (voir chapitres suivants).

Par contre, l'utilisation du "JavaScript" permet une certaine "réactivité" du site puisqu'il peut réagir à des événements liés à l'utilisateur. Mais cette technique ne permet pas non plus la gestion interactive des formulaires.

Valeur idéale: Faible (F).

4.8.3. Système serveur.

4.8.3.1. Configuration serveur(s).

Le *HTML* ne requiert qu'un serveur Web ordinaire.

Valeur idéale: Légère (L).

4.8.3.2. Distribution des applications.

Le *HTML* est une technologie purement "Client-Side", le code étant interprété entièrement par le navigateur.

Valeur idéale: Client (C).

4.8.3.3. Distribution des BD.

Le *HTML* seul ne permet pas la gestion des bases de données. Il n'est dès lors pas question de distribution des *BD*.

Valeur idéale: Inexistante (I).

4.8.4. Critère de conception.

4.8.4.1. Compétences locales.

Le *HTML* se veut expressément simple pour les développeurs. La création de page *HTML* ne pose donc pas de problèmes d'autant plus que de nombreux éditeurs permettent à des non-informaticiens de pouvoir se débrouiller. Créer une page Web est aussi simple que de se servir d'un traitement de texte. D'ailleurs, "Word" permet la conversion de ses fichiers ".doc" en fichiers ".html".

Valeurs idéales: Faible (F), Moyenne (M) ou Elevée (E).

4.9. Bibliographie.

[5] Neil Randall, "*J'utilise HTML*", Simon & Schuster Macmillan, Paris, France, 1996.

[6] David Flanagan, "*JavaScript, The Definitive Guide - Third Edition*", O'Reilly & Associates, United States of America, 1998.

Sites "*HTML*" et "JavaScript".

{3} "Un Nouveau Guide Internet" – Pour utilisateurs et concepteurs: http://ungi.cge.net/cyber/ (chap. 41 à 46: "JavaScript")

{4} Source programmes "JavaScript": http://www.developer.com

{5} Source programmes "JavaScript": http://www.essex1.com/people/timothy/js-index.htm

{6} Source programmes "JavaScript": http://www.dynamicdrive.com

{7} Spécifications officielles des "cookies *HTTP*": http://www.netscape.com/newsref/std/cookie_spec.html

Chapitre 5. DHTML.

5.1. *HTML* 4.

Le langage *HTML* évolue et l'on en est maintenant à la version 4 nommée *DHTML* pour "Dynamic *HTML*". Dans cette nouvelle version apparaissent quelques changements radicaux au niveau de la dynamique des pages. En effet, on peut maintenant paramètrer n'importe quelle balise afin de pouvoir la gérer à volonté, sans devoir recharger la page, en utilisant un programme "JavaScript" ou "JScript" (4.6). Le gain offert par cette version de *HTML* est donc principalement limité à l'aspect visuel des pages: il est maintenant possible d'imaginer de petites animations sans devoir créer pour cela des images animées souvent lourdes en place mémoire.

Malheureusement, les implémentations de cette nouvelle technologie dans "Netscape Navigator" et "Microsoft Internet Explorer" sont en grande partie incompatibles: les fonctions "JavaScript" de gestion dynamique des balises sont pour la plupart différentes d'un browser à l'autre.

5.2. DHTML et Bases de Données.

Chez "Microsoft", en plus des éléments cités ci-dessus, le *DHTML* offre des possibilités de connexion à des données à l'aide d'objets inclus dans la page *HTML*. Nous allons maintenant étudier plus particulièrement cet aspect du *DHTML*. Signalons directement que cette solution étant liée au *DHTML* de "Microsoft", seuls les navigateurs "Internet Explorer" conviennent.

5.2.1. "Data Binding".

Le principe poursuivit par "Microsoft" se base sur le fait qu'un document est constitué de deux composants bien distincts: la structure et les données. Bien souvent, nous désirons créer des documents similaires par leur structure mais différents par leur contenu. C'est le cas par exemple d'une liste d'informations à propos de cassettes vidéo: chaque cassette est représentée par un titre, une durée, une liste d'acteurs, etc. En "simple" *HTML*, les données seraient entrées les unes à la suite des autres, ce qui constituerait un fichier assez lourd à modifier: une modification de la structure entraînerait une mise à jour de tous les champs correspondants tandis qu'une modification du contenu nous obligerait à rechercher la donnée de manière fort peu aisée.

Un deuxième aspect pris en compte par Microsoft réside dans le fait que les solutions du type *CGI* (voir plus loin) sont "Server-Side", ce qui signifie que chaque requête est envoyée au serveur pour une analyse puis un renvoi des données correspondantes vers le

client et cela même si la requête précédente désirait les mêmes informations mais par ordre alphabétique au lieu d'un ordre chronologique par exemple.

Le "Data Binding" a donc été ajouté au *DHTML* pour répondre aux remarques précitées.

5.2.2. Exemple.

Voici le code d'un document *DHTML* utilisant le "Data Binding" et que nous allons détailler par la suite:

```
<HTML>
<HEAD>

<OBJECT ID="Content" WIDTH=0 HEIGHT=0
CLASSID="CLSID:333C7BC4-460F-11D0-BC04-0080C7055A83">
        <PARAM NAME="TextQualifier" VALUE="%">
        <PARAM NAME="FieldDelim" VALUE="|">
        <PARAM NAME="DataUrl" VALUE="demo.txt">
        <PARAM NAME="UseHeader" VALUE="true">
</OBJECT>

<SCRIPT language=JSCRIPT>
var rs

function goForward()
{
        if (rs.absolutePosition != rs.recordCount)
        {
            rs.MoveNext() ;
        }
        else
        {
            rs.MoveFirst() ;
        }
}

function initialize()
{
        rs = Content.recordset ;
}

window.onload = initialize() ;
</SCRIPT>
</HEAD>

<BODY>

<SPAN onclick="goForward()">Next Story</SPAN>
<BR>
<DIV ID=TheStory datasrc=#Content datafld="Story" dataformatas="HTML"></DIV>

</BODY>
</HTML>
```

Exemple 22: Data Binding.

On peut observer 3 parties importantes sur cette page:

- L'"**objet source des données**" ou "**Data Source Object**" qui est, dans ce cas, un "Tabular Data Control" (TDC) dont nous dirons un mot plus tard (voir 5.2.4). Le but de cet objet est d'aller chercher l'information pour la mettre à la disposition du Navigateur. Sa déclaration dans le fichier exemple est la suivante:

```
...
<OBJECT ID="Content" WIDTH=0 HEIGHT=0
CLASSID="CLSID:333C7BC4-460F-11D0-BC04-0080C7055A83">
        <PARAM NAME="TextQualifier" VALUE="%">
        <PARAM NAME="FieldDelim" VALUE="|">
        <PARAM NAME="DataUrl" VALUE="demo.txt">
        <PARAM NAME="UseHeader" VALUE="true">
</OBJECT>
...
```

Exemple 23: Déclaration du Data Source Object.

- L'"**élément lié**" aux données correspond à l'endroit dans la page où doivent apparaître ces données. Pour cela, il faut définir 3 nouveaux attributs qui sont les suivants:

  datasrc: l'identifiant du "Data Source Object" (précédé par un #);

  datafld: la colonne parmi l'ensemble des données du "Data Source Object";

  dataformatas: indique si l'information doit être considérée en tant que *HTML* ou en tant que simple texte.

```
...
<DIV ID=TheStory datasrc=#Content datafld="Story" dataformatas="HTML"></DIV>
...
```

Exemple 24: "Elément lié" à un "Data Source Object".

- Le "**script de contrôle**" permet de contrôler quel enregistrement ("record") est sélectionné parmi l'ensemble de "records" fournit par le "Data Source Object". Une fois que les "records" sont accessibles via le "Data Source Object" (rs.recordset dans la fonction "initialize()"), la sélection peut être incrémentée ("MoveNext()"), décrémentée ("MovePrevious()"), mise au début ("MoveFirst()") ou à la fin ("MoveLast()") de l'ensemble des "records". Simplement en changeant l'enregistrement courant, la nouvelle information est affichée sur la page.

```
...
<SCRIPT language=JSCRIPT>
var rs

function goForward()
{
        if (rs.absolutePosition != rs.recordCount)
        {
            rs.MoveNext() ;
        }
        else
        {
            rs.MoveFirst() ;
        }
}

function initialize()
```

```
{
        rs = Content.recordset ;
}

window.onload = initialize() ;
</SCRIPT>
...
```

Exemple 25: Script de gestion du "Data Source Object".

Le fichier de données utilisé pour cet exemple pourrait être le suivant:

```
%Story%
%Ceci est la première histoire. Il contient du <B>HTML</B>%
%Ceci est la <I>deuxième histoire</I> qui contient une image: <IMG SRC="Logo.gif"%
%Tout ceci est du <INPUT TYPE=Button VALUE="Dynamic HTML"
        onclick="alert('Dynamic HTML')">%
```

Exemple 26: Fichier utilisé par un TDC.

Comme on peut le voir, la page d'affichage en *DHTML* ne devra plus être changée si une nouvelle histoire est créée: seul le fichier de données devra être modifié. De plus, on notera que l'image de la deuxième histoire sera téléchargée uniquement si l'utilisateur sélectionne cette histoire, ce qui constitue un gain de place et de rapidité.

5.2.3. Architecture.

Le "Data Binding" de Microsoft se base sur une architecture constituée de 4 pièces principales (voir Figure 8):

- les "Data Source Objects" (*DSO*) qui fournissent les données à la page *HTML*;
- les "Data Consumers" ou "consommateurs" des données chargés de l'affichage de ces données;
- l'agent de liaison ("Binding Agent") qui assure la synchronisation entre les 2;
- l'agent de répétion de table ("Table Repetition Agent") qui est aussi une agent de liaison mais spécifique aux données affichées sous forme de tableau.

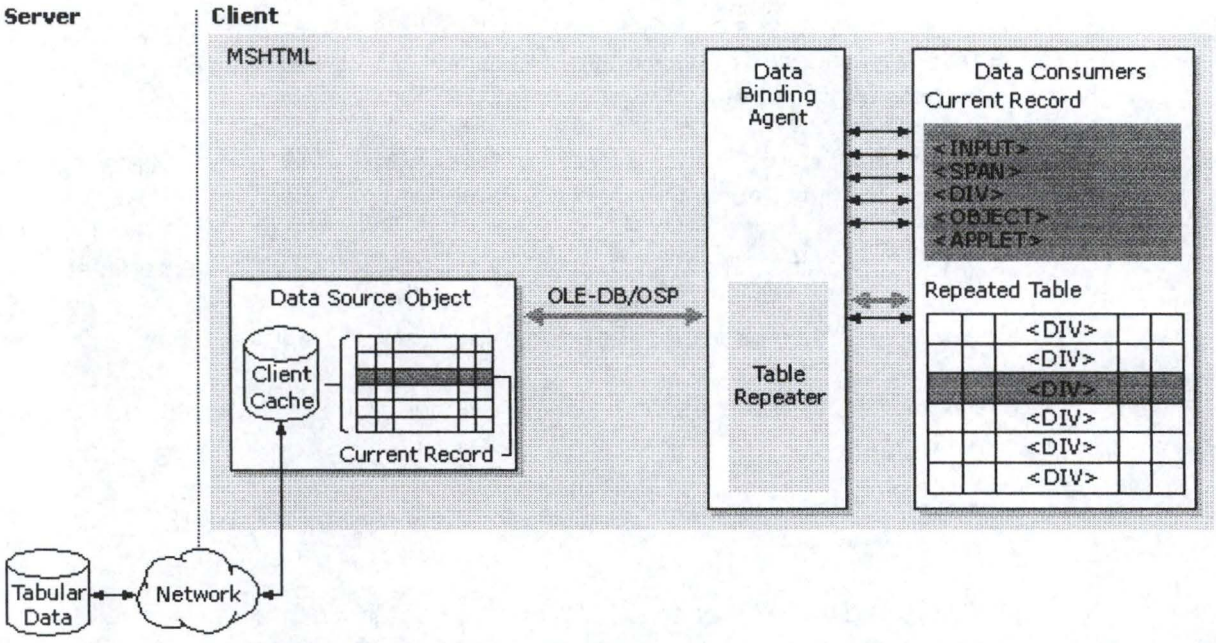

Figure 8: Architecture du "Data Binding" (*DHTML*).

5.2.3.1. Data Source Object (DSO).

Un *DSO* détermine les aspects suivants:

- Transmission des données. Un *DSO* peut utiliser n'importe quel protocole de transport tel HTTP ou une simple entrée/sortie sur des fichiers. De plus, la transmission peut se faire de manière synchrone ou asynchrone, ce dernier cas étant meilleur pour l'interactivié vis-à-vis de l'utilisateur puisque les données peuvent être affichées même si tout le document n'a pas été chargé entièrement;

- Spécification des données. Un *DSO* peut nécessiter un "string" de connexion à un *ODBC* ainsi qu'une requête *SQL*, ou bien uniquement une simple adresse Internet (*URL*);

- Manipulation des données via les scripts. Puisque le *DSO* garde les données sur la machine cliente, il doit aussi gérer la manière dont ces données sont filtrées;

- Modification des données. Il est à noter que si les données peuvent être modifiées localement, cela ne signifie pas que ces changements interviennent au niveau du serveur. C'est au *DSO* de fixer la possibilité éventuelle d'une mise à jour des données réelles.

La seule contrainte pour un *DSO* est qu'il doit offrir l'accès aux données au travers des interfaces (*API*s) "OLE DB" ou "OLE DB Simple Provider" (interfaces "Microsoft"), ce qui permet l'utilisation de ce "DSO" via Internet Explorer qui est agrémenté d'un nouvel ensemble de propriétés, méthodes et événements permettant de gérer cette nouvelle technologie.

5.2.3.2. Data Consumers.

Afin d'afficher les données fournies par un *DSO* sur une page, on peut utilisé des éléments *HTML* mais aussi d'autres objets implémentés, comme des applets "Java" (voir Chapitre 7) ou des "contrôles ActiveX" (technologie "Microsoft").

De plus, "Internet Explorer" supporte des extension *HTML* (arguments de balises) permettant de lier un marqueur à une colonne de données spécifique:

- DATASRC: spécifie l'identification du *DSO* auquel le "Data Consumer" est lié;
- DATAFLD: identifie la colonne spécifique auquel l'élément est lié;
- DATAFORMATAS: indique comment la donnée doit être affichée (simple texte ou *HTML*);
- DATAPAGESIZE: indique combien de "records" doivent être affichés à la fois dans le cas des tables.

On trouve 2 types de "Data Consumers", ceux qui sont liés à une seules données à la fois ("single-valued") et ceux qui affichent plusieurs données du même type à la fois ("tabular data"). Pour ce dernier cas, on peut citer la balise <TABLE>.

5.2.3.3. Binding Agent.

Lors du chargement d'une page, "l'agent de liaison" doit situer les *DSO*s et les "data consumers" pour pouvoir maintenir la synchronisation entre les deux: quand le *DSO* reçoit des données du serveur, c'est à "l'agent de liaison" de les transmettre au "consumer"; inversement, lors d'une modification au niveau de la page, "l'agent" est chargé de la faire parvenir au *DSO*.

5.2.3.4. Table Repetition Agent.

Le but de cet agent est similaire à celui de l'agent de liaison décrit ci-dessus mais pour les "tabular datas".

5.2.4. Les "Data Source Objects" existants.

Un *DSO* peut être implémenté comme un "contrôle ActiveX" ou comme une applet "Java" (voir Chapitre 7). Pour insérer un "contrôle ActiveX", on utilise la balise *HTML* **<OBJECT>**, tandis que dans le cas d'une applet "Java", c'est la balise **<APPLET>** qui doit être utilisée.

"Internet Explorer 4" supporte les *DSO* suivants:

5.2.4.1. Tabular Data Control (TDC).

Le *TDC* est un simple *DSO* qui permet l'accès à des fichiers textes dans lesquels les données sont séparées par des caractères particuliers comme par exemple "%", ";" ou encore "|" (délimitateurs paramètrables).

L'utilisation d'un *TDC* peut être envisagée si:

- l'ensemble des données à afficher est simple;
- les données doivent être lues hors connexion: le fichier transmis par le *TDC* peut être caché sur le client pour être lu "offline";
- l'accès direct à une base de données veut être évité: un script serveur pourrait être écrit dans le but de traduire une *BD* en un fichier compatible *TDC*. D'ailleurs, la majorité des *SGBD* offre cette possibilité.

On notera donc bien que ce type de *DSO* ne permet que de la consultation de données dans un fichier délimité par un caractère spécial à définir. Il n'est dès lors pas possible de faire des mises à jour, des insertions ou des suppressions.

5.2.4.2. Remote Data Service (RDS).

Le *RDS* est un *DSO* plus sophistiqué que le *TDC*. Il obtient les données à partir d'une *BD* en utilisant "OLE-DB" ou *ODBC*.

On utilisera un *RDS* si:

- les données sont contenues dans un *SGBD* compatible "OLE-DB" ou *ODBC* tel que "SQL Server", "Microsoft Access" ou "Oracle";
- on désire spécifier les données via des requêtes *SQL*;
- la mise à jour, l'insertion et la suppression des données sont recherchées;
- l'accès aux données doit être fait en temps réel.

5.2.4.3. JDBC DataSource Applet.

Ce *DSO* est implémenté en "Java" et il utilise *JDBC* (voir Chapitre 7) pour ses accès aux données.

Cette "applet" sera utilisée si:

- les données sont contenues dans un *SGBD* compatible *ODBC* tel que "SQL Server", "Microsoft Access" ou "Oracle";
- on désire spécifier les données via des requêtes *SQL*;
- la mise à jour, l'insertion et la suppression des données sont recherchées;
- on préfère utiliser *JDBC* pour le transport des données.

Une grande distinction doit être faite ici par rapport au *RDS* présenté précédemment. En effet, "l'applet" *DSO* permet des mises à jour, des insertions et des suppressions mais uniquement au niveau local, les changements n'étant pas transférés jusqu'à la base de

données initiales. Ce *DSO* ne sera donc pas utilisé pour des accès complet à la BD, mais permettra par exemple des simulations locales, sans modification des données serveurs.

5.2.4.4. XML Data Source.

Le *XML* ("eXtensible Markup Language") permet de décrire des données et des textes structurés de manière standard pour le Web. Un chapitre complet lui est consacré par la suite (voir Chapitre 6). Le "XML Data Source" est un *DSO* implémenté en "Java" permettant la gestion de données *XML* en lecture seulement (pas de mise à jour possible, donc). On utilisera donc ce *DSO* pour l'affichage hiérarchique de données contenues dans des fichiers *XML*.

5.2.4.5. MSHTML Data Source.

On peut également définir les données directement dans un fichier *HTML* en spécifiant pour chaque élément un attribut **ID=**. Dans ce cas, l'ensemble des données ayant le même identifiant "ID" seront considérés comme faisant partie d'une même colonne et seront donc mis en correspondance avec le DATAFLD correspondant.

```
<H1 ID=COMPSR_FIRST>Hector</H1>
<MARQUEE ID=COMPSR_LAST>Berlioz</MARQUEE>
<DIV ID=COMPSR_BIRTH>1803</DIV>

<H2 ID=COMPSR_FIRST>Modest</H2>
<H3 ID=COMPSR_LAST>Moussorgsky</H3>
<BUTTON ID=COMPSR_BIRTH>1839</BUTTON>

<TEXTAREA ID=COMPSR_FIRST>Franz</TEXTAREA>
<XMP ID=COMPSR_LAST>Liszt</XMP>
<SPAN ID=COMPSR_BIRTH>1811</SPAN>
```

Exemple 27: MSHTML Data Source.

Une telle solution ne permet pas la mise à jour des informations mais uniquement de la consultation.

5.3. Première évaluation.

Nous allons maintenant décrire les particularités du *DHTML* et du "Data Binding" par rapport aux critères définis précédemment (voir Partie I.Chapitre 2: Critères de conception.). Une évaluation de ces critères en fonction des valeurs proposées dans cette même partie sera donnée en fonction des possibilités offertes par cette technologie.

Pour une utilisation du *DHTML* comme une version du *HTML*, on se rapportera au chapitre précédent. Ici nous analysons le cas de la gestion des données via le "Data Binding".

5.3.1. Critères informationnels.

5.3.1.1. Structure des données.

Comme on l'a vu au chapitre précédent, le *HTML* pur permet de gérer des fichiers sans structure particulière.

Pour ce qui est des documents plus structurés, l'utilisation du "Data Binding" peut être envisagée: pour les données "semi-structurées" (voir 1.2.2), un *TDC* ou du "*MSHTML* Data Source" convient parfaitement. La gestion de fichier *XML* est également possible. Si les données sont contenues dans une base de données "relationnelle", le recours à un *RDS* ou à une "*JDBC* DataSource applet" sera inévitable.

Valeur idéale: Elevée (E).

5.3.1.2. Volume.

Le "Data Binding" permet de gérer des bases de données "relationnelles". La gestion du volume des données dépend donc étroitement du *SGBD* correspondant mais il pourra généralement être très conséquent.

De plus, les *BD* "semi-structurées" peuvent être utilisées mais uniquement en consultation. Etant donné que tout le fichier contenant les données sera chargé sur le disque client, on recommandera cette solution pour de petits volumes.

Valeurs idéales: Petit (P) (*BD* "semi-structurées") ou Gros (G) (*BD* "relationnelles").

5.3.1.3. Fréquence des changements.

L'utilisation de bases de données "relationnelles" permet une mise à jour aisée des informations. Par contre, les *BD* "semi-structurées" demandent une analyse de fichiers peu structurés et donc peu avantageuse pour de nombreuses mises à jour manuelles. On conseillera donc ce type de *BD* uniquement lorsque les mises à jour seront peu fréquentes, qu'elles consisteront essentiellement en des ajouts d'informations ou que la solution choisie permet la mise à jour automatique à partir de l'application.

Valeurs idéales: Faible (F) (*BD* "semi-structurées"), Moyenne (M) (*BD* "relationnelles") ou Elevée (E) (*BD* "relationnelles").

5.3.1.4. Sécurité.

Les protections d'accès à des *BD* "relationnelles" sont du ressort du *SGBD* correspondant. Toutefois, le mot de passe devant être inséré dans le code *HTML*, l'accès ne pourra pas être paramétré en fonction de la personne connectée. Si l'utilisateur a accès au fichier *HTML* contenant le "Data Binding", il aura alors accès à la base de données sous-jacente.

De même, les *BD* "semi-structurées" sont de simples fichiers de texte dont l'accès est géré par le serveur Web.

Valeur idéale: Accès libre (L).

5.3.2. Système client.

5.3.2.1. Configuration cliente.

Le "Data Binding" n'est supporté que par "Microsoft" à partir de la version 4 de son "Internet Explorer". La connexion aux *BD* exige donc que le client utilise ce browser.

Valeur idéale: Légère (L) mais contraignante.

5.3.2.2. Interface graphique.

Voir *HTML* (4.8.2).

Valeurs idéales: Simple (S) ou Moyenne (M).

5.3.2.3. Interactivité.

Le "Data Binding" permet un niveau supérieur d'interactivité par rapport au *HTML* pur puisqu'il permet la consultation et la mise à jour d'informations contenues dans des bases de données.

Valeurs idéales: Faible (F), Moyenne (M) ou Elevée (E).

5.3.3. Système serveur.

5.3.3.1. Configuration serveur(s).

Le "Data Binding" ne requiert qu'un serveur Web ordinaire.

Valeur idéale: Légère (L).

5.3.3.2. Distribution des applications.

Le "Data Binding" est une technologie "Client-Side".

Valeur idéale: Client (C).

5.3.3.3. Distribution des BD.

Avec le "Data Binding", la distribution des *BD* n'est pas possible. Celles-ci doivent donc se trouver sur le système où est installé le serveur Web.

Valeur idéale: Inexistante (I).

5.3.4. Critère de conception.

5.3.4.1. Compétences locales.

Pour ceux qui veulent utiliser le "Data Binding", des compétences "informatiques" seront nécessaires, sans toutefois exiger une grande maîtrise en la matière. D'ailleurs, la solution basée sur le "*MSHTML* Data Source" ne consiste qu'en la création de fichiers *HTML* un peu plus particuliers (ajout d'identificateurs pour chaque élément). De même, le *TDC* offre la possibilité de créer de simples fichiers textes avec des délimitateurs, ce qui ne constitue pas une difficulté majeure.

Certaines solutions exigent la connaissance du *SQL* pour questionner les *BD*. Ce langage est simple pour les requêtes simples mais peut devenir plus complexe pour des requêtes plus exigeantes.

Valeurs idéales: Moyenne (M) ou Elevée (E).

5.4. Bibliographie.

[7] Neil Randall, "*J'utilise HTML*", Simon & Schuster Macmillan, Paris, France, 1996.

Sites "*DHTML*" et "Data Binding".

{8} "Un Nouveau Guide Internet" – Pour utilisateurs et concepteurs: http://ungi.cge.net/cyber/ (chap. 29: *HTML* 4.0)

{9} "Dynamic HTML" – Tout savoir: http://msdn.microsoft.com/developer/sdk/inetsdk/help/dhtml/dhtml.htm

{10} "Dynamic Drive DHTML (Dynamic HTML) Code Library!" – Répertoire de scripts DHTML: http://www.dynamicdrive.com

{11} "Data Binding" – Le *DHTML* et les *BD*: http://msdn.microsoft.com/developer/sdk/inetsdk/help/dhtml/content/data_binding.htm#dhtml_databind

{12} "Remote Data Service" – Microsoft: http://msdn.microsoft.com/developer/sdk/inetsdk/help/rds/gs01.htm

Chapitre 6. SGML-XML.

6.1. Introduction.

Après l'utilisation massive de l'Internet et du *HTML*, les concepteurs se sont tournés vers le standard qui en est l'origine: le *SGML*. Afin que les choses soient claires dès le début, il faut savoir que le *HTML* est une application particulière du *SGML* alors que le *XML* en est une simplification comme nous allons le voir maintenant. Après avoir défini ce qui se cache sous les lettres *SGML* et *XML*, nous verrons les nouvelles possibilités qui nous sont offertes par ces deux technologies.

La standardisation du *SGML* et du *XML* est étudiée par le *W3C*, le "World Wide Web Consortium", une association de sociétés et d'informaticiens s'occupant de la standardisation des technologies qui touchent à l'Internet.

6.2. SGML.

Le *SGML* ou "Standard Generalized Markup Language" est le standard général des "markup language", c'est-à-dire des langages basés sur le concept de "balises" ou "marqueurs" ("tags"). Le principe des "balises" est simple: chaque objet différent est représenté par un "tag" différent. Ainsi, si l'on veut exprimer le fait qu'une phrase est un titre, on pourrait par exemple écrire:

```
<TITRE> Ceci est un titre </TITRE>
```

Exemple 28: Principe des "balises", "marqueurs" ou "tags".

Ce standard permet de créer des **structures** de documents et ainsi de valider des fichiers par rapport à ces structures. Prenons par exemple le cas d'un "article". Afin que tous les articles que l'on va écrire aient la même structure, on peut en définir un schéma tel que celui-ci: un "article" est constitué d'un "titre", d'un "auteur" et d'un "texte". A son tour, le "texte" se décompose en "paragraphes" suivit d'une "conclusion". Les "paragraphes" et la "conclusion" seront des chaînes de caractères. Afin de définir notre "markup language" suivant le standard *SGML*, on devrait donc spécifier un arbre qui permettrait de valider nos futurs "articles":

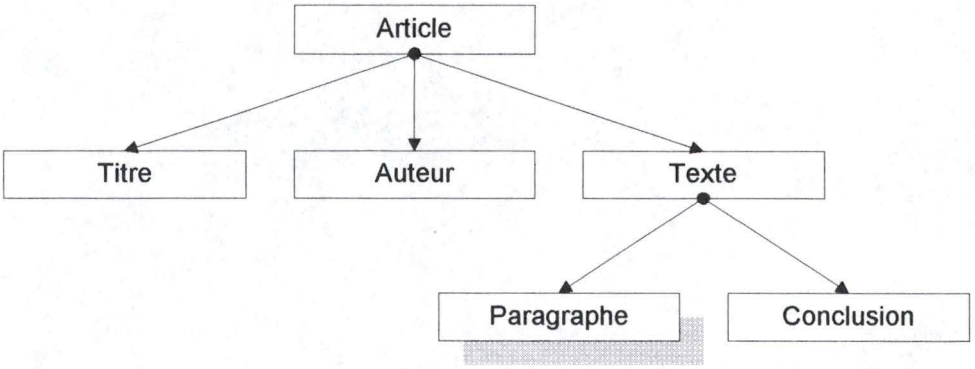

Figure 9: Structure d'un document *SGML*

Une telle structure se déclare dans un *DTD* ou "Document Type Declaration" suivant une syntaxe bien définie comme on peut le voir dans l'exemple qui suit et qui reprend le concept "d'article" défini ci-dessus.

```
...
<!ELEMENT Article          - o      (Titre, Auteur, Texte)>
<!ATTLIST Article Année (#PCDATA) REQUIRED>
<!ELEMENT Titre            - o      (#PCDATA)>
<!ELEMENT Auteur           - o      (#PCDATA)>
<!ELEMENT Texte            - o      (Paragraphe+, Conclusion)>
<!ELEMENT Paragraphe - o            (#PCDATA)>
<!ELEMENT Conclusion  - o           (#PCDATA)>
...
```

où -<!ELEMENT Article - o (Titre, Auteur, Texte)> définit une balise *Article* qui devra contenir un *Titre*, suivit d'un *Auteur*, suivit d'un *Texte*

-<!ATTLIST Article Année (#PCDATA) REQUIRED> spécifie que la balise *Article* contient un attibut obligatoire (*REQUIRED*) nommé *Année*. On écrira donc <Article Année="1998"> par exemple

-#PCDATA spécifie un type "string"

-Paragraphe+ signale que la balise *Paragraphe* peut apparaître plusieurs fois dans le bloc de *Texte*; un **?** signifierait que le champ est facultatif (0 ou 1), tandis qu'une * veut dire O, 1 ou plusieurs

Exemple 29: *DTD* en *SGML*

Une fois la structure bien en place, on peut écrire nos articles en utilisant les "balises" correspondantes.

```
...
<Article Année="1998">
        <Titre> Ceci est le titre de mon article </Titre>
        <Auteur> Olivier Davreux </Auteur>
        <Texte>
                <Paragraphe> Voici le 1er paragraphe </Paragraphe>
                <Paragraphe> Et ici le 2ème paragraphe </Paragraphe>
                <Conclusion> Une petite conclusion pour finir </Conclusion>
        </Texte>
</Article>
...
```

Exemple 30: Document *SGML*

On le voit, le *SGML* permet de définir des structures hiérarchiques très complexes de façon très aisée. L'avantage des documents ainsi créés réside dans la possibilité de faire

des recherches très précises, facilement et sur plusieurs documents à la fois. Nous reparlerons de ces recherches un peu plus loin dans ce chapitre.

Comme exprimé dans l'introduction, l'exemple le plus connu d'application *SGML*, est le *HTML*. Ce langage contient en effet un certain nombre de "balises" définissant une structure de document qu'il faut respecter. On peut dire que le *DTD* est défini dans les navigateurs qui peuvent ainsi valider nos documents *HTML* et les afficher à l'écran. Remarquons ici que cet affichage doit également être défini pour chaque élément contenu dans le *DTD*. Dans le cas du *HTML*, les "tags" sont interprétés par le navigateur de manière plus ou moins standard (il existe en effet quelques différences comme par exemple entre "Netscape Navigator" et "Internet Explorer"; toutefois, les apparences restent assez semblables). Par contre, puisque le *SGML* nous laisse la liberté de création de nos propres "balises", il est évident que c'est à nous également de définir leur apparence à l'écran. Ceci peut se faire grâce au principe des "feuilles de styles" ("stylesheets") qui sont en cours de standardisation pour le *SGML*.

6.3. XML.

Le *XML* est l'abréviation de "eXtensible Markup Language" et est un sous-ensemble du *SGML*. En effet, les concepts définis dans le *SGML* ont été quelque peu simplifiés pour permettre un meilleur accès à cette technologie.

En ce qui concerne les *DTD*, les simplifications résident essentiellement en la suppression d'éléments de la syntaxe ainsi qu'en son caractère facultatif: en effet, un document *XML* peut exister indépendamment d'un *DTD*. Dans ce dernier cas, le document sera considéré comme étant "bien formé" si ses "balises" sont utilisées "proprement" ("balises" d'ouverture et de fermeture; exemples: <Adresse></Adresse>, <Titre></Titre>).

Pour ce qui est du "design" abordé un peu plus haut, il existe plusieurs solutions pour la création des "feuilles de style", comme le *DSSSL* ("Document Style Semantics and Specification Language"), les *CSS* ("Cascading StyleSheets"), le *XSL* ("eXtensible Style Language") et bien d'autres, mais ces concepts en sont encore principalement à l'état de standardisation.

Le *XML* est donc un intermédiaire entre l'inflexibilité du *HTML* et la complexité du *SGML*. Les développements technologiques se tournent donc très fortement vers le *XML*, ce qui en fait dès à présent le "markup (meta-)language" de demain.

6.4. Utilisation du *SGML* et du *XML*.

Un des avantages du *XML/SGML* réside dans le fait que la structure est séparée des données contenues dans les documents. Dès lors, l'affichage et l'impression de ces documents peuvent être développés indépendamment et même de différentes façons en fonction de la personne à qui on les destine.

Toutefois, le point important qu'il faut bien voir, c'est la possibilité de faire des recherches sur ces documents de manière assez variées. On pourrait par exemple se demander combien "d'articles" parlent du bogue de l'an 2000 dans leur "conclusion". A cette fin, beaucoup d'efforts sont fournis par les concepteurs dans le but de créer un langage de requête comme il en existe pour les bases de données. Notamment, à ce

propos, on peut citer le *XQL* ("*XML* Query Language") que nous analyserons plus loin dans ce chapitre (voir 6.5).

D'autre part, la conception d'une nouvelle application, c'est-à-dire d'un nouveau langage, n'est pas chose aisée et demande des compétences informatiques plus évoluées que celles exigées par l'utilisation du *HTML*. La difficulté principale réside bien évidemment dans la création d'un *DTD* qui soit complet, performant et conforme au modèle que l'on s'est fixé. C'est pourquoi on voit dès à présent se développer des langages dans des domaines assez variés comme la médecine, les mathématiques ou les finances. Une fois ces applications bien au point, le non-informaticien pourra sélectionner celle qui cadre le mieux avec ses objectifs, y apporter quelques (légères) modifications si nécessaire, et l'utiliser à sa guise dans le même esprit d'utilisation que le *HTML*.

6.5. XML et bases de données.

Considérons un document reprenant des informations générales sur des personnes. En *XML*, cela pourrait s'écrire:

```
...
<Personne>
        <Nom>Davreux</Nom>
        <Adresse>Wierde</Adresse>
        <Téléphone>081/00.00.00</Téléphone>
</Personne>
<Personne>
        <Nom>Dessy</Nom>
        <Adresse>Malonne</Adresse>
        <Téléphone>081/11.11.11</Téléphone>
</Personne>
...
```

Exemple 31: Document XML de personnes.

On observe sur cet exemple qu'un tel fichier *XML* ressemble assez fort aux bases de données classiques. De plus, les questions qui se posent quant à l'utilisation de tels documents font également penser aux problèmes soulevés par ces bases de données:

- comment extraire des données contenues dans des documents *XML* ?
- comment intégrer des données provenants de sources *XML* différentes ?

Les réponses à ces questions sont fournies par le *XML-QL* ("*XML* Query Language") qui considère un document *XML* en tant que base de données et son *DTD* correspondant en tant que schéma de la base de données. La base du *XML-QL* repose sur un langage de requête inspiré du *SQL* utilisé en base de données "relationnelles".

6.5.1. Fonctionnalités du XML-QL.

Une simple requête en *XML-QL* ressemble à ceci:

```
WHERE <book>
            <publisher><name> Addison-Wesley </name></publisher>
            <title> $t </title>
```

```
            <author> $a </author>
        </book> IN "www.a.b.c.d/bib.xml"
CONSTRUCT $a
```

ou sous forme simplifiée:

```
WHERE  <book>
            <publisher><name> Addison-Wesley </></>
            <title> $t </>
            <author> $a </>
       </> IN "www.a.b.c.d/bib.xml"
CONSTRUCT $a
```

ce qui fournit la liste des auteurs (<author>) de livres (<book>) dont le nom (<name>) de l'éditeur (<publisher>) est "Addison-Wesley". La recherche s'effectue sur le fichier *XML* se trouvant à l'adresse "www.a.b.c.d/bib.xml".

Remarque: le signe "$" est utilisé pour définir des paramètres.

Exemple 32: Requête *XML-QL*

Comme on le voit, le principe est similaire à celui du *SQL* mais en utilisant les "balises" *XML* correspondantes.

Bien entendu, les fonctionnalités offertes sont plus vastes que celles montrées dans l'exemple. Nous allons maintenant énumérer ces possibilités sans entrer dans les détails, le lecteur intéressé par plus d'informations à ce sujet pouvant se rapporter à l'article du *W3C* référencé dans la bibliographie de ce chapitre ("XML-QL: A Query Language for XML": http://www.w3.org/TR/NOTE-xml-ql/).

- Le résultat de la requête peut être décrit sous format *XML*:

```
WHERE  <book>
            <publisher><name> Addison-Wesley </></>
            <title> $t </>
            <author> $a </>
       </> IN "www.a.b.c.d/bib.xml"
CONSTRUCT
       <result>
            <author> $a </>
            <title> $t </>
       </>
```

Exemple 33: Création d'un document *XML* à partir d'une requête *XML-QL*

- Il est également possible de faire des requêtes imbriquées:

```
WHERE  <book>
            <title> $t </>
            <publisher><name> Addison-Wesley </></>
       </> CONTENT_AS $p IN "www.a.b.c.d/bib.xml"
CONSTRUCT
       <result>
            <title> $t </>
            WHERE <author> $a </> IN $p
            CONSTRUCT <author> $a </>
       </>
```

Le résultat de la première requête est sauvée sous le nom de variable **$p** puis réutilisé pour la deuxième requête

Exemple 34: Requêtes imbriquées en *XML-QL*

- Si des objets ont certains champs en commun, on peut joindre ces éléments:

```
WHERE <article>
          <author>
              <firstname> $f </>
              <lastname> $l </>
          </>
      </> CONTENT_AS $a IN "www.a.b.c.d/bib.xml"

      <book year=$y>
          <author>
              <firstname> $f </>
              <lastname> $l </>
          </>
      </> IN "www.a.b.c.d/bib.xml"

      $y > 1995
CONSTRUCT <article> $a </>
```

En spécifiant un même nom de variable pour des champs provenant de sources différentes (*article* et *author*), on obtient les *articles* dont l'auteur (*author*) a également écrit des bouquins (*book*): c'est une **jointure**

Exemple 35: JOINTURE en *XML-QL*

- Pour supporter le partage de données entre plusieurs éléments, le *XML* possède un attribut de type *ID* ("Identifiant") qui permet d'associer une "clé unique" à un élément. L'utilisation de cette clé en *XML-QL* n'est rien d'autre qu'un paramétrage d'attribut de balise.

DTD

```
...
<!ATTLIST person ident ID #REQUIRED>
<!ATTLIST article author IDREFS #IMPLIED>
...
```

Document *XML*

```
...
<person ID="o123">
      <firstname>John</firstname>
      <lastname>Smith</lastname>
</person>
<person ID="o124">
      <firstname>Grand</firstname>
      <lastname>Schtroumph</lastname>
</person>
...
<article author="o123 o124">
...
</article>
...
```

L'article a pour auteurs (*author*) les personnes (*person*) identifiées "o123" et "o124"

XML-QL

```
WHERE <article author=$i>
          <title$t</>
      </>
```

```
        <person ID=$i>
            <lastname>$l</>
        </>
CONSTRUCT <result> $t $l </>
```

Exemple 36: Clés de référence en *XML* et utilisation en *XML-QL*

- De même, les "balises" peuvent être paramètrées ("tag variables"):

```
WHERE <$p>
            <title> $t </>
            <year> 1995 </>
            <$e> Smith </>
        </> IN "www.a.b.c.d/bib.xml"
        $e IN (author, editor)
CONSTRUCT
        <$p>
            <title> $t </title>
            <$e> Smith </>
        </>

Cherche toute publication (book ou article) publiée en 1995 et pour laquelle Smith est soit
    l'auteur (author), soit l'éditeur (editor)
```

Exemple 37: "Tag variables" en *XML-QL*

- Transformation de données *XML*. Afin de traduire des données d'un *DTD* vers un autre *DTD*, le langage utilise des identifiants d'objets (*OID's*) et des fonctions de Skolem.

```
A partir des données sur les livres et les publications, on voudrait reprendre certains
    éléments pour correspondre au DTD suivant:
    <!ELEMENT person (lastname, firstname, adress?, phone?, publicationtitle*>

on utilise alors une requête du genre de celle-ci:

WHERE <$>
            <author>
                <firstname> $fn </>
                <lastname> $ln </>
            </>
            <title> $t </>
        </> IN "www.a.b.c.d/bib.xml"
CONSTRUCT
        <person ID=PersonID($fn, $ln)>
            <firstname> $fn </>
            <lastname> $ln </>
            <publicationtitle> $t </>
        </>

Chaque fois qu'une personne (<person>) est produite, un OID lui est associé:
    PersonID($fn, $ln). PersonnID est une fonction de Skolem dont le but est de
    générer un nouvel identifiant pour chaque couple distinct de ($fn, $ln). Plus tard, si
    une autre instance présente le même couple (c'est donc une publication
    (<publicationtitle>) différente pour un auteur (<author>) déjà enregistré), alors la
    requête ne crée pas une nouvelle entité de <person> mais concatène la nouvelle
    information à celle existant.
```

Exemple 38: Transformation de données *XML*

- Intégration de données provenants de sources *XML* différentes. Les requêtes ne sont pas limitées à une seule source à la fois:

On peut par exemple constituer un fichier *XML* avec les données contenues dans deux sources différentes ayant un type d'élément en commun:

```
WHERE <person>
            <name></> CONTENT_AS $n
            <ssn> $ssn </>
      </> IN "www.a.b.c.d/data.xml"

      <taxpayer>
            <ssn> $ssn </>
            <income></> CONTENT_AS $i
      </> IN "www.irs.gov/taxpayers.xml"
CONSTRUCT <result> $n $i </>
```

Dans cet exemple, le résultat sera constitué uniquement des *person* qui sont des *taxpayer*. Par contre, si l'on veut obtenir en sortie toutes les informations contenues dans les deux sources, il faut recourir au concept de foncion de Skolem développé précédemment:

```
{ WHERE
      <person>
            <name></> CONTENT_AS $n
            <ssn> $ssn </>
      </> IN "www.a.b.c.d/data.xml"
  CONSTRUCT <result ID=SSNID($ssn)> $n </>
}
{ WHERE
      <taxpayer>
            <ssn> $ssn </>
            <income></> CONTENT_AS $i
      </> IN "www.irs.gov/taxpayers.xml"
  CONSTRUCT <result ID=SSNID($ssn)> $i </>
}
```

Exemple 39: Intégration de données *XML*

6.6. Première évaluation.

Comme on l'a vu, le *XML* peut être utilisé de deux manières.

D'abord, le *XML* constitue une évolution du *HTML* permettant de définir notre propre structure de texte. On peut donc utiliser le *XML* pour définir son propre *HTML* et écrire ainsi des documents assez similaires (avec une structure très simplifiée). Les avantages et les inconvénients seront donc assez semblables à ceux énumérer pour le *HTML* (voir 4.8). Un point positif supplémentaire sera l'utilisation d'un langage de requête tel que *XML-QL*.

D'un autre côté, le *XML* permet de créer des fichiers de données qualifiés de bases de données "semi-structurées". A l'aide d'un langage de requête, il est possible d'imiter le comportement général d'une base de données habituelle mais de manière somme toute simplifiée et de façon moins performante puisqu'il n'existe pas de technologie d'indexage complexe en *XML*.

Nous allons maintenant décrire les particularités du *XML* et du *XML-QL* par rapport aux critères définis précédemment (voir Partie I.Chapitre 2: Critères de conception.). Une évaluation de ces critères en fonction des valeurs proposées dans cette même partie sera donnée en fonction des possibilités offertes par cette technologie.

6.6.1. Critères informationnels.

6.6.1.1. Structure des données.

Le *XML* permet de définir des structures simples (hiérarchie de titres par exemple) ou plus complexes (décomposition de champs en sous-champs, eux-mêmes décomposés en sous-champs, etc.). Si les données sont peu structurées (structure hiérarchique simple), on pourra utiliser le *XML* comme on le fait avec le *HTML*. Toutefois, on ne tirera aucun avantage bien réel d'une telle utilisation du *XML*. Si par contre les données peuvent être décomposées en tables de champs simples, on pourra utiliser le *XML* comme une base de données "semi-structurée".

Valeurs idéales: Hiérarchique (H) ou Elevée (E).

6.6.1.2. Volume.

Pour l'utilisation du *XML* de façon similaire au *HTML*, on se référera à 4.8.1.

Si on utilise le *XML* en tant que *BD*, étant donné que la recherche sur des fichiers *XML* s'effectue sans l'aide d'index, le volume des données ne doit pas être trop conséquent. Nous dirons donc que le *XML* peut convenir pour des petits volumes d'autant que tout le fichier doit être chargé sur la station cliente.

Valeur idéale: Petit (P).

6.6.1.3. Fréquence des changements.

Etant donné que les fichiers *XML* sont de "simples" documents de texte, leur mise à jour représente encore un travail assez pénible. Toutefois, si le but de ces fichiers consiste uniquement en des ajouts de données ainsi que de la consultation et non pas des suppressions et des mises à jour, l'utilisation du *XML* peut constituer une solution très intéressante.

L'utilisation du *XML* en tant que base de données constitue une solution aux fréquences de changement plus élevée. En effet, on peut imaginer que les données soient stockées dans des documents *XML* bien structurés, à la manière des tables dans une *BD* "relationnelle". D'autre part, un document *XML* pourrait intégrer des requêtes en *XML-QL* et ainsi proposé un affichage similaire à un document *HTML* (structure légère). De cette manière, les données et la structure des document finaux sont séparés, ce qui permet une gestion plus adéquate de l'information. Toutefois, seule la consultation est possible à distance. Dès lors, on ne peut utiliser le *XML* pour les mises à jour qui doivent donc se faire manuellement ou via d'autres applications externes.

Les *BD* "semi-structurées" demandent une analyse de fichiers peu structurés et donc peu avantageuse pour de nombreuses mises à jour. On conseillera donc ce type de *BD* uniquement lorsque les mises à jour seront peu fréquentes ou qu'elles consisteront essentiellement en des ajouts d'informations.

Valeurs idéales: Faible (F) ou Moyenne (M).

6.6.1.4. Sécurité.

Les protections possibles pour des fichiers *XML* sont assez limitées et elles correspondent au cas du *HTML* (voir au point 4.8.1).

Valeur idéale: Accès libre (L).

6.6.2. Système client.

6.6.2.1. Configuration cliente.

La configuration exigée du côté du client est très légère. La seule nécessité réside dans l'utilisation d'un navigateur compatible *XML*. Etant donnés les développements technologiques actuels, nous pouvons penser que les principaux navigateurs comme "Netscape Navigator" et "Internet Explorer" intégreront bientôt cette fonction. Restera également à voir comment les langages de requête y seront intégrés.

Valeur idéale: Légère (L).

6.6.2.2. Interface graphique.

Les possibilités d'interfaces offertes par le *XML* sont assez limitées puisque restreintes à du simple texte. On peut toutefois imaginer que les développements futurs permettront l'utilisation de formulaires comme c'est le cas en *HTML*.

Valeur idéale: Simple (S).

6.6.2.3. Interactivité.

Il n'y a pas d'interactivité possible en *XML*, pas même les "hyperliens" (voir *HTML*).

De plus, les requêtes doivent être écrites dans un fichier *XML* par l'utilisateur qui doit donc connaître la source des données qui l'intéresse. Ou alors, ces fichiers doivent être prédéfinis sur le serveurs, ce qui signifie que l'utilisateur n'a pas le choix des paramètres pour ces requêtes.

Valeur idéale: Faible (F).

6.6.3. Système serveur.

6.6.3.1. Configuration serveur(s).

Le *XML* ne requiert qu'un serveur Web ordinaire.

Valeur idéale: Légère (L).

6.6.3.2. Distribution des applications.

Les applications, c'est-à-dire les requêtes, sont interprétées par le navigateur. Le *XML* constitue donc une solution "Client-Side".

Valeur idéale: Client (C).

6.6.3.3. Distribution des BD.

Les *BD* sont des fichiers *XML*. Ceux-ci peuvent se trouver n'importe où derrière un serveur Web.

Valeur idéale: Existante (E).

6.6.4. Critère de conception.

6.6.4.1. Compétences locales.

Le *XML* se veut expressément simple pour les futurs développeurs. La syntaxe du *SGML* a été revue pour pouvoir offrir un (méta-)langage accessible à tous. De plus, comme dans le cas du *HTML*, une multitude d'éditeurs commence déjà à être créé, ce qui ouvrent la porte de la création *XML* à toute personne utilisant un traitement de texte comme Word et compagnie. Attention toutefois, la création de *DTD* complexes peut s'avérer très compliquée !

Valeurs idéales: Moyenne (M) ou Elevée (E).

6.7. Bibliographie.

[8] Peter Flynn, "Understanding SGML and XML tools", Kluwer Academic Publishers, United States of America, 1998.

Sites "*XML-SGML*".

{13} "The Annotated XML Specification" – site du W3C: http://www.xml.com/axml/testaxml.htm

{14} http://www.tetrasys.fr/

{15} "XML: faq" – Forum aux questions: http://www.ucc.ie/xml/

Chapitre 7. JAVA.

7.1. Le langage "Java".

"Java" est un langage de programmation "orienté objets" tout comme le "C++", "Delphi" ou "JavaScript". C'est un produit de la division "JavaSoft" chez "SUN Microsystems Inc.". Bien qu'il partage certaines similarités avec le "C++", le "Java" en est tout de même très différent.

Principalement, "Java" est indépendant de la plate-forme, ce qui signifie qu'un programme tournant sous "Unix" sera également supporté par "Windows" ou "Macintosh" par exemple. Le secret de cette indépendance réside dans le fait qu'un programme "Java" n'est pas compilé en code machine mais en un code intermédiaire appelé "bytecode". La véritable compilation a donc lieu au moment de l'exécution, et est possible grâce à la "Machine Virtuelle Java" ("Java Virtual Machine" ou *JVM*) qui, elle, est dépendante de la plate-forme. C'est cette "machine Virutelle" qui est chargée d'interpréter le "bytecode". Pour cela, chaque concepteur doit écrire la *JVM* qui convient à sa plate-forme. Notons que les principaux navigateurs ("Netscape Communicator" et "Internet Explorer") disposent d'une *JVM* depuis leurs versions 3. En principe, l'interprétation du "bytecode" est un processus plus lent que l'exécution d'un programme compilé dans le langage du processeur. Cependant, dans la plupart des cas, la différence est relativement minime.

"Java" est un langage simple à apprendre mais nécessite bien évidemment des connaissances en informatique bien plus poussées que pour le *HTML* par exemple. En fait, il est beaucoup plus simple que le "C++", mais cette simplicité a un prix: les concepts les plus complexes de "C++" ont été simplement éliminés. Mais cela n'affecte en rien l'efficacité de "Java". en effet, tout ce qui peut être fait en "C++" (sauf certaines erreurs graves de programmation) peut l'être en "Java". Il faut seulement parfois utiliser des outils différents, plus simples à manipuler, et pas forcément moins performants. On dira que la simplicité de "Java" est en fait directement liée à la simplicité du problème à résoudre.

7.2. "Java" et "JavaScript".

Malgré la similarité dans le nom de ces deux langages, "Java" et "JavaScript" (développé en 4.6) sont très différents.

"JavaScript" et "Java" sont tous les deux des langages "orienté objets" (*O.O.*) mais si le "Java" impose le téléchargement de "classes", c'est-à-dire de programmes précompilés, depuis le serveur, "JavaScript" provoque l'exécution de programmes non compilés mais interprétés et contenus dans le corps de la page *HTML*.

La grande différence entre les deux langages réside donc dans le fait que "JavaScript" est interprété là où "Java" est précompilé. Ceci a plusieurs implications:

- le code "Java" est protégé des actes de copie frauduleuse (attention toutefois: la précompilation n'est pas de la compilation dans le sens qu'un fichier compilé ne contient que du code source, illisible sur écran, alors qu'une précompilation ne rend pas le programme illisible à proprement parler; il existe d'ailleurs des programmes capables de vous traduire un fichier "Java" en un fichier texte);

- le code "JavaScript" est moins typé et donc moins robuste (exemple: une variable déclarée dans un type peut être utilisée dans un autre type sans que de message d'erreur n'apparaissent ou alors le navigateur provoque une erreur mais ne dit pas exactement pourquoi);

- "JavaScript" est accessible à des auteurs de pages *HTML* alors que le "Java" est réservé à un public plus professionnel.

7.3. Les "Applets Java".

Un des gros avantages au niveau de l'Internet réside dans les "Applets Java", petits programmes insérés dans la page *HTML* et pouvant s'exécuter "à distance". Pour cela, on utilise la balise **<APPLET>** déterminant le programme "Java" a charger ainsi que d'éventuels paramètres pour ce programme.

```
<APPLET CODE="Mon_Applet.class" WIDTH="100" HEIGHT="25">
<PARAM NAME="maValeur" VALUE="Un superbe slip fluo">
</APPLET>

Une telle "Applet" chargera le programme Mon_Applet.class" et l'affichera dans un cadre
        de 100 pixels en largeur et de 25 pixels en hauteur
Le programme appelé requiert un paramètre nommé maValeur et pour lequel le string
        "Un superbe slip fluo" est donné comme valeur
```

Exemple 40: Balise <APPLET> pour l'insertion "d'Applets Java" dans un fichier *HTML*

Lors de l'exécution d'une "Applet Java", toutes les parties du programme (les "classes") sont chargées sur la machine cliente à l'exception des classes de base fournies par "Java" et qui se trouvent dans la *JVM*.

On notera également que le navigateur de l'utilisateur doit intégrer une console "Java" (une *JVM*) et être configuré de manière à accepter les "Applets Java". En effet, la majorité des navigateurs permettent d'empêcher l'exécution de tels programmes, ceci en raison des problèmes de sécurité que cela peut engendrer: exécuter un programme extérieur sur sa propre machine peut être dangereux si la source du programme n'est pas connue. Toutefois, "Java" est un langage très sécurisé à ce niveau. Par exemple, il n'existe pas de pointeur en "Java", au contraire du "C++" qui les utilise en abondance. L'espace mémoire est donc plus sécurisé. D'autre techniques au sein de la *JVM* sont utilisées pour assurer un langage le plus sûr possible, mais des failles existent et pourraient être utilisées à des fins malintentionnées (détournement de mots de passe, virus, etc.). Nous ne nous attarderons pas plus sur ce problème de sécurité puisque c'est surtout au client de faire attention aux contenus qu'il exécute sur sa machine. On pourra lire le livre de Anup K. Ghosh [15] pour plus d'informations à propos de la sécurité des contenus exécutables sur l'Internet. Sachons toutefois que la gestion de cette sécurité par la *JVM* (côté client, donc) réduit les performances des applications, que ce soit au niveau de la vitesse d'exécution ou au niveau des permissions d'accès au système client (pas d'écriture ou de lecture possible sur un fichier client par exemple).

7.4. Les "Servlets".

Avec les "Applets", "Java" fournit une solution "Client-Side" à la conception de sites Web. Mais "JavaSoft" a également développé une solution "Server-Side" pour "Java": c'est ce qu'on appelle les "Servlets".

7.4.1. Définition.

Une "Servlet" représente une extension au serveur Web, c'est-à-dire un ensemble de classes "Java" pouvant être chargées dynamiquement pour étendre les fonctionnalités du serveur. De manière similaire aux "Applets" détaillées ci-dessus, les "Servlets" utilisent une "Java Virtual Machine" (*JVM*) mais installée sur le serveur cette fois-ci. Dès lors, le code est toujours aussi sûr et "portable" d'une plate-forme à l'autre mais aussi d'un serveur Web à l'autre. Notons à ce propos que la plupart des serveurs Web supportent "Java" et ses "Servlets". Le "Java Web Server" de "Sun", "l'Enterprise Server" de "Netscape" et le "Domino Go Webserver" de "Lotus" intègrent "Java" d'origine tandis que d'autres offrent cette possibilité au travers de "add-ons", petits programmes se greffant au serveur.

7.4.2. Utilisation des "Servlets".

La première utilisation des "Servlets" consiste à créer dynamiquement toute une page Web avec les données fournies via un formulaire ou contenues dans une base de données par exemple. Dans le cas des formulaires, "Java" fournit deux fonctions spécifiques pour la gestion de données provenant d'une méthode POST ou d'une méthode GET (voir 4.4.1). Ici, une "Servlet" est donc un programme "Java" ordinaire.

Si les données dynamiques ne constituent qu'une partie de la page à créer, c'est-à-dire si une grande partie du document à renvoyer est statique, on peut utiliser la technique des *SSI* ("Server-Side Includes - voir 3.4). Dans ce cas, le code "Java" est inséré directement dans le code *HTML* de la page résultat. Au moment de l'appel, le code est exécuté et le résultat est affiché à l'endroit où la "Servlet" est insérée. On utilise pour cela la "balise" **<SERVLET>** qui est assez similaire à la balise <APPLET> vue plus haut dans ce texte. Voici donc à quoi ressemble un document *HTML* contenant une "Servlet" (*MonHorloge*) prenant en argument un pays et renvoyant l'heure locale:

```
<HTML>
<HEAD>
<TITLE>Document avec une "Servlet"</TITLE>
</HEAD>
<BODY>
Bonjour,
Il est exactement :
<SERVLET CODE="MonHorloge">
<PARAM NAME="Pays" VALUE="Belgique">
</SERVLET>
</BODY>
</HTML>
```

Exemple 41: Une "Servlet Java" en mode "Server-Side Include" (*SSI*).

Une dernière possibilité d'utilisation a fait son apparition récemment: les "JavaServer Pages" ou *JSP*, réponse aux *ASP* de "Microsoft" (voir Chapitre 10). La différence avec le cas précédent des *SSI*, c'est que le code "Java" peut être inséré n'importe où au sein des

"balises" *HTML*. Chaque bloc de code "Servlet", appelé "Scriptlet", utilise les balises <% et %>. Lors de l'appel d'une telle page, le serveur Web va automatiquement créer une "Servlet" équivalente, va la compiler en "ByteCode" et va l'exécuter pour renvoyer la page résultat. La deuxième fois qu'un client voudra accéder à cette même page, le serveur ne fera plus qu'exécuter le code compilé et gagnera donc en performance.

En *JSP*, on peut de plus insérer des expressions et des directives dans le code *HTML* même. Ainsi, en utilisant les balises <%= et <%@, la *JVM* considérera l'instruction respectivement comme une expression ou comme une directive.

Imaginons par exemple qu'un formulaire *HTML* propose à l'utilisateur d'entrer son nom dans un champ "Nom_Client" (<INPUT TYPE="text" NAME="Nom_Client">). A ce formulaire, on pourrait faire correspondre le document *JSP* suivant (sachant que *request.getParameter("Nom_Client")* fournit la valeur entrée par le client):

```
<HTML>
<HEAD>
<TITLE>Document avec un JSP</TITLE>
</HEAD>
<BODY>
<%@ Ma_Variable = request.getParameter("Nom_Client") %>
<% if (Ma_Variable == "") %>
<% { %>
        <H1>Bonjour</H1>
        vous n'avez pas entré votre nom dans le fomulaire précédent...
        <BR>
<% } %>
<% else %>
<% { %>
        <H2>Salut <%= Ma_Variable %> </H2>
        merci de votre visite
        <BR>
<% } %>
</BODY>
</HTML>
```

Exemple 42: Une "Servlet Java" en mode "JavaServer Pages" (*JSP*).

En parcourant le fichier, la *JVM* va effectuer les opérations suivantes:

- stocker la valeur fournie par le client dans la variable *Ma_Variable*;
- effectuer le test *if*: si le client n'a pas fournit son nom, la variable *Ma_Variable* contient une chaîne vide ("") et on affiche donc le code *HTML* entre les premières parenthèses ({ et }); dans le cas contraire, la *JVM* passe directement au *else* et affiche donc le *HTML* qui suit

7.4.3. Gestion des accès aux "Servlets".

Le cycle de vie des "Servlets" constitue un gros avantage de cette technologie. En effet, et au contraire des "Scripts *CGI*" (voir Chapitre 8), lors du chargement d'une "Servlet", le serveur crée une et une seule instance de l'application. Dès lors, tout autre connexion à cette "Servlet" se fera sur la même instance. En *CGI*, chaque accès crée sa propre instance de l'application. L'amélioration des performances se fait donc ressentir de trois manières:

- l'espace mémoire nécessaire est moindre;
- la création de la "Servlet" au moment de l'appel est évité si elle a déjà été chargée par un appel précédent. Gain de temps, donc;
- cela permet la "persistance": par exemple, une connexion à une *BD* ne doit pas être ré-initialisée à chaque requête mais peut être utilisée plusieurs fois. Cela permet aussi de gérer plusieurs utilisateurs d'une même "Servlet" (jeux, communication électronique, compteurs, etc.).

7.4.4. Accès aux informations.

Afin de connaître l'environnement dans lequel le programme "Java" tourne, un ensemble d'objets lui est fournit. En fait, toutes "Servlets" peut accéder aux valeurs contenues dans les variables d'environnement aussi appelées "variables d'environnement *CGI*" (voir Chapitre 8), Ainsi, il est possible de connaître le nom du serveur, le software utilisé, le protocole supporté, la méthode (GET/POST) requise par l'utilisateur, etc. ainsi que le "string de requête" ("query string" - voir 3.1.1) bien entendu.

De plus, étant donné que les "Servlets" s'exécute au sein du serveur Web (via la *JVM*), il leur est possible d'accéder à plus d'informations que les "Scripts*CGI*". En effet, l'interaction avec le serveur permet à une "Servlet" de connaître les droits d'accès aux fichiers Web, les différents "alias", etc. c'est-à-dire l'ensemble des entités paramètrables d'un serveur Web.

7.5. JDBC.

7.5.1. Introduction.

"SUN" a bien évidemment pensé à la connexion aux bases de données. Sa solution à ce niveau s'appelle *JDBC* ("Java DataBase Connectivity"). *JDBC* est un ensemble de classes "Java" fournissant une interface standard permettant la connexion aux *BD* en utilisant uniquement "Java".

En utilisant *JDBC*, on peut envoyer facilement des requêtes *SQL* vers n'importe quelle "*BD* relationnelle". Autrement dit, avec *JDBC*, il n'est pas nécessaire d'écrire un programme pour accéder à une *BD* "Sybase", un autre pour accéder une *BD* "Oracle", encore un autre pour une *BD* "Informix", etc. On peut donc écrire un simple programme capable d'envoyer les requêtes *SQL* à la base de données appropriée. Et puisque c'est du "Java", pas besoin de s'occuper de la plate-forme sur laquelle le programme doit tourner. D'où l'adage de "Sun", souvent cité dans le monde "Java": "Write it once, run it anywhere".

JDBC étend les possibilités de "Java". Par exemple, avec "Java" et *JDBC*, une entreprise peut connecter tous ses employés (même si ceux-ci utilisent des plates-formes différentes) à une ou plusieurs *BD* internes via Internet. L'accès à l'information est possible via une interface unique même si les données sont contenues dans des *BD* différentes. De plus, le programme et les données restant sur le serveur, la gestion des mises à jour est automatique: l'application est chargée depuis le serveur, ce qui veut dire que l'utilisateur exécutera toujours la dernière version.

7.5.2. Fonctionnement de JDBC.

Le fonctionnement est très simple et est composé de trois étapes:

- établissement d'une connexion avec une base de données;
- envoi de requêtes *SQL*;
- réception des résultats envoyés par le *SGBD*.

Par exemple, la connexion à une base de données "Interbase" nommée "donnees.gdb" pourrait se faire comme suit:

```
String url = "jdbc:interbase:donnees.gdb" ;
Connection con = DriverManager.getConnection ( url , "SYSDBA" , "masterkey" ) ;

url sert à identifier et à localiser la BD; dans ce cas, on spécifie l'utilisation de
      "donnees.gdb" à l'aide du pilote "Interbase";
"SYSDBA" est le login de connection;
"masterkey" est le mot de passe correspondant au login.
```

Exemple 43: Connexion via *JDBC* (Java).

Une fois la connexion établie, l'objet *con* de type *Connection* peut être utilisé pour envoyer les requêtes et réceptionner les résultats. Pour cela, il suffit de créer un objet spécifique appelé *Statement* et d'exécuter une des méthodes offertes par cet objet:

```
Statement stmt = con.createStatement () ;
ResultSet result = stmt.executeQuery ( "SELECT colonne1 FROM table1") ;

exécution de la requête SQL demandant l'extraction des valeurs contenues dans la
      colonne nommée colonne1 de la table table1.
```

Exemple 44: Requête en *JDBC* (Java).

On utilise alors l'objet *result* de type *ResultSet* pour analyser les résultats renvoyés par le *SGBD*. Cet objet offre un ensemble de méthodes permettant d'associer une zone mémoire à chaque valeur de l'ensemble résultat. Par exemple, si le résultat est de type entier (*int*), on peut utiliser le code suivant pour afficher à l'écran la *colonne1* sélectionnée dans l'exemple précédent:

```
while ( result.next () )
      {
        int i = result.getInt ( "colonne1" ) ;
        System.out.println ( i ) ;
      }

result.next () parcourt l'ensemble des données que contient l'objet result. Lorsque toutes
      les données ont été parcourues, cette méthode renvoie false, ce qui arrête la
      boucle while;
result.getInt ( "colonne1" ) spécifie que la valeur nommée colonne1 est de type entier. on
      peut également utiliser getString pour les chaînes de caractères, getDate pour
      des dates, etc.
```

Exemple 45: Récupération des données en *JDBC* (Java).

On le voit, *JDBC* n'est qu'un intermédiaire entre le programme et le *SGBD*. Son travail consiste uniquement au passage de la requête *SQL* vers le *SGBD* et ensuite en la réception des résultats et sa traduction en objets "Java".

Le *SQL* est le langage standard pour accéder à des "*BD* relationnelles". Seulement, bien que la plupart des *SGBD* utilisent une forme standard de *SQL* pour les fonctionnalités de base, ils ne sont pas nécessairement conforme au standard le plus récent pour les fonctionnalités avancées. Par exemple, toutes les *BD* ne supportent pas les procédures internes ("stored procedure") ou les "jointures externes" ("outer joins"), et celles qui les supportent ne sont pas nécessairement compatibles entre elles. En attendant que le standard reprennent de plus en plus de fonctionnalités, *JDBC* doit supporter le *SQL* tel qu'il est pour le moment. Pour cela, plusieurs optiques ont été suivies.

Premièrement, *JDBC* permet à n'importe quelle chaîne de caractères d'être passée comme requête au *SGBD* correspondant. Cela signifie qu'une application est libre d'utiliser autant de fonctionnalités qu'elle le désire, mais tout en gardant un risque de recevoir une erreur sur certains *SGBD*.

Une deuxième manière de contourner le problème du *SQL*, c'est l'utilisation par *JDBC* d'informations concernant le *SGBD*. *JDBC* offre en effet un moyen de conversation entre le programme et le *SGBD* au sujet de ce dernier. Ainsi, le programme peut connaître, au moment de l'exécution, les fonctionnalités supportées par le *SGBD* et agir en conséquence.

7.5.3. JDBC versus ODBC.

L'interface *ODBC* ("Open DataBase Connectivity") de "Microsoft" est probablement la plus utilisée pour accéder à des bases de données "relationnelles". Elle permet en effet de se connecter à la plupart des *BD*, sur la plupart des plates-formes. Il est tout à fait possible d'utiliser directement *ODBC* à partir de "Java" mais l'utilisation de *JDBC* pour accéder à *ODBC* est une solution bien meilleure, et ce pour plusieurs raisons:

- *ODBC* n'est pas approprié pour une utilisation directe depuis "Java" parce que c'est une interface écrite en "C". Cela implique un certain nombre de failles dans la sécurité, dans l'implémentation, dans la robustesse, ainsi qu'au niveau de la "portabilité" des applications d'un environnement à un autre;

- *ODBC* est difficile à apprendre, au contraire de *JDBC*;

- *JDBC* est nécessaire pour permettre une solution 100% "Java pure". Avec *ODBC*, les drivers (pilotes) *ODBC* doivent être installés manuellement sur chaque machine cliente. Par contre, avec le driver *JDBC* écrit entièrement en "Java", le code *JDBC* est automatiquement portable et sûr pour toute plate-forme.

Pour accéder à l'interface *ODBC* à partir de *JDBC*, on utilise une passerelle: "*JDBC-ODBC* Bridge" (voir ci-dessous).

7.5.4. Architecture de JDBC.

La figure suivante montre l'architecture et le fonctionnement d'une application *JDBC*.

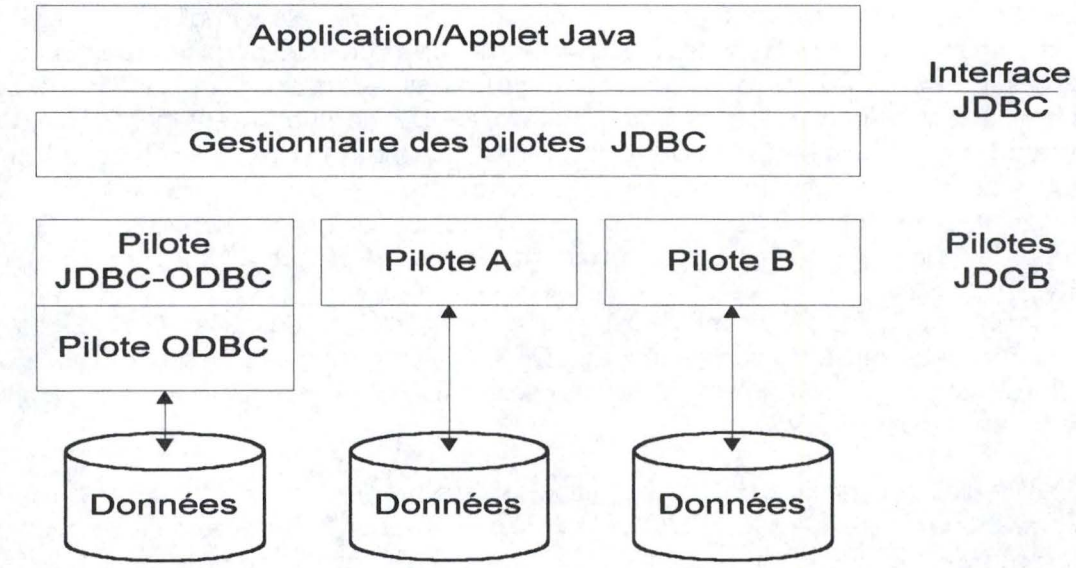

Figure 10: Architecture *JDBC*.

Une "Applet" ou une application "Java" utilise les objets et invoque les méthodes proposées par le gestionnaire de pilotes qui communique avec les différents pilotes spécifiques au *SGBD*. Chacun de ces pilotes spécifiques communique avec la source de données correspondante.

Le gestionnaire de pilotes a comme rôle principal de charger les pilotes *JDBC* correspondant aux sources de données auxquelles l'application souhaite se connecter. Le pilote spécifique interagit avec la base de données: il établit la connexion, soumet les requêtes et renvoie les résultats sous forme d'objets à l'application.

7.5.4.1. Types de pilote JDBC.

Pour pouvoir se connecter à une base de données, un programme "Java" a donc besoin d'un pilote *JDBC*. Celui-ci doit être fournit par le concepteur de la *BD* correspondante à moins qu'il n'existe déjà un pilote *ODBC* pour celle-ci, auquel cas le programme peut utiliser un pilote spécial nommé "*JDBC-ODBC* Bridge" connectant l'application au pilote *ODBC*.

Analysons maintenant les différents types de pilotes *JDBC*.

- Type 1: le pilote "*JDBC-ODBC* Bridge" associé au pilote *ODBC* fournit un accès *JDBC* via un driver *ODBC*. Il est à noter que du code binaire *ODBC* doit être chargé sur chaque machine cliente, ainsi que du code lié à la *BD* dans la plupart des cas. Cette solution sera donc envisageable pour un réseau interne où l'installation de software sur les machines clientes ne pose pas de problème majeur. Elle sera également appropriée pour une application "middle-thier" dans un modèle "à trois étages";

- Type 2: les pilotes utilisant les interfaces directes des bases de données ("native drivers"). Ces pilotes écrits en "Java" appellent directement les fonctions "C" ou "C++" offertes par le *SGBD*. On passe donc par une interface intermédiaire qui est l'interface du *SGBD*. De même que dans le type de pilote précédent, le chargement de parties de software sur la machine cliente est généralement nécessaire;

- Type 3: les pilotes de type 3 fournissent au client une interface générique qui sera ensuite traduite en requêtes pour la *BD* au niveau du serveur. En d'autres mots, le pilote *JDBC* sur le client transmet ses requêtes (standards quelle que soit la *BD* correspondante) à un "middleware" (programme intermédiaire) sur le serveur. Celui-ci les traduit ensuite pour le pilote du *SGBD* concerné. Ce type de driver est très flexible puisqu'il ne requiert aucune installation de code sur la machine cliente et qu'un simple pilote permet des accès à des *BD* multiples;

- Type 4: en utilisant le protocole réseau lié au *SGBD*, les pilotes de type 4 correspondent directement avec la *BD* en utilisant du "Java". C'est donc la solution la plus directe en "Java" pur. Ce type de driver provient généralement du constructeur de la *BD* correspondante: "Sybase", "Borland", "SAS", etc.

Comme on peut déjà le remarquer, les pilotes des catégories 3 et 4 sont à préférer aux catégories 1 et 2 puisqu'ils ne nécessitent aucune installation manuelle sur le client: ce sont des solution 100% "Java" et donc le chargement du pilote est automatique lors du chargement de l'"Applet". Les pilotes de types 1 et 2 seront donc à utiliser lorsque des drivers "purs Java" ne seront pas encore disponibles.

7.5.5. Modèles "Two-Tier" et "Three-Tier".

L'interface *JDBC* supporte deux modèles: le modèle "Two-Tier" (ou "à deux étages") et le modèle "Three-Tier" ("à trois étages").

7.5.5.1. Modèle "Two-Tier".

Dans ce modèle "à deux étages", une "Applet" ou une application "Java" communique directement avec la base de données. Cela nécessite un driver *JDBC* sachant converser avec le *SGBD* correspondant. L'application et le pilote s'exécutent donc sur la machine cliente tandis que le *SGBD* est placé sur un serveur (ou sur le client, mais tel n'est pas le but ici).

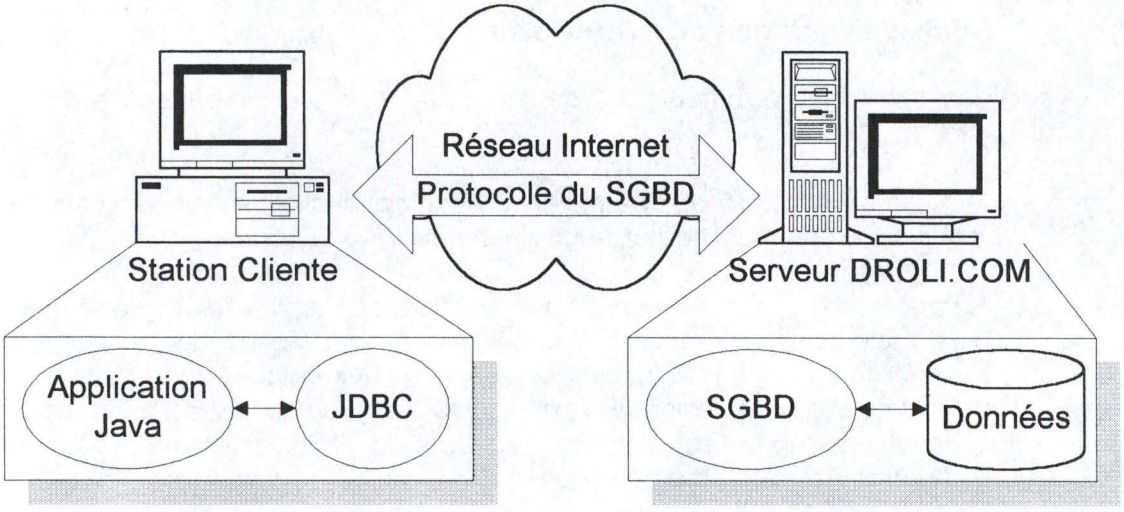

Figure 11: Modèle "Two-Tier" pour *JDBC*.

7.5.5.2. Modèle "Three-Tier".

Dans le modèle "Three-Tier", l'application (ou l'"Applet") ne dialogue plus directement avec le *SGBD* mais les commandes sont envoyées à un "middle tier" qui envoie à son tour les requêtes *SQL* à la base de données. Le *SGBD* exécute alors ces requêtes et renvoie les résultats au "middle tier". Ces résultats sont ensuite communiqués à "l'Applet" sous forme d'appels *HTTP*.

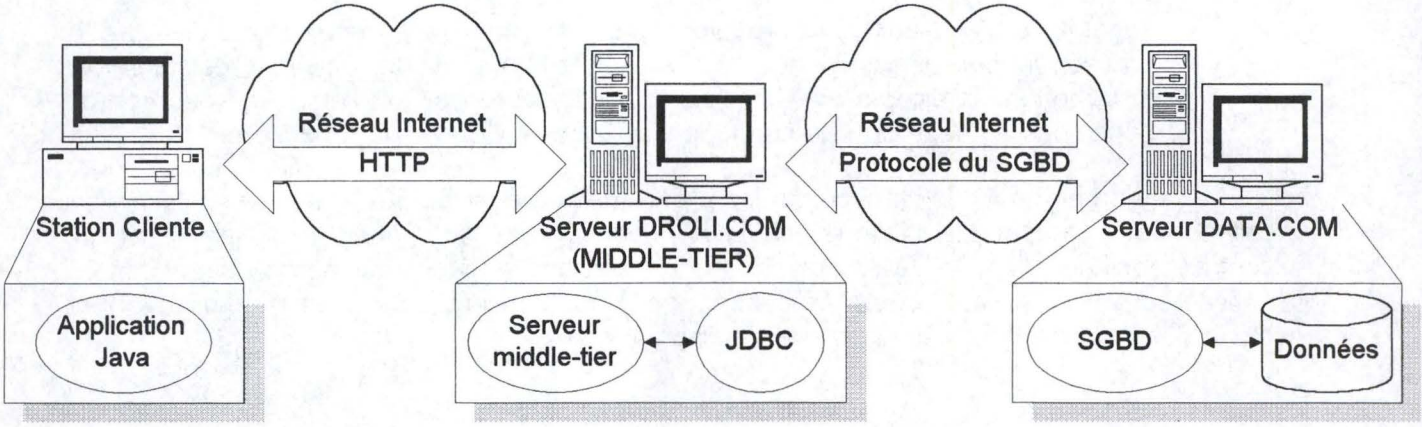

Figure 12: Modèle "Three-Tier" pour *JDBC*.

Bien entendu, le "middle-tier" peut se trouver sur la même machine que le *SGBD*. Ce serveur "middle-tier" est généralement écrit en "C" ou en "C++", lesquels offrent de bonnes performances de vitesse. Toutefois, avec l'introduction de compilateurs traduisant de manière efficace le "bytecode" en code machine, il est devenu bien plus pratique d'implémenter le "middle-tier" en "Java". C'est un grand avantage permettant de profiter de la robustesse et de la sécurité offertes par "Java". De plus, avec l'apparition des "Servlets" (voir 7.4), une solution 100% "Java" devient possible et aisée. On notera également que la présence de *JDBC* est important pour ce modèle aussi afin d'accéder à la *BD* depuis le "middle-tier".

7.5.5.3. Avantages du "Three-Tier".

Parmi les avantages qu'offre la solution du modèle "à trois étages", on peut citer les points suivants:

- le "middle-tier" permet de garder le contrôle des accès et de gérer le type de modification que l'utilisateur peut effectuer sur les données;
- le "middle-tier" peut servir à offrir une interface de plus haut niveau que *JDBC* et plus facile d'utilisation par le programme final: étant donné le type de *BD* connectée au "middle-tier", celui-ci peut offrir des objets et des fonctions de base utilisables par l'application destinée à être exécutée sur le client. On facilite ainsi la création de nouvelles "Applets" et c'est le "middle-tier" qui s'occupe de la traduction des requêtes pour *JDBC*;
- si l'on désire fournir un accès à plusieurs *BD* différentes, le "middle-tier" pourra offrir une interface commune et se charger lui-même de traduire les requêtes

pour le *SGBD* correspondant. De plus, ces *BD* pourront se trouver sur des machines différentes, n'importe où sur le réseau Internet;

- dans beaucoup de cas, le modèle "Three-Tier" offrent des avantages au niveau des performances puisque le "middle-tier" est une machine externe au client (elle appartient à l'environnement du concepteur) et que donc le code peut être optimisé pour cette plate-forme (voir discussion au point précédent sur la compilation du "bytecode").

7.6. Première évaluation.

Nous allons maintenant décrire les particularités du "Java" par rapport aux critères définis précédemment (voir Partie I.Chapitre 2: Critères de conception.). Une évaluation de ces critères en fonction des valeurs proposées dans cette même partie sera donnée en fonction des possibilités offertes par cette technologie.

7.6.1. Critères informationnels.

7.6.1.1. Structure des données.

"Java" est idéal pour insérer un programme ("Applet") dans une page *HTML*. Cette application servira généralement aux endroits nécessitant des calculs en temps réels comme dans le cas d'une horloge, d'un convertisseur en "Euro" ou de toute autre application "Client-Side".

Mais là où "Java" nous intéresse fortement dans le cadre de ce mémoire, c'est au niveau des connexions aux bases de données. Comme on l'a vu dans ce chapitre, *JDBC* permet à une "Applet Java" ou à une "Servlet" de consulter et de mettre à jour des *BD* "relationnelle" par l'utilisation du *SQL*. On dira donc que "Java" est à considérer pour des données à structure élevée.

"Java" permet également la gestion de fichiers et donc de *BD* "semi-structurées".

Valeur idéale: Elevée (E).

7.6.1.2. Volume.

Le volume supporté par une application "Java" dépendra du *SGBD* sous-jacent mais la plupart de ceux-ci permet une gestion de volumes assez conséquents. On se référera donc aux particularités du *SGBD* pour connaître le volume maximum des données.

Valeurs idéales: Petit (P) ou Gros (G).

7.6.1.3. Fréquence des changements.

L'utilisation de *JDBC* est évidemment idéale en ce qui concerne la fréquence des changements des données. En effet, toute modification dans la *BD* se fera ressentir automatiquement au niveau de la page Web. En connectant directement le client à la base de données, on lui assure des informations de dernière minute et la possibilité de les mettre à jour directement.

Valeurs idéales: Faible (F), Moyenne (M) ou Elevée (E).

7.6.1.4. Sécurité.

Une fois l'accès à "l'Applet Java" ou à une "Servlet" offert au client, il est possible de programmer une interface de connexion pour laquelle il devra fournir un mot de passe. De plus, si le *SGBD* sous-jacent requiert une authentification lors de la connexion, celle-ci pourra être exigée de la part du client. Dans ce cas, il est courant que le *SGBD* permettent différents types d'accès aux données: écriture, lecture ou mise à jour. Ces permissions d'accès, définies au niveau du serveur par l'administrateur des bases de données, restent valables pour l'application "Java".

Valeurs idéales: Accès libre (L) ou Elevée (E).

7.6.2. Système client.

7.6.2.1. Configuration cliente.

Pour les "Applets", la station cliente nécessite un navigateur compatible "Java" comme "Internet Explorer 3+", "Netscape Communicator 3+", "HotJava", etc. Rappelons que la plupart de ces browsers possède une fonction de désactivation du "Java" permettant à l'utilisateur de refuser l'exécution de ces programmes.

Pour les "Servlets", aucun outil spécial n'est nécessaire en plus du navigateur puisque les pages envoyées sont 100% *HTML*.

Au niveau des pilotes, nous avons montré qu'il en existait de 4 types (voir 7.5.4). Les deux premiers nécessitent l'installation manuelle de software sur la station cliente alors que pour les deux autres cela est fait automatiquement lors du chargement de "l'Applet".

Valeurs idéales: Légère (*BD* "semi-structurées"), Moyenne (M) (chargement automatique des pilotes) ou Lourde (Lo) (installation de software).

7.6.2.2. Interface graphique.

Tout objet graphique peut être inséré dans une "Applet Java". Cela va de l'image à la vidéo en passant par les graphiques. De plus, les boutons, listes déroulantes et autres boîtes à cocher sont des objets "Java" prédéfinis. On notera que la nouvelle version "Java 2" intègre encore plus de fonctionnalité à ce niveau, ce qui permet de créer des applications complètes comme on le ferait avec "Delphi" ou "C++": menus, fenêtres internes, etc.

Valeurs idéales: Simple (S), Moyenne (M) ou Complexe (C).

7.6.2.3. Interactivité.

L'interactivité est évidemment maximale puisque l'utilisateur peut consulter une *BD* comme si elle se trouvait sur sa propre machine. C'est au concepteur de décider des fonctions qu'il va offrir à l'utilisateur. Ensuite, celui-ci peut paramètrer les requêtes au choix et recevoir les informations voulues.

Valeurs idéales: Faible (F), Moyenne (M) ou Elevée (E).

7.6.3. Système serveur.

7.6.3.1. Configuration serveur(s).

Les "Applets" sont juste chargées sur le client. Dès lors, un simple serveur Web suffit. Pour l'utilisation des "Servlets", le serveur Web devra posséder une *JVM*. La plupart des serveur Web actuels supportent très bien cette technologie qui y est intégrée d'origine.

Pour ce qui est du reste, en fonction de l'architecture désirée, la configuration des serveurs va varier.

En modèle "Two-Tier", le serveur devra contenir un serveur Web ainsi que les pilotes des *SGBD* sous-jacents si le chargement de ceux-ci sont prévus automatiquement (voir 7.5.4).

Pour le modèle "Three-Tier", les applications du "Middle-Tiers" doivent pouvoir utiliser *JDBC*. Celui-ci doit donc être installé en plus du serveur Web.

Valeurs idéales: Légère (L) ou Moyenne (M).

7.6.3.2. Distribution des applications.

Grâce aux "Applets" et aux "Servlets", il est possible d'utiliser "Java" en version "Client-Side" ou/et en version "Server-Side". De plus, ces deux technologies peuvent être combinées: une "Servlet" peut envoyer une page contenant une "Applet".

Pour les "Servlets", celles-ci doivent absolument se trouver avec le serveur Web, ce qui signifie que la distribution des application sur plusieurs serveurs n'est pas possible.

Valeurs idéales: Client (C), Serveur (S) ou Client/Serveur (CS).

7.6.3.3. Distribution des BD.

Le modèle "Three-Tier" permet la gestion de *BD* sur des systèmes distants du serveur Web.

Valeurs idéales: Existante (E).

7.6.4. Critère de conception.

7.6.4.1. Compétences locales.

La programmation "Java" requiert des connaissances informatiques en langage "orienté objets". Les programmeurs habitués au "C++", au "Delphi" ou au "JavaScript" n'auront pas de problèmes pour programmer en "Java". Toutefois, "Java" est un langage assez simple à apprendre (moins complexe que "C++" en tout cas). Le tout étant de s'habituer à travailler avec des objets. La connaissance du *SQL* est bien entendu requise pour l'utilisation de bases de données.

Valeurs idéales: Elevée (E).

7.7. Bibliographie.

[9] George Reese, "*Database Programming with JDBC and JAVA*", O'Reilly & Associates, United States of America, June 1997.

[10] Jason Hunter et William Crawford, "*JAVA Servlet Programming*", O'Reilly & Associates, United States of America, Octobre 1998.

[11] Antoine Mirecourt, "*Le développeur Java 2 - Edition 1999*", Osman Eyrolles Multimedia, Paris, France, 1999.

[12] Graham Hamilton & Rick Cattell, *"JDBC : A Java SQL API – version 1.20"*, Javasoft, January 10, 1997.

[13] Ph. Thiran, O. Demoulin, *"Projet RW n°3062, Inter-DB : Interopérabilité des Systèmes d'Information hétérogènes et distribués – JDBC"*, Facultés Universitaires Notre-Dame de la Paix, Institut d'informatique, Namur, Belgique, 4 février 1998.

[14] Bradley F. Burton and Victor W. Marek, *"Applications of JAVA programming language to database management"*, Department of Computer Science, University of Kentucky, Lexington.

[15] Anup K. Ghosh, "*E-Commerce Security - Weak links, Best defenses*", Wiley Computer Publishing, 1998.

[16] Kristof Vermeire, "*Special Java - Le lien entre serveurs et surfeurs*", Computer Magazine n°94/95, pp. 73-75, Juillet-Août 1999.

Site "Java".

{20} "The Java Development Kit" – Site officiel (SUN): http://java.sun.com/products/jdk/index.html

{21} "JavaMed" – Tout savoir sur Java (en français): http://www.med.univ-rennes1.fr/~courtin/JavaMed/MultimediaJava.html

{22} "Visual J++" – Version 6.0 (Microsoft): http://www.asia.microsoft.com/visualj/featurepeek/

{23} "Academic Program for Java" (IBM): http://www.ibm.com/java/academic/

{24} "The JDBC database access API" – Site officiel (SUN): http://java.sun.com/products/jdbc/

{25} "JDBC documentation" (SUN): http://java.sun.com/products/jdk/1.1/docs/guide/jdbc/index.html

{26} "DevEdge Online" – Site de Netscape:
http://developer.netscape.com/docs/technote/index.html?content=database/web/index.html

{27} "The Java Language Environment ; A white paper" (SUN):
http://java.sun.com/docs/white/langenv/

Chapitre 8. SCRIPTS CGI.

8.1. Introduction.

Le *CGI*, "Common Gateway Interface", a été la première technologie permettant de générer dynamiquement des informations pour le Web: les "scripts *CGI*" (programmes utilisant le *CGI*) offrent la possibilité de générer des pages Web instantanément à la demande de l'utilisateur plutôt que de transférer des pages écrites entièrement à l'avance.

Les possibilités du *CGI* sont multiples: gestion de formulaires avec "feedback" immédiat, gestion des "imagemaps" qui permettent d'atteindre une page différente en fonction de l'endroit "cliqué" sur une image, création d'un compteur affichant le nombre d'utilisateurs ayant accédé au site, mais aussi les connexions aux bases de données (recherches variées - cfr. moteurs de recherche par exemple).

Un des gros avantages du *CGI*, c'est sa grande simplicité: n'importe qui ayant une légère expérience en programmation peut écrire des scripts élémentaires qui fonctionnent. C'est seulement quand les besoins deviennent plus importants que la complexité se fait ressentir. A ce propos, Shishir Gundavaram résume bien la situation en ce sens: "In a way, CGI is easy the same way cooking is easy: anyone can toast a muffin or poach an egg. It's only when you want a Hollandaise sauce that things start to get complicated" [17].

8.2. Architecture.

Le *CGI* est la partie du serveur Web qui peut communiquer avec d'autres programmes tournant sur ce serveur. La figure suivante illustre le procédé qui est suivi:

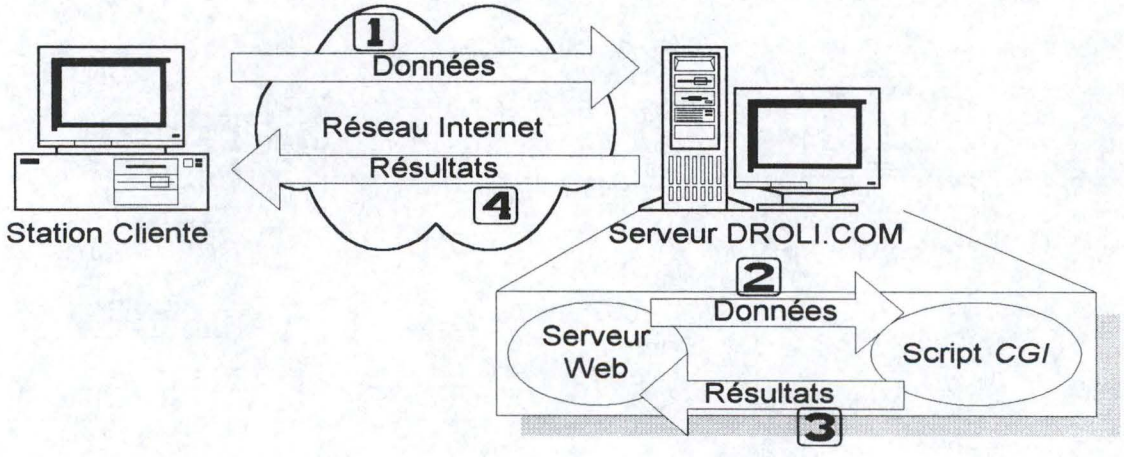

Figure 13: Architecture *CGI*.

Le client envoie ses données au serveur (1) qui peut alors appeler un programme ("script *CGI*") et lui passer ces données (2). Le programme s'exécute et renvoie la réponse au serveur Web (3) qui la retransmet au client sous format *HTML* par exemple (4). Ce document qui est envoyé au client avec le résultat de la requête est donc créé dynamiquement.

8.3. Applications.

Le *CGI* permet donc l'exécution d'applications sur le serveur, on parle de technologie "server-side", littéralement "s'exécutant du côté serveur", pour les différencier des applications "client-side" qui s'exécutent "côté client". Nous avons vu précédemment que ces deux types de technologies avaient leurs avantages et leurs inconvénients (3.4). Voyons maintenant les grands types d'applications que le *CGI* permet.

8.3.1. La gestion des formulaires.

Les scripts *CGI* sont souvent utilisés pour gérer les informations introduites dans les formulaires *HTML* (voir 4.4). Généralement, les formulaires ont pour fonction de rapatrier des informations à propos d'un utilisateur dans un but de marketing, ou pour lui envoyer un produit quelconque par exemple. Mais l'utilisation des formulaires permet également à l'utilisateur de faire des recherches approfondies sur les documents se trouvant sur le serveur: un "script *CGI*" peut analyser les informations fournies par l'utilisateur et lui retourner le ou les documents qui correspondent à ces critères.

8.3.2. Les passerelles.

Comme son nom l'indique, le *CGI* peut servir de "gateway", c'est-à-dire de "passerelle" vers des informations non directement accessibles par le client. Par exemple, si vous désirez mettre une base de donnée à disposition des utilisateurs, un lien *URL* vers le fichier correspondant ne fonctionnera certainement pas. Par contre, il existe des langages *CGI* (voir plus loin) qui permettent d'interroger ces *BD* (par exemple en *SQL*) et de fournir ainsi le résultat de la requête au client (voir figure).

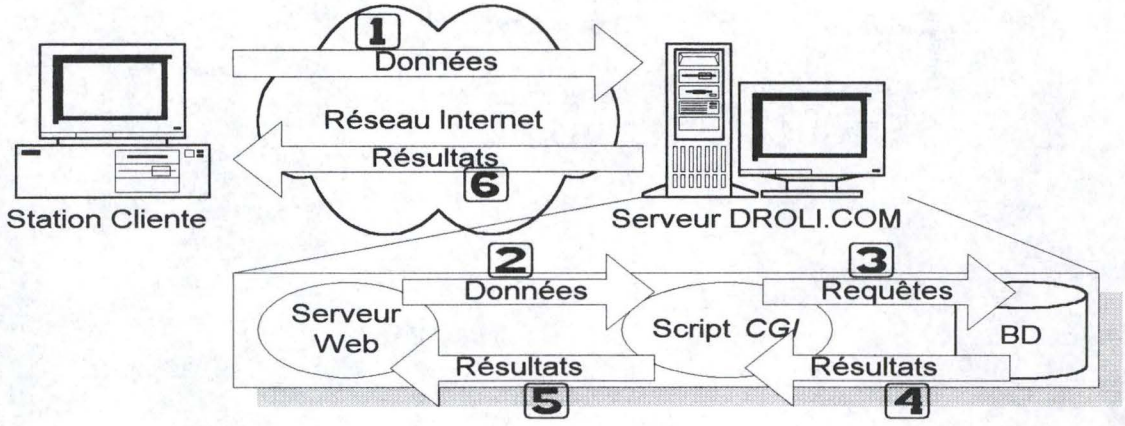

Figure 14: Passerelle *CGI* vers une *BD*.

Les données sont envoyées au serveur Web par l'utilisateur, par exemple via un formulaire (1). Ces données sont passées au "Script *CGI*" (2) et correspondent en fait à des requêtes sur une base de données. Le "Script *CGI*" traduit donc ces informations en requêtes qu'il envoie au *SGBD* (3). Ce dernier exécute la requête sur la *BD* et envoie le résultat au "Script" (4) qui peut alors construire un document résultat qu'il passe au serveur Web (5). Celui-ci n'a alors plus qu'à transférer le fichier vers la station cliente (6).

En fait, la "passerelle" peut se faire vers n'importe quel programme tournant sur le serveur. Par exemple, avec les informations reçues par le serveur Web, le "Script *CGI*" peut lancer le serveur de mails et ainsi envoyer un courrier électronique à quiconque possèdant une adresse e-mail. Les possibilités sont infinies et dépendent donc des programmes installés sur le serveur.

8.3.3. Les documents virtuels.

La création de document "virtuels" ou "dynamiques" constitue le cœur du *CGI*. Les documents "virtuels" sont créés sur le tas, en réponse à une requête particulière. Ces documents se déclinent sous plusieurs types: du *HTML*, du simple texte ("plain text"), des images, ou même un document audio.

Un document "virtuel" peut être très simple et contenir par exemple le nom de l'utilisateur et un remerciement pour l'envoi d'un formulaire. D'un autre côté, il est possible de créer des documents "virtuels" multimédia très complexes. On peut penser à une application de modification d'images: l'utilisateur choisit une image dans le répertoire qui lui est proposé; ensuite, il décide de modifier le contraste; pour se faire, un "Script *CGI*" est appelé, avec un ensemble d'arguments fixés par l'utilisateur; le "Script" lance alors un programme adéquat sur le serveur chargé de modifier l'image comme demandé; finalement, l'image résultat est envoyée et affichée sur l'écran de la station cliente.

8.4. Exemples.

On trouve énormément de sites utilisant les "Scripts *CGI*". Quelques exemples typiques peuvent être trouvés aux adresses qui suivent:

- http://www.lycos.com: Lycos est un moteur de recherche; à partir de mots clés il effectue une recherche sur les sites répertoriés dans sa base de données ayant un rapport avec ces mots;
- http://www.ravenna.com/coloring: Coloring Book est une application permettant à l'utilisateur de mettre une image en couleur;
- http://www.cosy.sgb.ac.at/rec/guestbook: Ce site propose un "livre d'or" ("guestbook") permettant aux visiteurs d'une page de laisser leur avis à propos du site qu'ils viennent de visiter;
- http://enterprise.ic.gc.ca/cgi-bin/j-e: Dictionnaire Anglais/Japonnais.

8.5. Fonctionnement du *CGI*.

La plupart des serveurs exigent que les "Scripts *CGI*" se situent dans un répertoire particulier, généralement appelé "cgi-bin", à des fins de sécurité. En effet, les "Scripts *CGI*" étant des programmes exécutés sur le serveur, il est primordial de pouvoir les surveiller afin qu'ils ne représentent pas une faille dans la sécurité du système. L'accès en écriture à ce répertoire spécifique sera donc fournit avec précaution aux programmeurs dignes de confiance et ayant des connaissances approfondies du système serveur. De plus, ces programmes doivent avoir une extension de fichier particulière, le plus souvent ".cgi". Ces paramètres sont caractéristiques du serveur et peuvent être modifiés par le responsable attitré.

Lors de la connexion à une adresse *URL* associée à un "Script *CGI*", tout se passe quasiment comme pour n'importe quelle autre requête de document. La différence principale se situe au niveau du serveur Web qui reconnaît l'adresse comme étant un programme *CGI*. Dès lors, le serveur ne va pas simplement retourner le fichier vers le client mais va plutôt essayer d'exécuter le programme.

De manière générale, une connexion fournit au serveur un certain nombre de paramètres concernant le protocole à utiliser, les formats de données acceptés, le type de navigateur utilisé, etc. Toutes ces informations concernant le système client sont donc fournies au serveur Web qui va de même les mettre à disposition du "Script *CGI*". Ces données sont donc utilisables par le programme. De plus, on sait que l'utilisateur peut passer ces propres paramètres (entrés dans un formulaire par exemple). La façon dont le "Script *CGI*" peut accéder à ces informations dépend du système d'exploitation du serveur. Sur un système "Unix", les programmes *CGI* accèdent aux données via l'entrée standard (*STDIN*) et via les variables d'environnement "Unix" (voir 4.4.1).

Une fois que le programme *CGI* commence son exécution, il peut soit créer un nouveau document, soit fournir une adresse *URL* vers un document existant. Sur "Unix", les programmes envoient leur résultat vers la sortie standard (*STDOUT*) en un "flux de données" ("data stream"). Ce "flux" est constitué de deux parties. La première partie représente un "en-tête *HTTP*" décrivant (au minimum) le format des données retournées (*HTML*, texte, *GIF*, *JPEG*, etc.). La seconde partie constitue le "corps", contenant les lignes de données conformes au format décrit dans l'en-tête. Ce "corps" de données ne sera ni modifié, ni interprété par le serveur.

Un programme *CGI* peut choisir d'envoyer directement son résultat au client ou bien de l'envoyer indirectement via le serveur Web. Si l'"en-tête" constitue un "en-tête *HTTP*" complet, le serveur Web ne fait aucune modification et envoie directement les données au client. Par contre, si "l'en-tête" n'est pas complet, le serveur Web va le compléter avant de l'envoyer. Généralement, on préférera fournir uniquement un "en-tête" partiel et laisser le serveur faire le reste du travail. Toutefois, on peut avoir envie d'envoyer directement les données au client et ainsi gagner du temps puisque le serveur ne doit pas analyser le document fournit.

8.6. Programmation *CGI*.

La question que l'on se pose maintenant est la suivante: "quel langage doit-on utiliser pour faire du *CGI* ?". La réponse est très simple: n'importe quel langage peut être utilisé. Toutefois, certains conviennent mieux que d'autres pour la programmation *CGI*. Dès lors, avant de choisir un langage, il convient de considérer les points suivants:

- la facilité de manipulation de textes: la plupart des applications *CGI* impliquent une manipulation de textes d'une manière ou d'une autre (par exemple, pour décoder les données fournies par l'utilisateur, contenue dans une chaîne de caractères). Dès lors, un langage offrant des possibilités de "pattern matching" sera bien utile;

- la possibilité d'interface avec d'autres applications: par exemple avec des bases de données. C'est cet aspect du langage qui détermine les possibilités de création de "passerelles" ("gateway") vers d'autres sources de données;

- la possibilité d'accès aux variables d'environnement (sous "Unix"): puisque les données d'entrée du programme *CGI* proviennent de ces variables, l'accès à ces informations est primordial.

Les langages les plus populaires pour la programmation *CGI* sont "AppleScript", "C/C++", "C Shell", "Perl", "Tcl" et "Visual Basic". La suite décrit les avantages et les inconvénients de chacun d'eux:

8.6.1. "AppleScript" ("Macintosh" seulement).

Depuis la version 7.5 du système d'exploitation des "Macintosh", le langage "AppleScript" en fait partie intégrante. Bien que "AppleScript" manque d'opérateurs de "pattern-matching", certaines extensions ont été écrites pour pouvoir gérer facilement les chaînes de caractères. "AppleScript" peut également servir pour les "passerelles" vers des applications "Macintosh" grâce à "AppleEvents". Par exemple, un programmeur *CGI* sous "Mac" peut écrire un programme qui présente un formulaire à l'utilisateur, décode les données fournies, et questionne un base de données "Microsoft Fox Pro" directement avec "AppleScript".

8.6.2. "C/C++" ("Unix", "Windows", "Macintosh").

"C" et "C++" sont des langages très populaires chez les programmeurs, et certains les utilisent pour la programmation *CGI*. Ces langages ne sont pas recommandés au programmeur novice car ils imposent des règles strictes en ce qui concerne les déclarations de variables et de l'espace mémoire, ainsi qu'au niveau de la vérification des types ("type-checking"). De plus, ces langages manquent d'extensions pour la gestion des *BD* et de possibilités de "pattern-matching".

Toutefois, le "C" et le "C++" ont un avantage assez majeur: les "Scripts *CGI*" ainsi créés sont compilés en programmes binaires exécutables, ce qui nécessite moins de ressources système que l'utilisation d'interpréteurs comme "Perl" ou "Tcl" (voir ci-dessous).

8.6.3. "C Shell" ("Unix" uniquement).

Le "C Shell" ne possède pas d'opérateurs de "pattern-matching" et il faut donc recourir à d'autres ressources "Unix" pour contourner le problème. Toutefois, il existe un outil software, appelé "uncgi" et écrit en "C", qui décode les données d'un formulaire et les enregistre dans des variables d'environnement facilement accessibles par le programme *CGI*. De même, la communication directe avec des bases de données est impossible, à moins de recourir de nouveau à une application externe. Pour finir, le "C Shell" contient

quelques bogues ("bugs" ou erreurs de programmation) et des limitations qui en font un langage inadéquat, surtout pour les débutants.

8.6.4. "Perl" ("Unix", "Windows", "Macintosh").

"Perl" est de loin le langage le plus utilisé pour la programmation *CGI*. Il contient un grand nombre de fonctions puissantes et est facile à apprendre. Les avantages du "Perl" sont les suivants:

- il est "portable" (passage d'une plate-forme à une autre très aisé);
- il contient des opérateurs de manipulation de "strings" extrêmement puissants;
- il permet des appels de "commandes shell" (instructions directes pour le système d'exploitation) très facilement, et il fournit des fonctions équivalentes à certaines fonction du système "Unix";
- Il existe un grand nombre d'extensions offrant des fonctions spécialisées. Par exemple, il y a "oraperl", contenant des fonctions d'interface avec la base de données "Oracle".

8.6.5. "Tcl" ("Unix" seulement).

"Tcl" commence à gagner en popularité en tant que langage de programmation *CGI*. "Tcl" consiste en un "shell", "tclsh", pouvant être utilisé pour exécuter des "Scripts *CGI*". Tout comme "Perl", "tclsh" contient également des constructions simples mais il est plus difficile à apprendre et à utiliser. De même, il possède aussi des extensions pour les bases de données. Les opérateurs de "pattern matching" existent mais sont moins efficaces que ceux de "Perl".

8.6.6. "Visual Basic" ("Windows" uniquement).

"Visual Basic" est à "Windows" ce que "AppleScript" est à "Macintosh" en ce qui concerne la programmation *CGI*. Avec "Visual Basic", on peut communiquer avec d'autres applications "Windows" comme des bases de données par exemple. C'est donc un outil très performant pour développer des "Scripts *CGI*" sur "PC", d'autant plus qu'il est facile à apprendre. Toutefois, et une fois encore, "Visual Basic" manque d'opérateurs pour le "pattern matching".

8.7. *CGI* et bases de données.

Le *CGI* permet la gestion de bases de données "semi-structurées" et "relationnelles".

8.7.1. BD "semi-structurées".

Avec la programmation *CGI*, il est possible de lire et d'écrire des fichiers sur le serveur. On peut donc créer des fichiers représentant des bases de données "semi-structurées" en choisissant certaines conventions au niveau des délimitateurs. Ainsi, le document suivant représenterait une *BD* reprenant des informations sur des clients:

```
Nom;Prénom;Téléphone;Rue;Localité
PicPic;André;081/40.81.00;Rue de la Pince à Cheveux,69;Namur
Nime;Jannot;075/42.38.65;Av. du Pommeau de Douche,14;Bruxelles
Faindi;Roger;02/347.865;Rue de Laloix,27b;Bruxelles
...

La première ligne détermine les noms des colonnes et le délimitateur est le point-virgule
```

Exemple 46: *BD* "semi-structurée".

La gestion de ce genre de fichier ne pose pas de problème en *CGI* à condition de choisir un langage facilitant le "pattern matching" (voir 8.6). On pourra, par exemple, proposer à l'utilisateur d'entrer directement ses coordonnées ou de faire une recherche sur les données enregistrées. C'est au concepteur d'écrire les fonction de recherche sur sa base de données.

8.7.2. BD "relationnelles".

Ici aussi, la difficulté de gestion des *BD* "relationnelles" va dépendre du langage choisit pour la programmation du "Script *CGI*". Ce langage doit fournir les extensions nécessaires (c'est-à-dire une interface) permettant de "questionner" la *BD* et de "réceptionner" les résultats des requêtes. On se rapportera à la discussion sur les différents langages (8.6) pour savoir quel langage peut convenir. Citons par exemple les extensions "Oraperl" et "Sybperl" fournissant des fonctions de connexion aux bases de données "Oracle" et "Sybase" respectivement. L'équivalent de ces extensions existent également pour le langage "Tcl".

8.8. Première évaluation.

Nous allons maintenant décrire les particularités des "Scripts *CGI*" par rapport aux critères définis précédemment (voir Partie I.Chapitre 2: Critères de conception.). Une évaluation de ces critères en fonction des valeurs proposées dans cette même partie sera donnée en fonction des possibilités offertes par cette technologie.

8.8.1. Critères informationnels.

8.8.1.1. Structure des données.

Le *CGI* permet la création de fichier *HTML* de manière dynamique. Dès lors, son utilisation n'est pas nécessaire dans le cas de documents statiques, bien définis et peu structurés. En effet, pourquoi recourir à une création dynamique si le résultat affiché est toujours le même quelque soit le moment de la connexion ? Par contre, le recours au *CGI*

sera plus adéquat lorsque les données seront structurées, soit en bases de données "semi-structurées" soit en *BD* "relationnelles".

Valeur idéale: Elevée (E).

8.8.1.2. Volume.

De le cas données structurées en *BD* "relationnelles", le volume pourra être plus conséquent que dans le cas de *BD* "semi-structurées". En effet, la recherche sur des *BD* "relationnelles" se font par des *SGBD* utilisant des "index" accélérant le processus. Par contre, les recherches sur des *BD* "semi-structurées" correspondent à des analyses de fichiers de textes, c'est-à-dire des analyses de caractères ("pattern matching"). Dès lors, le volume des documents ne devra pas être trop gros pour être efficace.

Valeurs idéales: Petit (P) (*BD* "semi-structurées") ou Gros (G) (*BD* "relationnelles").

8.8.1.3. Fréquence des changements.

Lorsque les informations changent peu souvent, le recours à des documents dynamiques n'est pas nécessairement idéal. Par contre, l'utilisation de *BD* permet la mise à jour facile par des recherches appropriées. Et à ce niveau, on l'a vu, le *CGI* offre une solution tout à fait intéressante.

Valeurs idéales: Faible (F), Moyenne (M) ou Elevée (E).

8.8.1.4. Sécurité.

Les "Scripts *CGI*" sont situés sur le serveur au même titre que les autres documents destinés au Web (*HTML*, images, etc.). dès lors, les droits d'accès sont identiques, c'est-à-dire que l'utilisateur qui a droit d'accès au serveur Web sera habilité à exécuter les programmes *CGI* qui s'y trouvent. Les protection d'accès doivent donc être fixées au niveau du serveur Web. Toutefois, rien n'empêche un concepteur d'écrire sa propre procédure d'authentification. Par contre, la protection des données contenues dans une *BD* "relationnelle" est du ressort du *SGBD* correspondant. Si celui-ci fournit un moyen d'authentification, il pourra être utilisé par le concepteur pour la connexion à la *BD*.

Valeurs idéales: Accès libre (L) (*BD* "semi-structurées") ou Elevée (E) (*BD* "relationnelles").

8.8.2. Système client.

8.8.2.1. Configuration cliente.

Le *CGI* étant une technologie "server-side", la configuration cliente nécessaire est minimale. Un simple navigateur suffit puisque les documents réellement envoyés sont des documents 100% *HTML*..

Valeur idéale: Légère (L).

8.8.2.2. Interface graphique.

L'interface graphique correspond à celle offerte par le navigateur. C'est donc l'interface du *HTML* avec ses formulaires.

Valeurs idéales: Simple (S) ou Moyenne (M).

8.8.2.3. Interactivité.

Le niveau d'interactivité offert par le *CGI* est bien entendu maximal puisque les pages qui sont affichées sur son écran peuvent être "fraîchement" créées en fonction des informations fournies par l'utilisateur ou du moment de la connexion.

Valeurs idéales: Faible (F), Moyenne (M) ou Elevée (E).

8.8.3. Système serveur.

8.8.3.1. Configuration serveur(s).

La majorité des serveurs Web habituels supporte les "Scripts *CGI*". Toutefois, les possibilités de connexions externes (applications et bases de données) vont varier d'un serveur à l'autre, essentiellement au niveau de la plate-forme utilisée. On se référera à la discussion sur les langages *CGI* au point 8.6 pour plus d'informations à ce sujet.

Toutefois, il faut noter que le langage "Perl" convient pour les plates-formes usuelles comme "Unix", "Windows" ou "Macintosh". La migration d'un programme "Perl" vers une de ces "plates-formes" n'impose que de légers changements de programmation (fonctions différentes, appel de commandes système, etc.).

Valeur idéale: Légère (L).

8.8.3.2. Distribution des applications.

Le *CGI* est ue technologie purement "Server-Side".

De plus, tout "Script *CGI*" doit se trouver sur la même machine que le serveur Web puisque c'est lui qui lance le processus correspondant et qui récupère les informations pour les transmettre au client.. Il n'est donc pas possible de communiquer avec d'autres applications distantes.

Valeur idéale: Serveur (S).

8.8.3.3. Distribution des BD.

Les *BD* doivent se trouver sur le même disque que le serveur Web.

Valeur idéale: Inexistante (I)

8.8.4. Critère de conception.

8.8.4.1. Compétences locales.

Le *CGI* repose sur de la programmation. Toutefois, des langages comme "Perl" sont faciles à apprendre et à utilisés pour des programmes simples (formulaires d'inscription par exemple). Pour des choses plus compliquées, la programmation exigera une expérience informatique plus avancée.

Un des avantages principaux, c'est la variété des langages pouvant être utilisé pour la programmation *CGI*. N'importe quel langage convient, ce qui signifie qu'un programmeur connaissant le "C" par exemple peut programmer des "Scripts *CGI*". Attention toutefois à la facilité d'utilisation de certains langage pour la gestion des chaînes de caractères fort nécessaire en *CGI* (voir discussion en 8.6).

Valeurs idéales: Moyenne (M) ou Elevée (E).

8.9. Bibliographie.

[17] Ian Graham, *"Introduction to HTTP and CGI"*, Instructional and Research Computing, University of Toronto.

[18] Shishir Gundavaram, "*CGI Programming on the World Wide Web - First Edition*", O'Reilly & Associates, Mars 1996.

Sites "CGI".

{28} "Un Nouveau Guide Internet" – Pour utilisateurs et concepteurs: http://ungi.cge.net/cyber/ (chap. 36 à 40)

{29} "Big Nose Bird" – Introduction au *CGI*: http://www.bignosebird.com/

{30} "Publier sur le World Wide Web" – section *CGI*: http://fp.planete.net/Publier/

{31} "Munica Web Database Solution Provider" – création d'un site avec gestion de *BD* à partir d'un logiciel payant: http://www.munica.com/dp-tutor.htm

{32} "Guide de programmation HTML" – Aide sur *CGI*: http://www.webdeveloppeur.com/Conception/introductionCGI.html

{33} "The Common Gateway Interface" – Page de la *NCSA*: http://hoohoo.ncsa.uiuc.edu/cgi/overview.html

{34} "CGI Made Really Easy" – *CGI* et formulaires: http://www.jmarshall.com/easy/cgi/

{35} Sites *CGI* de "Yahoo.com": http://www.yahoo.com/Computers_and_Internet/Internet/World_Wide_Web/CGI__Common_Gateway_Interface/

Chapitre 9. JAVASCRIPT.

9.1. Du déjà vu ?

Nous avons déjà parler de JavaScript au point 4.6 (chapitre sur le *HTML*) où je vous signalais qu'il était possible d'insérer un petit programme "JavaScript" au sein de vos documents *HTML*. Ce programme est alors interprété par votre navigateur lors du chargement de la page. Cette technologie porte le nom de "Client-Side JavaScript" puisque utilisant les ressources du système client. Mais ce langage permet également une technologie "Server-Side" et c'est ce que nous analysons ici.

9.2. "Client-Side JavaScript" et "Server-Side JavaScript".

Comme déjà signalé, "JavaScript" est un langage développé par "Netscape" (souvenez-vous, "Microsoft" en a développé sa propre version appelée "Jscript" - voir 4.6). C'est donc un langage "orienté objets" permettant de développer des applications clientes ou serveurs. Dans sa version "Sever-Side", "JavaScript" est assez similaire aux "Scipts *CGI*" (Chapitre 8) puisqu'il permet de créer des pages *HTML* de manière dynamique en utilisant des entrées utilisateur et des données contenues dans des fichiers ou des bases de données "relationnelles".

Le langage utilisé pour la technologie "Server-Side" correspond au langage "Client-Side" augmenté de fonctionnalités purement dédiées aux applications serveurs. De plus, certaines fonctions agissent différemment quand elles sont exécutées côté serveur. Une autre grande différence réside dans le fait que les applications "Server-Side" sont compilées avant installation pour améliorer les performances. Rappelons que le "Client-Side JavaScript" est compilé et interprété par le navigateur au moment du chargement de la page.

La figure suivante illustre la décomposition du langage entre les technologies "Server-Side" et "Client-Side".

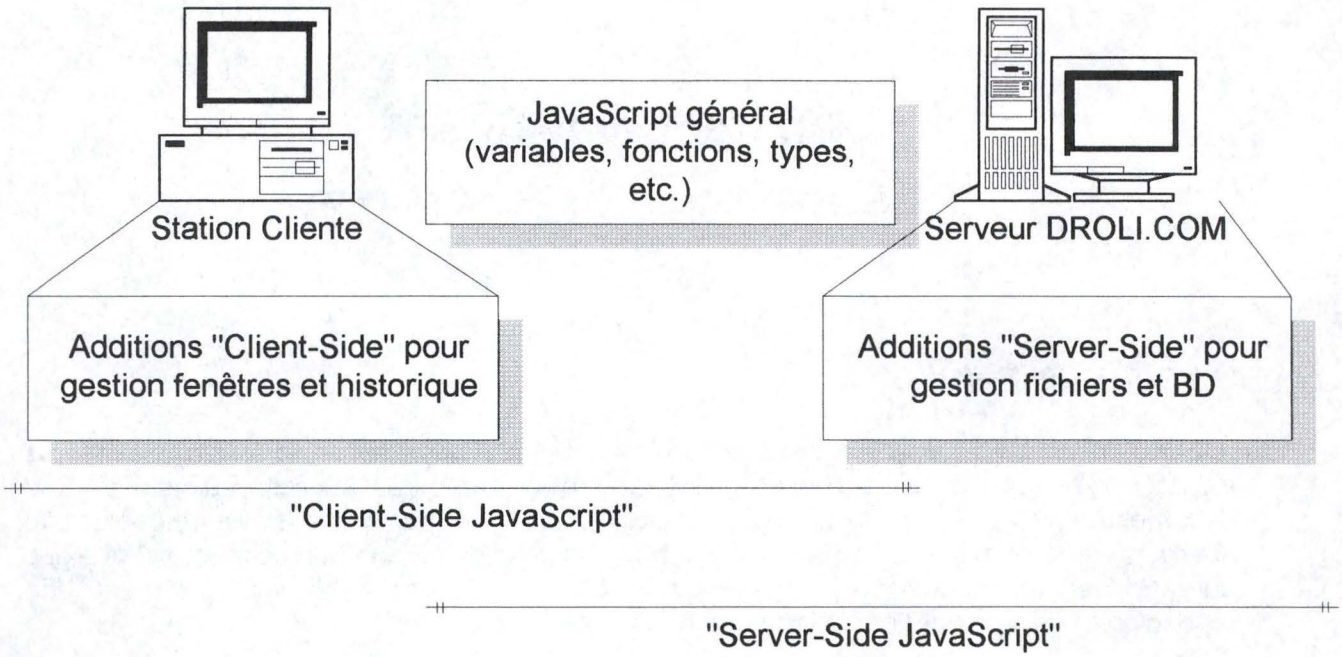

Figure 15: Langage "JavaScript" entre "Server-Side" et "Client-Side".

Le "Client-Side JavaScript" correspond donc au langage général plus des extra sous forme d'objets uniquement nécessaires pour un navigateur. De l'autre côté, le "Server-Side JavaScript" reprend également le même langage général plus des extra sous forme d'objets et fonctions de gestion applicables à un serveur.

9.2.1. "Client-Side JavaScript".

Les navigateurs tels que "Netscape Navigator 2.0+" et "Microsoft Internet Explorer 2.0+" sont capables d'interpréter des commandes "Client-Side JavaScript" insérées dans une page *HTML* (balise **<SCRIPT>**). Lors de l'appel d'une telle page par le client via son browser, le server Web envoie le contenu du document incluant le *HTML* et le "JavaScript". Le navigateur lit alors la page de haut en bas, affichant le résultat du code *HTML* et exécutant les commandes "JavaScript" au fur et à mesure.

Cette utilisation de "JavaScript" permet essentiellement de répondre à des événements liés à l'utilisateur comme un "click" de souris, une entrée de données dans un formulaire ou une activation d'un lien sur la page. Par exemple, on pourrait écrire une fonction "JavaScript" vérifiant la validité des données entrées dans un formulaire requérant un numéro de téléphone (nombre de chiffre correct, présence d'un préfixe valide, etc.) ou une adresse email (présence du "@"). Sans aucune transmission sur le réseau, le "JavaScript" peut ainsi vérifier les informations et afficher une boîte de dialogue si l'utilisateur entre une donnée invalide.

9.2.2. "Server-Side JavaScript".

Sur le serveur, le "JavaScript" est également inséré au milieu du code *HTML* (balise **<SERVER>**). Les commandes "server-side" peuvent alors connecter la page à une base de données "relationnelle", partager de l'information entre plusieurs utilisateurs d'une même

application, accéder aux fichiers du système ou communiquer avec d'autres applications au travers de "LiveConnect" et "Java" (voir 9.6.3). Le code *HTML* contenant le "Server-Side JavaScript" peut également inclure du "Client-Side JavaScript".

Contrairement à la version "Client-Side", les pages *HTML* contenant du "Server-Side JavaScript" sont compilées en fichiers exécutables. Cela signifie donc que le serveur Web doit pouvoir interpréter l'application ainsi créée, ce qui nécessite un software particulier: le "JavaScript runtime engine" (inclus avec les serveurs Web de "Netscape"). Nous en discuterons plus en détail un peu plus loin. Notons également que le code "JavaScript" peut être inséré dans un fichier séparé des documents contenant le code *HTML* et le "Client-Side JavaScript". On ne retrouve ainsi dans le fichier *HTML* que les appels de fonctions prédéfinies et utilisables dans plusieurs documents en même temps.

Lors de l'appel d'une page contenue dans l'application compilée, le "runtime engine" trouve la version compilée de cette page dans le fichier exécutable, exécute les commandes "Server-Side JavaScript" contenues dans cette page, et génère dynamiquement la page *HTML* à renvoyer au client. Le résultat de l'exécution du "Server-Side JavaScript" permet d'ajouter du nouveau code *HTML* ou/et des commandes "Client-Side JavaScript" à la page *HTML* originale. Une fois la page créée dynamiquement, celle-ci est envoyée au client comme tout autre document *HTML*. Le code *HTML* et les éventuelles commandes "Client-Side JavaScript" sont interprétés par le navigateur.

Au contraire des "Scripts *CGI*", le code source "JavaScript" peut être totalement intégré directement au sein des pages *HTML*, ce qui facilite le développement rapide et la maintenance des applications Internet. De plus, le "Session Management Service" contient des objets permettant de préserver des données persistantes entre plusieurs requêtes d'un même client, entre clients différents connectés à une même application, ainsi qu'entre plusieurs applications, ce qui n'est pas offert par les "Scripts *CGI*". Par contre, "JavaScript" permet la connexion à des bases de données "relationnelles" tout comme le *CGI*.

9.3. Architecture.

Comme le montre la figure suivante, "JavaScript" possède une architecture "à trois étages" ("Three-Tier Architecture") semblable au modèle "Three-Tier" de "Java" (voir 7.5.5).

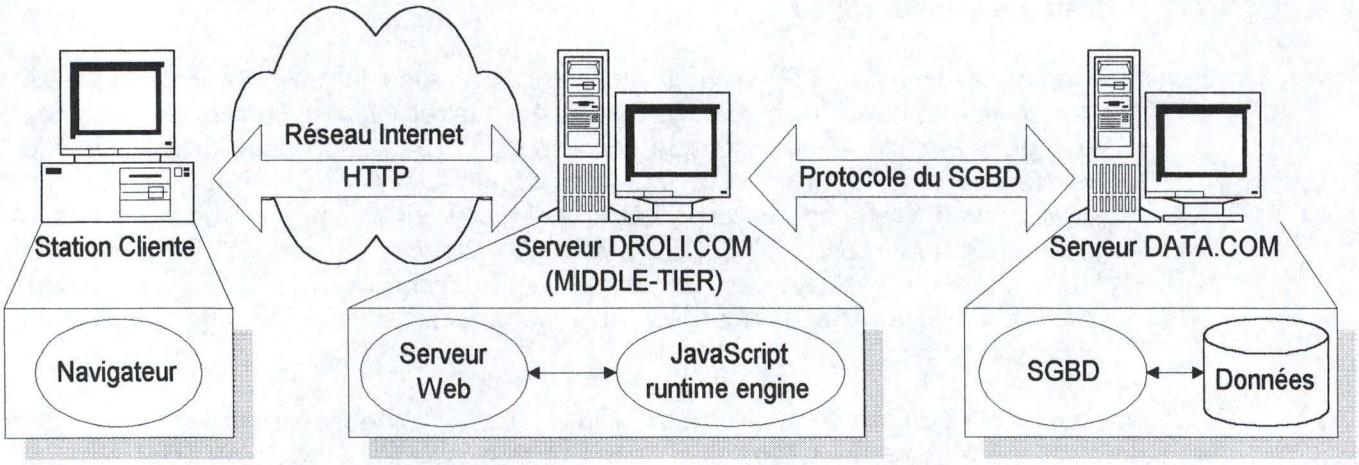

Figure 16: Architecture "Server-Side JavaScript".

Les trois "Tiers" sont donc les suivants:

- Un navigateur sur la station cliente tel que "Netscape Communicator" ou "Microsoft Internet Explorer". Si les pages Web contiennent du "Client-Side JavaScript", ce navigateur doit être compatible "JavaScript" et être configuré de manière à exécuter ces applications;

- Le "middle-tier" consiste en un serveur Web ainsi que le "JavaScript runtime engine" fourni avec les serveurs Web de "Netscape". C'est ce "JavaScript runtime engine" qui est chargé de l'exécution des pages contenant le "Server-Side JavaScript". C'est également lui qui doit gérer la sécurité des applications qu'il contient et qui doit contrôler les accès à une même application par plusieurs utilisateurs;

- Dans le cas de connexion à des *BD*, le troisième "Tier" est constitué des différents *SGBD*, tournant sur la même machine que le serveur Web ou externe au système. Nous verrons plus tard les types de *SGBD* supportés par la technologie "JavaScript".

9.4. Matériel nécessaire.

L'utilisation du "Server-Side JavaScript" pour ses pages Web requiert un certain nombre d'outils pour le développement et la mise à disposition au public. Ces outils sont contenus pour l'instant uniquement dans les serveurs Web de "Netscape". L'utilisation des serveurs de "Netscape" est donc nécessaire ce qui peut déjà constituer un premier frein envers un choix technologique.

Citons toutefois les outils à utiliser:

- Puisque l'application "Server-Side JavaScript" doit être compilée, un compilateur spécifique est nécessaire. "Nestcape" le fournit avec ses serveurs Web;

- Pour le déploiement du site sur Internet, un "JavaScript runtime engine" doit être associé au serveur Web en place. De nouveau, "Netscape" en intègre un d'origine dans ses serveurs Web;

- Si la connexion à des *BD* "relationnelles" est demandée, les *SGBD* correspondants doivent être installés et configurés (pas nécessairement sur la même machine). Pour permettre à "JavaScript" de correspondre avec ces bases de données, le "JavaScript runtime engine" utilise un outil appelé "LiveWire DataBase Service". Le serveur Web "Netscape Enterprise Server" fournit un tel support pour les bases de données suivantes: *ODBC*, "DB2", "informix", "Oracle" et "Sybase". Par contre, le "Netscape FastTrack Server" ne permet la connection qu'à *ODBC*. Pour rappel, *ODBC* est une interface "Microsoft" permettant la connexion à bon nombre de *SGBD* (voir 7.5.3).

Notons que si le *SGBD* ne se trouve pas sur la même machine que le serveur Web, une partie du software lié à ce *SGBD* doit généralement être installée avec ce serveur afin que ce dernier puisse devenir un client pour la *BD*.

9.5. Sécurité des applications "Server-Side JavaScript".

La gestion des applications "Server-Side JavaScript" se fait via "l'Application Manager" accessible au travers du serveur Web (c'est-à-dire avec une adresses *URL*). Dès lors, il est important de restreindre l'accès à ce "Manager" pour éviter que quiconque puisse ajouter, retirer, modifier, stopper ou lancer une application sur le serveur. "Netscape" fournit donc la possibilité de protéger l'accès à "l'Application Manager". Toute personne désirant s'y connecter devra posséder un login et un mot de passe définis par l'administrateur du serveur. Il est à noter que cette protection n'entre pas en compte pour l'exécution des applications. L'accès aux fichiers reste du ressort de serveur Web lui-même. Une application "Server-Side JavaScript" reste identique à ce niveau à un document *HTML*. Les restrictions d'accès doivent donc être définies au niveau du serveur Web.

9.6. Fonctionnalités principales.

9.6.1. Variables CGI.

Comme on l'a montré dans le chapitre sur le *CGI* (voir Chapitre 8), il existe un certains nombre de variables que le système met à jour pour chaque connexion: ce sont les variables d'environnement appelées "variables *CGI*". Comme pour les "Scripts *CGI*", les application "Server-Side JavaScript" peuvent accéder à ces données mais, ici, elles sont contenues dans des objets et des fonctions "JavaScript" directement utilisables par le programme. On se rappellera que dans le cas du *CGI*, l'accès à ces informations dépendait fortement du langage choisi.

Ces variables permettent notamment de connaître l'adresse *IP* du client, le protocole utilisé pour la connexion, etc.

9.6.2. Communication entre le serveur et le client.

Le "JavaScript" offre de nouveaux objets dans sa version "Server-Side" permettant de partager des données entre plusieurs requêtes d'un même client, entre plusieurs utilisateurs d'une même application, ou même entre plusieurs applications tournant sur le serveur. Ces objets sont gérés automatiquement par le "JavaScript runtime engine" et peuvent être utilisés pour toute communication entre le serveur et le client.

Les objets prédéfinis *request*, *client*, *project* et *server* contiennent des informations persistantes pour différentes périodes et sont disponibles pour les différentes applications "Server-Side JavaScript". Il y a:

- un objet *server* partagé par toutes les applications du serveur.
- des objets *project* séparés pour chaque application;
- un objet *client* par navigateur connecté à une application particulière;
- des objet *request* séparés pour chaque requête d'un client particulier envers une application particulière.

La figure suivante illustre l'ensemble de ces objets à un moment donné:

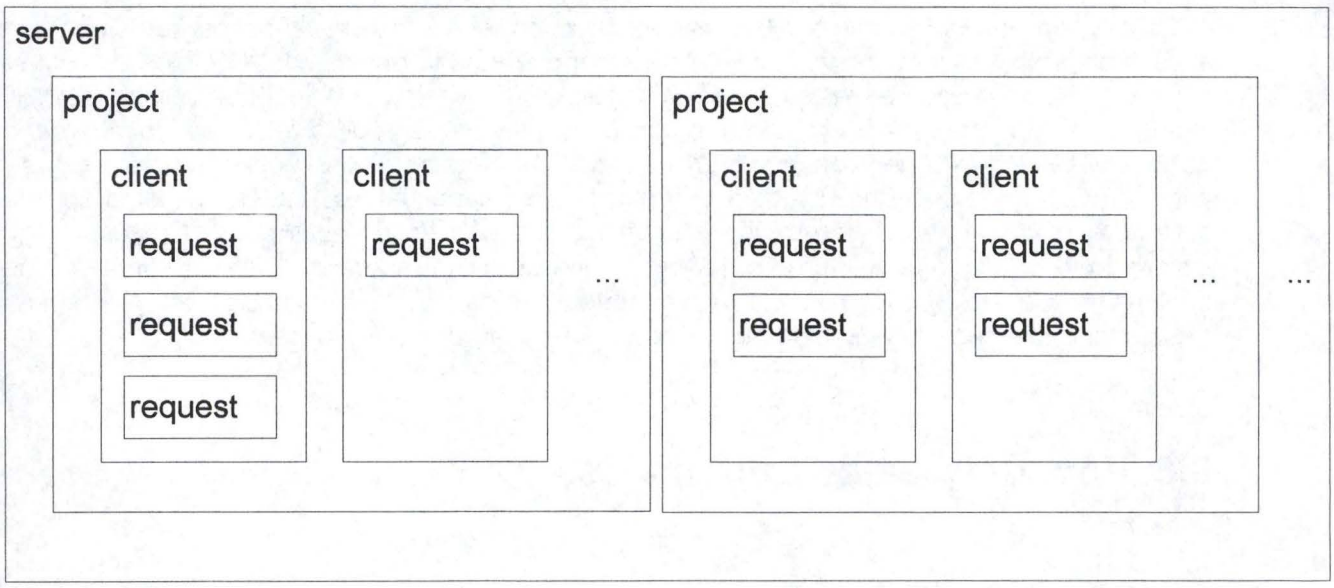

Figure 17: Les objets "JavaScript" de liaison client-serveur.

9.6.2.1. L'objet "request".

L'objet *request* contient des données spécifiques à la requête courante du client: type de navigateur client, adresse *IP* du client, protocole utilisé, etc. Cet objet regroupe également les données formulaire sous forme de variables séparées comme l'illustre l'exemple qui suit:

```
FORMULAIRE HTML
<FORM ACTION="Test.html">
Prénom: <INPUT TYPE="text" NAME="Prenom" SIZE="15"><BR>
Nom: <INPUT TYPE="text" NAME="Nom" SIZE="15"><BR>
<INPUT TYPE="submit" VALUE="envoi des données">
</FORM>

ENTREES UTILISATEUR
Prénom: OLIVIER
Nom: DAVREUX

OBJET request LORS DE L'ENVOI DES DONNEES
Le "JavaScript runtime engine" crée et initialise automatiquement les propriétés
      suivantes:
      request.Prenom = "OLIVIER"
      request.Nom = "DAVREUX"
Ces valeurs sont directement accessibles par un programme "Server-Side JavaScript"
```

Exemple 47: Valeurs de formulaire dans l'objet *request* en "Server-Side JavaScript".

Il est également possible de créer ses propres propriétés pour l'objet *request* mais il faut garder à l'esprit que ces variables n'existeront qu'un instant puisqu'elles disparaîtront en même temps que l'objet *request*.

Le "JavaScript runtime engine" construit un nouvel objet *request* pour chaque requête qu'il reçoit, c'est-à-dire lorsqu'un utilisateur entre une adresse *URL* dans son navigateur,

lorsqu'il clique sur un hyperlien, ou lorsqu'un programme "JavaScript" ("Server-Side" ou "Client-Side") choisit de charger une nouvelle page par l'utilisation d'une méthode adéquate.

Une fois la réponse fournie (envoi de la page demandée), l'objet *request* est détruit par le "JavaScript runtime engine". La durée de vie de cet objet est donc très petite (moins d'une seconde en général).

9.6.2.2. L'objet "client".

Plusieurs navigateurs peuvent accéder en même temps à une application "JavaScript" sur le serveur. L'objet *client* permet de gérer ces appels séparément et de garder la trace des connections simultanées d'un même client à une même application "JavaScript".

Le "JavaScript runtime engine" construit un nouvel objet *client* pour chaque paire client/application: un navigateur connecté à deux applications différentes possède donc deux objets *client* différents (un pour chaque application). A chaque première connexion à une application, le "runtime engine" construit donc un objet *client* différent et il peut exister des milliers d'objets *client* actifs en même temps.

L'objet *client* ne contient aucune propriété prédéfinie, c'est donc un objet vide initialement servant à recevoir des données spécifiques à l'application. Par exemple, une propriété à assigner à cet objet pourrait être un identifiant d'utilisateur permettant de gérer plusieurs utilisateurs sur une même application. Ce sera le cas pour des jeux ou des communications entre plusieurs personnes.

Une fois qu'un utilisateur accède à une application, il n'y a aucune garantie que ce client exécute une nouvelle requête dans un futur proche. Dès lors, l'objet *client* possède un mécanisme d'expiration permettant au "JavaScript runtime engine" de faire le ménage de temps en temps. Chaque fois que le serveur reçoit une requête pour une même application, la durée de vie de l'objet *client* est réinitialisée. Notons que cette durée de vie peut être changée à tout moment en utilisant la fonction "JavaScript" correspondante.

9.6.2.3. L'objet "project".

L'objet *project* contient les données globales pour une application particulière et permet le partage d'informations entre plusieurs clients utilisant une même application. Cet objet est créé lors de la mise en place de l'application sur le serveur et chaque client y accédant partage le même objet *project*. Lors de la déconnexion du serveur, cet objet est détruit. Sa durée de vie est donc limitée dans le temps et toute information devant persister au-delà de ce délai doit être conservée autre part (dans une *BD* par exemple).

Ici non plus, il n'existe aucune propriété prédéfinie pour l'objet *project*. Il sera donc dédié aux données spécifiques à l'application pour le partage entre plusieurs utilisateurs. On pourra par exemple retenir un numéro d'identifiant pour le client suivant qui viendra se connecter à l'application. Ainsi, en enregistrant cet identifiant dans l'objet *client* et en l'incrémentant à chaque connexion dans l'objet *project*, il est possible de gérer chaque client séparément.

9.6.2.4. L'objet "server".

L'objet *server* contient les données globales pour le serveur et permet le partage d'informations entre applications tournant sur ce serveur. Il contient également les informations concernant ce serveur tels que son nom de domaine, le numéro de port et le protocole utilisés, la plate-forme, etc. Cet objet est créé lors de la mise en route du serveur et est détruit lors de sa déconnexion. Toutes les applications qui tournent sur le serveur partage le même objet *server*.

En plus des propriétés prédéfinies pour cet objet *server*, il est possible de créer ses propres propriétés afin de gérer un partage entre plusieurs applications différentes.

9.6.3. Communication avec du code externe.

Une application "JavaScript" peut vouloir accéder à du code écrit dans un autre langage. Avec la technologie "LiveConnect" de "JavaScript", il est possible d'établir une telle communication entre du code "Java" et du code "JavaScript". Cette technologie existe également pour la version "Client-Side" et a été étudiée auparavant (voir 4.6). Toutefois, quelques différences existent dans l'utilisation sur le serveur. Notamment, pour qu'une classe "Java" puisse appeler une méthode d'un objet "JavaScript", il faut que cet objet ait d'abord fait appel à cette classe auparavant. Ainsi, la communication existe bien dans les deux sens mais l'initiative doit venir du "JavaScript: "JavaScript" → "Java" → "JavaScript". En "Client-Side JavaScript", "Java" peut initialiser une interaction avec "JavaScript", mais pas en "Server-Side JavaScript".

Avec le "Server-Side JavaScript", il est également possible d'utiliser des fonctions écrites en "C", "C++", "Pascal", etc. et compilées en "librairies externes" sur le serveur. Ces "librairies" sont en fait des "objets partagés" sous "Unix" ou des "Dynamic Link Libraries" (*DLL*) sous "Windows". Ces fonctions seront à prendre avec précaution si elles permettent l'exécution de commandes directes pour le système d'exploitation. On surveillera donc de près que l'utilisateur ne puisse pas entrer une telle commande dans un formulaire par exemple, et que cette commande ne soit exécutée par le système (imaginez les dégâts causés si l'utilisateur arrivait à exécuter la commande *format c:* !!!). Toutefois, il peut être intéressant d'utiliser de telles "libraries" dans les cas suivants:

- une fonction complexe existe et pourrait convenir pour l'application "JavaScript";
- l'application nécessite beaucoup de calcul de la part du processeur. En général, les fonctions compilées en "libraries" sont plus performantes que leurs équivalentes en "JavaScript";
- lorsque l'application requiert une tâche non directement exécutable en "JavaScript".

9.6.4. Envoi de mails.

Un objet spécifique est offert par "JavaScript" permettant d'envoyer des emails très facilement, ce qui n'est pas toujours le cas pour les "Scripts *CGI*" (dépend du langage choisi et de la plate-forme utilisée).

9.6.5. Accès au système de fichiers du serveur.

Le "JavaScript" fournit un objet permettant de lire et d'écrire des fichiers sur le serveur. Ainsi, il est aisé de stocker des informations persistantes sans devoir recourir aux bases de données. Attention toutefois ! Une application "Server-Side JavaScript" peut lire et écrire des fichiers se trouvant n'importe où sur le serveur, y compris les fichiers systèmes. Dès lors, la prudence est de mise pour éviter de laisser un accès aux mots de passe par exemple.

En créant un fichier de données, un programme "JavaScript" peut donc accéder à un base de données que nous avons qualifiée de "semi-structurée". De tels fichiers peuvent en effet reprendre un certain nombre d'informations décomposées en plusieurs champs séparés par des délimitateurs à choisir au préalable (voir Exemple 46).

9.7. Connexion aux bases de données.

Avec le serveur "Netscape Enterprise Server", la technologie "LiveWire Database Service" permet d'accéder aux bases de données "Informix", "Oracle", "Sybase" et "DB2", ainsi qu'au *SGBD* utilisant le standard *ODBC*. Par contre, avec le "Netscape FastTrack Server", la connexion n'est possible qu'au travers de l'interface *ODBC*.

Pour le reste, la connexion et l'envoi de requête s'effectue à l'aide d'objets spécifiques "JavaScript". Toute requête consiste en l'envoi d'un string représentant une requête *SQL* que le *SGBD* va interpréter. Une fois le résultat trouvé, le *SGBD* l'envoie au programme sous la forme d'un objet "JavaScript". Le procédé est donc similaire à celui utilisé par *JDBC* pour "Java" (détaillé en 7.5). La grande différence réside dans le fait que tout se passe sur le serveur et qu'il ne faut donc rien charger sur la machine cliente. De plus, *JDBC* permettait des connexions à un nombre plus élevé de types de *SGBD*. Mais n'oublions pas qu'une application "JavaScript" peut utiliser des classes "Java" et que donc "JavaScript" peut profiter des capacités de connexion aux *BD* de "Java".

9.8. Première évaluation.

Nous allons maintenant décrire les particularités du "Server-Side JavaScript" par rapport aux critères définis précédemment (voir Partie I.Chapitre 2: Critères de conception.). Une évaluation de ces critères en fonction des valeurs proposées dans cette même partie sera donnée en fonction des possibilités offertes par cette technologie.

9.8.1. Critères informationnels.

9.8.1.1. Structure des données.

Le "JavaScript" permet la gestion de fichiers sur le système serveur. On peut donc l'utiliser pour créer des bases de données "semi-structurées". De plus, la technologie "LiveWire" autorise la connexion d'une application avec une base de données "relationnelle" de type "Informix", "Oracle", "Sybase", "DB2" ou utilisant l'interface *ODBC*. Les données fortement structurées sont donc utilisables facilement en "Server-Side JavaScript".

Valeur idéale: Elevée (E).

9.8.1.2. Volume.

Le volume dépend du *SGBD* sous-jacent dans le cas d'une utilisation de *BD* "relationnelles". Il pourra généralement être assez conséquent.

Si l'on a recourt aux fichiers comme source de données "semi-structurées", la recherche et la mise à jour se feront de manière séquentielle, ce qui signifie que le volume ne devra pas être trop gros pour permettre un bon régime.

Valeurs idéales: Petit (P) (*BD* "semi-structurées") ou Gros (G) (*BD* "relationnelles").

9.8.1.3. Fréquence des changements.

Puisque l'application "JavaScript" peut utiliser et mettre à jour les dernières informations directement à leur source, il apparaît qu'une telle solution soit idéale pour des changements fréquents des données. Même si les changements sont effectués directement via le *SGBD* ou sur le fichier (cas des *BD* "semi-structurées") et non via l'application Internet, toute personne connectée aura l'information la plus récente au moment de l'appel.

Notons encore que la création dynamique de page convient spécialement pour des changements fréquents. On évitera donc cette méthode dans le cas de documents statiques.

Valeurs idéales: Faible (F), Moyenne (M) ou Elevée (E).

9.8.1.4. Sécurité.

Comme on l'a fait remarquer auparavant, certains problèmes de sécurités sont à prendre en compte lors de la conception d'une application en "Server-Side JavaScript", notamment en ce qui concerne l'exécution de commandes directes pour le système d'exploitation.

Pour ce qui est des accès aux bases de données "relationnelles", on se référera au *SGBD* correspondant pour savoir si des configurations sont possibles. En général, il est aisé de définir plusieurs niveaux d'accès en fonction de la personne connectée. Si c'est le cas, le programme "JavaScript" pourra exiger une authentification de la part de l'utilisateur afin d'éviter toute connexion non appropriée.

Valeurs idéales: Accès libre (L) ou Elevée (E).

9.8.2. Système client.

9.8.2.1. Configuration cliente.

Un simple navigateur est suffisant du côté client. Celui-ci devra peut-être supporter le "Client-Side JavaScript" si ce dernier est utilisé dans l'application. Dans l'affirmative, le navigateur devra également être configuré de manière à accepter l'exécution de ces programmes (les browsers permettent généralement de désactiver cette fonction pour des raisons de sécurité).

Valeurs idéales: Légère (L).

9.8.2.2. Interface graphique.

L'interface est celle offerte par le *HTML* avec les formulaires supportés par celui-ci.

Valeurs idéales: Simple (S) ou Moyenne (M).

9.8.2.3. Interactivité.

L'interactivité est maximale avec le "Server-Side JavaScript" puisque les pages qui sont affichées sont remplies "on the fly", en fonction des informations fournies par l'utilisateur. On notera également un petit "plus" pour cette technologie qui permet de garder en mémoire les différentes connexions, dans le temps, d'un même client.

De plus, le niveau d'interactivité est supérieur à certaines techniques comme les "Scripts *CGI*" puisque le "Server-Side JavaScript" permet de gérer plusieurs utilisateurs connectés à une même application. On voit donc apparaître une interactivité "inter-utilisateurs".

Valeurs idéales: Faible (F), Moyenne (M) ou Elevée (E).

9.8.3. Système serveur.

9.8.3.1. Configuration serveur(s).

Le "Server-Side JavaScript" nécessite l'utilisation des serveurs Web de "Netscape". De plus, le développement requiert un compilateur fourni également avec les serveur "Netscape". Ce type de solution est donc 100% "Netscape" du côté serveur.

Valeur idéale: Légère (L) (mais contraignante).

9.8.3.2. Distribution des applications.

Le "JavaScript" se décline sous deux formes: "Client-Side" et "Server-Side". Tout comme c'est le cas avec "Java", ces deux technologies peuvent être combinées: une page générée par le "Server-Side JavaScript" peut contenir du code en "Client-Side JavaScript". Toutefois, le "Client-Side JavaScript" n'offre aucune possibilité de connexion directe aux sources d'informations (*BD*).

Dans sa version "Server-Side", toute application doit se trouver sur la même machine que le serveur Web, la distribution avec des machines distantes n'est donc pas possible.

Valeurs idéales: Serveur (S) ou Client/Serveur (CS).

9.8.3.3. Distribution des BD.

Le "JavaScript" offre des possibilités d'architecture à trois étages ("Three-Tier"). Les *BD* peuvent donc se trouver à distance du serveur Web à condition d'y installer les parties de *SGBD* nécessaires à la communication. Bien sûr, les *BD* peuvent également être installées localement.

Valeur idéale: Existante (E).

9.8.4. Critère de conception.

9.8.4.1. Compétences locales.

Le "JavaScript" est un langage plus simple que le "Java". Toutefois, une connaissance en programmation "orienté objets" est nécessaire. La connaissance du *SQL* est bien entendu requise pour l'utilisation de bases de données.

Valeur idéale: Elevée (E).

9.9. Bibliographie.

Sites "Server-Side JavaScript".

{36} "Server-Side JavaScript" – Writing Server-Side JavaScript Applications:
http://developer.netscape.com/docs/manuals/enterprise/wrijsap/index.html

Chapitre 10. ACTIVE SERVER PAGES (*ASP*).

10.1. Le "Server-Side" de "Microsoft".

L'*ASP*, ou "Active Server Page" constitue la solution "Server-Side" de Microsoft. La similarité avec le "Server-Side JavaScript" de "Netscape" (Chapitre 9) est très forte, mais la technologie de "Microsoft" ne tourne que sur ses propres serveurs Web: "Windows NT Internet Information Server (*IIS*) 3.0", "Windows NT Workstation Personal Web Server" et "Windows 95 Personal Web Server" - il est à noter que la plate-forme doit donc être obligatoirement de type "Windows" !!! *ASP* permet donc de créer des scripts côté serveur pour la création dynamique de documents *HTML*. L'avantage par rapport aux "Scripts *CGI*" (Chapitre 8) par exemple étant l'intégration directe de cette technologie au serveur Web même, ce qui permet plus de fonctionnalités et plus de facilité comme pour le "Server-Side JavaScript": *ASP* fonctionne comme un service intégré au serveur Web et est optimisé pour des accès multiples et pour plusieurs utilisateurs. C'est donc un système rapide et facile à implémenter.

10.2. Les langages *ASP*.

Tout comme la programmation *CGI*, *ASP* est indépendant du langage utilisé. Par nature, le "VBScript" (forme de "Visual Basic") et le "Jscript" ("JavaScript" de "Microsoft") sont tout deux supportés d'origine par les serveurs Web de "Microsoft". Hormis ces langages "Microsoft", le concepteur d'une application *ASP* peut choisir également parmi les langages suivants: "Perl", "Rexx", "Python", etc. Toutefois, ces langages requièrent l'installation d'un software spécifique non fourni d'origine.

10.3. Principe du *ASP*.

Le principe de *ASP* est similaire au "Server-Side JavaScript" (voir Chapitre 9) et au *JSP* de "Java" ("Scriptlets Java" - voir 7.4).

Lorsqu'un navigateur demande un document *ASP* au serveur Web, celui-ci parcours ledit fichier, exécute les commandes *ASP* insérées dans le code *HTML* et renvoie au navigateur une page *HTML*. Un fichier *ASP* peut donc contenir du *HTML*, des commandes *ASP* (balises <% et %>) au sein de ce *HTML*, ainsi que des fonctions et procédures utilisables dans le corps du document. Par exemple, on peut créer le fichier suivant:

```
<HTML>
<BODY>
<% Prenom1 = "Olivier" %>
<% Prenom2 = "Alexandra" %>
```

```
<% If Time <= #6:00:00 PM# Then %>
Bonjour <%= Prenom1 %>
<% Else %>
Bonsoir <%= Prenom2 %>
<% End If %>
</BODY>
</HTML>
```

Si l'heure du serveur précède 18h00, le document affichera "Bonjour Olivier" à l'écran. Dans le cas contraire, il sera plus tard que 18h00 et le document affichera donc "Bonsoir Alexandra". On remarquera la balise <%= permettant d'insérer directement la valeur d'une variable définie préalablement.

Exemple 48: Document *ASP*.

10.4. Les objets *ASP*.

On se souvient qu'en "Server-Side JavaScript" (Chapitre 9), 4 objets permettaient de gérer les connexions aux différentes applications: *request*, *client*, *project* et *server*. En *ASP*, on retrouve l'équivalent en 5 objets standards:

- l'objet *request*, permet de gérer les informations provenant de l'utilisateur (données formulaire, "cookies", type de navigateur, etc.);
- l'objet *response* est utilisé pour envoyer des informations vers l'utilisateur (code *HTML*, "cookies", nouvelle *URL*, etc.);
- l'objet *server* permet l'interaction avec le serveur Web (accès aux *BD*, lecture de fichiers, etc.);
- l'objet *session* permet de stocker des informations à propos de la "session" courante de l'utilisateur. Les variables enregistrées dans cet objet existeront aussi longtemps que la session sera active. C'est l'équivalent de l'objet *project* en "Server-Side JavaScript". Chaque utilisateur possède un objet *session* par navigateur connecté);
- l'objet *application* permet de gérer l'ensemble des utilisateurs d'une même application. C'est l'équivalent de l'objet *server* en "Server-Side JavaScript";

10.5. L'accès aux bases de données.

L'utilisation des fichiers système est facilitée par un objet spécifique permettant la lecture et l'écriture sur le système serveur. A ce propos, il est possible de restreindre l'accès à certains fichiers afin d'éviter des problèmes de sécurité. Ainsi, des accès peuvent être définis pour un utilisateur particulier ou pour un groupe d'utilisateur. En fait, le principe repose sur le système de sécurité de "Windows NT". L'utilisation des fichiers permet donc de créer des bases de données "semi-structurées".

D'autre part, l'accès à des *BD* "relationnelles" est également possible via l'interface *ODBC*. Il faut donc que le SGBD utilisé soit supporté par la technologie *ODBC* (ce qui est le cas de la plupart des *SGBD* actuels: "Access", "MS-SQL Server", "Oracle", "Informix", etc.). A

l'aide d'objet *ASP* spécifiques, il est alors aisé d'envoyer des requêtes *SQL* vers le *SGBD* correspondant puis de traiter la réponse reçue sous forme d'objet.

10.6. Première évaluation.

Nous allons maintenant décrire les particularités du *ASP* par rapport aux critères définis précédemment (voir Partie I.Chapitre 2: Critères de conception.). Une évaluation de ces critères en fonction des valeurs proposées dans cette même partie sera donnée en fonction des possibilités offertes par cette technologie.

10.6.1. Critères informationnels.

10.6.1.1. Structure des données.

Le *ASP* permet la gestion de fichiers sur le système serveur. On peut donc l'utiliser pour créer des bases de données "semi-structurées". De plus, la connexion d'une application avec une base de données "relationnelle" est possible via l'interface *ODBC*.

Valeur idéale: Elevée (E).

10.6.1.2. Volume.

Le volume dépend du *SGBD* sous-jacent dans le cas d'une utilisation de *BD* "relationnelles". Il pourra généralement être assez conséquent.

Si l'on a recourt aux fichiers comme source de données "semi-structurées", la recherche et la mise à jour se feront de manière séquentielle, ce qui signifie que le volume ne devra pas être trop gros pour permettre un bon rendement.

Valeurs idéales: Petit (P) (*BD* "semi-structurées") ou Gros (G) (*BD* "relationnelles").

10.6.1.3. Fréquence des changements.

Puisque l'application *ASP* peut utiliser et mettre à jour les dernières informations directement à leur source, il apparaît qu'une telle solution soit idéale pour des changements fréquents des données. Même si les changements sont effectués directement via le *SGBD* ou sur le fichier (cas des *BD* "semi-structurées") et non via l'application Internet, toute personne connectée aura l'information la plus récente au moment de l'appel.

Notons encore que la création dynamique de page convient spécialement pour des changements fréquents. On évitera donc cette méthode dans le cas de documents statiques.

Valeurs idéales: Faible (F), Moyenne (M) ou Elevée (E).

10.6.1.4. Sécurité.

La sécurité des fichiers est gérée à la manière du système d'exploitation sous-jacent ("Windows NT"), ce qui permet de définir des accès personnalisés ou par groupe d'utilisateurs.

Pour ce qui est des accès aux bases de données "relationnelles", on se référera au *SGBD* correspondant pour savoir si des configurations sont possibles. En général, il est aisé de définir plusieurs niveaux d'accès en fonction de la personne connectée. Si c'est le cas, le programme *ASP* pourra exiger une authentification de la part de l'utilisateur afin d'éviter toute connexion non appropriée.

Valeurs idéales: Accès libre (L) ou Elevée (E).

10.6.2. Système client.

10.6.2.1. Configuration cliente.

Un simple navigateur est suffisant du côté client.

Valeurs idéales: Légère (L).

10.6.2.2. Interface graphique.

L'interface est celle offerte par le *HTML* avec les formulaires supportés par celui-ci.

Valeurs idéales: Simple (S) ou Moyenne (M).

10.6.2.3. Interactivité.

L'interactivité est maximale avec le *ASP* puisque les pages qui sont affichées sont remplies "on the fly", en fonction des informations fournies par l'utilisateur. On notera également un petit "plus" pour cette technologie qui permet de garder en mémoire les différentes connexions, dans le temps, d'un même client.

De plus, le niveau d'interactivité est supérieur à certaines techniques comme les "Scripts *CGI*" puisque le "Server-Side JavaScript" permet de gérer plusieurs utilisateurs connectés à une même application. On voit donc apparaître une interactivité "inter-utilisateurs".

Valeurs idéales: Faible (F), Moyenne (M) ou Elevée (E).

10.6.3. Système serveur.

10.6.3.1. Configuration serveur(s).

Le *ASP* nécessite l'utilisation des serveurs Web de "Microsoft".

Valeur idéale: Légère (L) (mais contraignante).

10.6.3.2. Distribution des applications.

Le ASP se décline uniquement en version "Server-Side".

Valeurs idéales: Serveur (S).

10.6.3.3. Distribution des BD.

Les *BD* doivent être avec le serveur Web.

Valeur idéale: Inexistante (I).

10.6.4. Critère de conception.

10.6.4.1. Compétences locales.

Le *ASP* est un langage plus simple que le "Java". Toutefois, une connaissance en programmation "orienté objets" est nécessaire. La connaissance du *SQL* est bien entendu requise pour l'utilisation de bases de données.

Valeur idéale: Elevée (E).

10.7. Bibliographie.

Sites "*ASP*".

{37} "MSDN online - Web Workshop" – An ASP you can grasp: the ABCs of Active Server Pages: http://msdn.microsoft.com/workshop/server/asp/ASPover.asp

{38} "MSDN online - Web Workshop" – ASP from A to Z: http://msdn.microsoft.com/workshop/server/asp/aspatoz.asp

{39} "MSDN online - Web Workshop" – ASP Technology feature overview: http://msdn.microsoft.com/workshop/server/asp/aspfeat.asp

{40} "MSDN online - Web Workshop" – ASP Tom: Server Q & A: http://msdn.microsoft.com/workshop/server/feature/morqa.asp

PARTIE III. SYNTHÈSE ET CHOIX TECHNOLOGIQUES.

Chapitre 11. SYNTHÈSE.

Maintenant que les technologies ont été décrites en profondeurs dans les chapitres précédents, le temps est venu d'en faire une synthèse dans laquelle nous situerons chaque technologie par rapport aux applications qui lui conviennent le mieux. Ensuite, nous comparerons ces techniques ensemble afin de pouvoir proposer des choix technologiques en fonction des critères définis au début de ce mémoire.

11.1. Tableau récapitulatif.

Afin de comparer les différentes technologies étudiées précédemment, voici le tableau récapitulatif reprenant, pour chaque critère défini au Chapitre 2, les valeurs idéales associées aux technologies. Ce tableau résulte simplement des remarques faites en fin de chaque chapitre de la Partie II sous le titre "Première évaluation".

Pour rappel, voici d'abord le tableau avec les différents critères et les abréviations utilisées pour chaque valeur correspondante.

Critères	Valeurs
Critères informationnels.	
Structure des données.	(I) Inexistante - (H) Hiérarchique - (E) Elevée { E(rel) pour "relationnelles" ou E(ss) pour "semi-structurées" }
Volume.	(P) Petit - (G) Gros
Fréquence des changements.	(F) Faible - (M) Moyenne - (E) Elevée
Sécurité.	(L) Accès libre - (E) Elevée
Système client.	
Configuration cliente.	(L) Légère - (M) Moyenne - (Lo) Lourde
Interface graphique.	(S) Simple - (M) Moyenne - (C) Complexe
Interactivité.	(F) Faible - (M) Moyenne - (E) Elevée
Système serveur.	
Configuration serveur(s).	(L) Légère - (M) Moyenne - (Lo) Lourde
Distribution des applications.	(C) Client - (S) Serveur - (CS) Client/Serveur - (M) Multiple serveurs
Distribution des BD.	(I) Inexistante - (E) Existante
Critère de conception.	
Compétences locales.	(F) Faible - (M) Moyenne - (E) Elevée

Nous pouvons maintenant proposer le tableau de synthèse des différentes technologies:

	HTML	DHTML	XML	JAVA	CGI	JAVASCRIPT	ASP
Critères informationnels.							
Structure des données.	I	E	H,E	E	E	E	E
Volume.	P	P,G	P	P,G	P,G	P,G	P,G
Fréquence des changements.	F	F,M,E	F,M	F,M,E	F,M,E	F,M,E	F,M,E
Sécurité.	L	L	L	L,E	L,E	L,E	L,E
Système client.							
Configuration cliente.	L	L*	L	L,M,Lo	L	L	L
Interface graphique.	S,M	S,M	S	S,M,C	S,M	S,M	S,M
Interactivité.	F	F,M,E	F	F,M,E	F,M,E	F,M,E	F,M,E
Système serveur.							
Configuration serveur.	L	L	L	L,M	L	L*	L*
Distribution des applications.	C	C	C	C,S,CS	S	S,CS	S
Distribution des BD.	I	I	E	E	I	E	I
Critère de conception.							
Compétences locales.	F,M,E	M,E	M,E	E	M,E	E	E

(*) = contraignante

Tableau 2: Synthèse des technologies suivant les valeurs des critères.

Pour les critiques générales concernant chaque technologie séparée, le lecteur pourra consulter le chapitre correspondant dans lequel chaque valeur des critères a été abordé (voir "Première évaluation" dans chacun des chapitre de la Partie II). Ci-dessous, nous rappelons brièvement les caractéristiques propres de chacune des technologies.

11.2. *HTML.*

Le *HTML* a été décrit au Chapitre 4.

Ce langage constitue actuellement le moyen le plus utilisé pour l'affichage des informations via l'Internet. La plupart des solutions technologiques décrites ici utilise le *HTML* pour l'interface avec l'utilisateur. Dès lors, quiconque désirant créer un site Web se verra obligé d'apprendre ce langage assez simple. La lecture du chapitre correspondant est donc fortement conseillée à toute personne n'ayant aucune connaissance en la matière.

On utilisera essentiellement le *HTML* pour tout document multimédia sans grande structure interne. Les hyperliens permettront de guider l'utilisateur vers l'information qui l'intéresse et cette information sera décomposée le plus possible en petit fichier pour éviter les temps de chargement trop long.

Le *HTML* seul ne convient pas pour la gestion des bases de données. Cette technologie sera donc utilisée en tant que support aux autres technologies, support essentiellement graphique (affichage) et de liaison (hyperliens).

11.3. *DHTML* et "Data Binding".

Le *DHTML* et le "Data Binding" ont été décrits au Chapitre 5.

Le *DHTML* est une version améliorée du *HTML*. On peut donc l'utiliser comme telle et se référer au point précédent pour une critique appropriée. Ce qui nous a intéressé particulièrement dans le *DHTML*, c'est le "Data Binding" offert par la version de "Microsoft". La première remarque concernera donc cette particularité assez contraignante: le "Data Binding" ne fonctionne que sur les navigateur "Internet Explorer" de "Microsoft".

Le plus apporté par cette technologie réside donc dans ses possibilités de gestion de bases de données, qu'elles soient "semi-structurées" (fichiers textes ou *XML*) ou même "relationnelles". Cette solution permet essentiellement la consultation de données mais également la mise à jour de ces informations. Ce langage ne requiert que des connaissances moyennes en informatique et est donc idéal pour les novices en la matière.

11.4. *XML*.

Le *XML* a été décrit au Chapitre 6.

Cette technologie permet la définition de langages similaires au *HTML*. Son but est de pouvoir structurer des documents destinés au Web. Le *XML* convient donc particulièrement aux informations structurées de manière hiérarchique (titre, sous-titre, paragraphe, etc.). On pourra ainsi créer des feuilles de style auxquelles tout nouveau document devra obéir afin de gérer l'information et son affichage de manière similaire pour toutes les publications d'un même serveur.

De plus, un langage de requête existe (*XML-QL*) ce qui ouvre la voie vers la gestion de bases de données "semi-structurées". Toutefois, seule la consultation est possible pour l'utilisateur final. Les mises à jour ne sont donc pas facilitées. Cette technologie possède cependant le grand avantage de pouvoir "questionner" des documents sur des serveurs différents. En effet, une requête peut combiner des données provenant de systèmes distants.

11.5. Java.

Le "Java" a été décrit au Chapitre 7.

"Java" combine des solutions "Client-Side" ("Applets") avec des applications "Server-Side" ("Servlets"). A l'aide d'une seule technologie, il est donc possible de créer un site "équilibré" où les applications sont partagées entre client et serveur. De plus, cette technologie est indépendante de la plate-forme. Tout programme "Java" peut donc être transféré d'une machine à l'autre, quel que soit le système d'exploitation utilisé.

Avec les "Servlets", la création dynamique des pages Web est possible. Ainsi, seule l'information nécessaire (sans code de programmation) est envoyée au client, ce qui permet un gain en temps de transfert. De plus, la gestion des accès au serveur Web est facilitée, ce qui permet de conserver des informations concernant un utilisateur connecté (par exemple en gardant une connexion à une base de données pendant tout le temps que l'utilisateur est connecté).

Dans sa version "Applet", "Java" offre une interface graphique digne des applications classiques créées avec "Delphi" ou "C++". Une fenêtre ("Applet") est insérée dans la page *HTML* et cette fenêtre peut accueillir tout objet graphique, des boutons aux menus en passant par les listes déroulantes et autres cases à cocher. Les "Applets" seront donc utilisées de préférence lorsque l'interface doit être complète plutôt que pour l'affichage de simples textes dans lesquels certaines données dynamiques proviennent d'une *BD*. Dans ce dernier cas, le recours à la génération dynamique des pages *HTML* sera plus adéquate via l'apport des "Servlets".

Bien que "Java" soit un outil très performant en ce qui concerne la gestion graphique au niveau de l'écran de l'utilisateur, c'est essentiellement son apport pour la gestion des *BD* qui nous intéresse ici. A ce point de vue, "Java" offre la possibilité de gérer des bases de données "semi-structurées" (gestion de fichiers) et "relationnelles" (via *JDBC*).

Les différentes architectures de "Java" offrent de grandes libertés quant à la connexion des *BD* "relationnelles". Ces *BD* peuvent être installées sur la machine serveur tout comme elles peuvent se trouver à distance de celle-ci. La distribution des informations est donc possible.

11.6. *CGI*.

Le *CGI* a été décrit au Chapitre 8.

Les "Scripts *CGI*" constituent une solution "Server-Side" permettant essentiellement la gestion des données provenant de formulaires *HTML*. Avec les informations fournies par l'internaute via les formulaires, la consultation de bases de données est possible, que se soit des *BD* "semi-structurées" (accès fichiers) ou "relationnelles". Toutefois, le *CGI* n'est pas adapté aux accès séquentiels puisqu'aucune technique n'est fournie pour gérer l'ensemble des appels d'un même utilisateur. Chaque nouvelle requête envers une *BD* doit donc forcément passer par une nouvelle connexion au *SGBD*, ce qui "coûte" cher en temps d'accès.

Le *CGI* peut constituer une solution rapide et assez facile pour toute personne n'ayant pas de connaissances approfondies en informatique puisque le choix du langage permet d'utiliser des langages faciles à apprendre.

11.7. JavaScript.

Le "JavaScript" a été décrit au Chapitre 9.

Le "Server-Side JavaScript" est une technologie de "Netscape" ne tournant que sur ses serveurs Web, ce qui peut constituer une contrainte dès le départ. Dans sa version "Client-Side", il est supporté par la plupart des navigateurs.

Tout comme "Java", le fait que cette technologie soit aussi bien "Client-Side" que "Server-Side" permet de créer des sites "équilibrés" où les applications sont partagées entre client et serveur. Toutefois, la version "Client-Side" n'offre aucune possibilité d'accès aux *BD*.

Un gros avantage du "Server-Side JavaScript" réside dans le fait que cette technologie est associée directement au serveur Web, ce qui lui permet de tirer pleinement profit des capacités de ce derniers. De plus, la gestion d'une session complète d'un utilisateur est possible, de même que la connexion entre plusieurs utilisateurs.

Les *BD* ne sont pas oubliées puisque l'accès aux principaux *SGBD* est offert ainsi que la possibilité de gérer les fichiers sur le système serveur. En ce qui concerne les *BD*, leur gestion est également possible via l'architecture "Three-Tier" ("à trois étages").

11.8. *ASP*.

Le *ASP* a été décrit au Chapitre 10.

Beaucoup de similarités existent avec la technologie "Server-Side JavaScript" mais il n'est présent qu'en version "Server-Side".

Le *ASP* est une technologie de "Microsoft" ne tournant que sur ses serveurs Web, ce qui peut constituer une contrainte dès le départ.

Un gros avantage du *ASP* réside dans le fait que cette technologie est associée directement au serveur Web, ce qui lui permet de tirer pleinement profit des capacités de ce derniers. De plus, la gestion d'une session complète d'un utilisateur est possible, de même que la connexion entre plusieurs utilisateurs.

Les *BD* ne sont pas oubliées puisque l'accès aux principaux *SGBD* (via *ODBC*) est offert ainsi que la possibilité de gérer les fichiers sur le système serveur. Les *BD* doivent cependant se trouver sur le disque local.

Chapitre 12. CHOIX TECHNOLOGIQUES.

12.1. Aide à la décision.

L'analyse du Tableau 2 va nous permettre de proposer des choix technologiques en fonctions des valeurs exigées pour chaque critère. Ainsi, il apparaît que certaines valeurs imposent directement une technologie alors que d'autres combinaisons de critères peuvent être résolues par plusieurs solutions technologiques. Pour mémoire, voici une copie du Tableau 2:

	HTML	DHTML	XML	JAVA	CGI	JAVASCRIPT	ASP
Critères informationnels.							
Structure des données.	I	E	H,E	E	E	E	E
Volume.	P	P,G	P	P,G	P,G	P,G	P,G
Fréquence des changements.	F	F,M,E	F,M	F,M,E	F,M,E	F,M,E	F,M,E
Sécurité.	L	L	L	L,E	L,E	L,E	L,E
Système client.							
Configuration cliente.	L	L*	L	L,M,Lo	L	L	L
Interface graphique.	S,M	S,M	S	S,M,C	S,M	S,M	S,M
Interactivité.	F	F,M,E	F	F,M,E	F,M,E	F,M,E	F,M,E
Système serveur.							
Configuration serveur.	L	L	L	L,M	L	L*	L*
Distribution des applications.	C	C	C	C,S,CS	S	S,CS	S
Distribution des BD.	I	I	E	E	I	E	I
Critère de conception.							
Compétences locales.	F,M,E	M,E	M,E	E	M,E	E	E

(*) = contraignante

Afin de faciliter la sélection d'une technologie pour un cas particulier, nous allons décomposer le problème en fonction des critères prédominants. Ces derniers permettent en effet une première sélection de technologie en fonction des valeurs associés. Une fois une première sélection opérée, les choix technologiques seront proposés en fonction des valeurs des autres critères: en effet, il apparaîtra que certaines valeurs ne pourront être atteinte que par une technologie unique. Dans ce cas, le choix sera évident. Pour les valeurs des critères étant couvertes par plusieurs solutions (nous les appellerons , "valeurs minimales"), les avantages et les inconvénients de chacune de ces technologies seront avancées afin de pouvoir les départager en fonction du souhait du lecteur.

La suite du chapitre va donc permettre une aide à la décision d'une technologie particulière en fonction des valeurs associées aux différents critères pour un projet déterminé. Ce projet ayant été évaluer par rapport aux critères [Chapitre 2], tout utilisateur pourra se diriger vers une solution en suivant la méthodologie qui va suivre.

12.2. Méthodologie.

La découpe du problème vers des choix technologiques est illustrée par l'arbre de décision suivant:

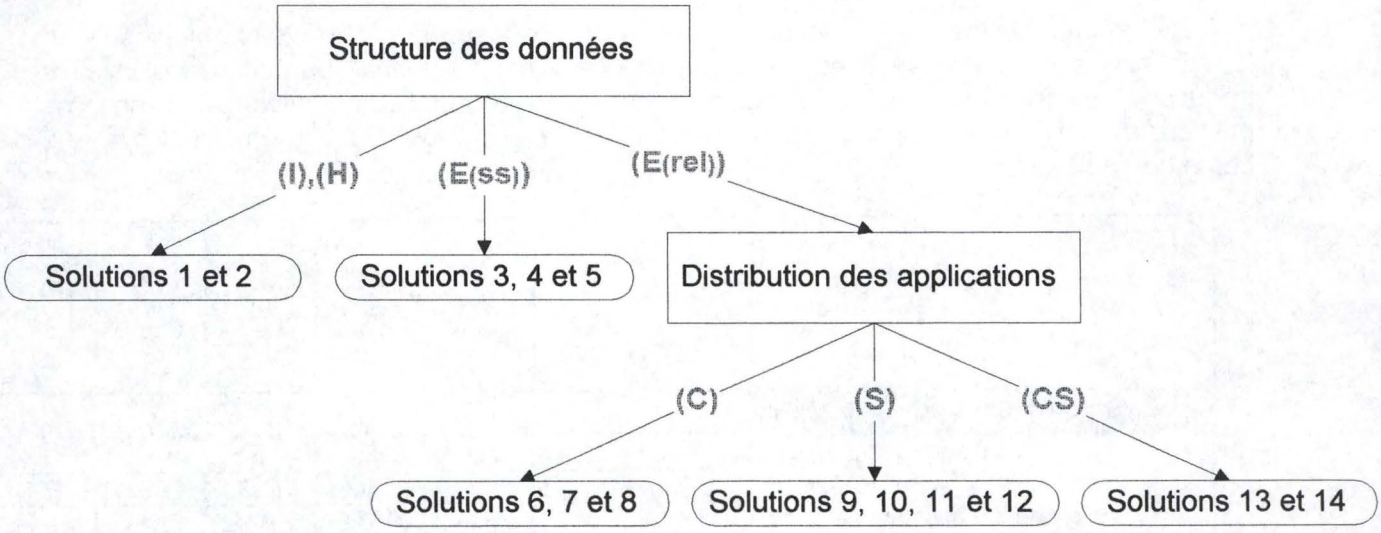

Figure 18: Arbre de décision.

La première découpe concernera le critère de *"Structure des données"*. Les solutions liées aux valeurs *Inexistantes* [I] et *Hiérarchiques* [H] mèneront directement vers les solutions 1 et 2. La valeur *BD "semi-structurées"* [E(ss)] sera quant à elle étudiée dans les solutions 3, 4 et 5.

Pour ce qui est de la valeur *BD "relationnelles"* [E(rel)], une deuxième découpe permettra de décomposer la recherche d'un choix technologique approprié. On se basera alors sur le critère *"Distribution des applications"*. Les applications *Client* [C] mèneront aux solutions 6, 7 et 8. Pour les applications *Serveur* [S], nous proposerons les solutions 9, 10 11 et 12. Et enfin, les solutions 13 et 14 répondront aux applications *Client/Serveur* [CS].

12.3. Choix technologiques.

Dans la plupart des cas, le futur concepteur sera dirigé vers une solution unique à son problème. Toutefois, nous devons émettre les deux remarques suivantes:

- certaines exigences envers un critère particulier peuvent mener vers une solution "forçant" les valeurs des autres critères. Par exemple, si l'on exige du critère A qu'il prenne la valeur X, il se peut qu'une seule solution ne permette pas de résoudre ce problème. Dès lors, si l'on désire également que le critère B prenne

la valeur Y mais que la technologie imposée ne supporte que la valeur Z pour ce critère, il est clair que c'est au concepteur de faire la part des choses en choisissant les critères primordiaux pour son application;

- d'un autre côté, si les exigences ne sont pas trop restrictives, il se peut que plusieurs solutions conviennent très bien au problème. Une analyse plus en détail des différents critères peut alors guider le concepteur vers une solution plutôt que l'autre. Afin de l'aider dans cette tâche, les grands avantages et inconvénients de chaque technique seront alors cités.

Pour commencer la décomposition, on peut observer dans le Tableau 2 que la *"Structure des données"* permet une première sélection de technologies. La première découpe se fera donc suivant les valeurs du critère précité:

- Structures inexistantes (I): correspond donc aux documents multimédia sans structure particulière: texte, images, vidéo, son, etc.;

- Structures hiérarchiques (H): ce sont les documents partageant une structure particulière décomposant l'information en petites parties plus facilement contrôlables. C'est le cas par exemple de la publication de livre sur le Net: une structure peut être définie pour que tout document partage les mêmes éléments (auteur, titre, sous-titre, paragraphe, note de bas de page, etc.). Une telle structure permet d'offrir une interface identique quel que soit le document affiché. On peut également ainsi pratiquer des recherches ciblées sur certains éléments de documents différents;

- Structures de bases de données (E): lorsque l'information est structurée de manière systématique en champs de type bien défini, on peut parler de base de données. On se souviendra de la distinction que nous avons fait au point 1.2 à propos de deux types spécifiques de bases de données: les *BD* "semi-structurées" et les *BD* "relationnelles". Nous proposerons ici cette même décomposition:

  BD "semi-structurées" (E(ss)): les informations sont contenues dans de simples fichiers de texte avec des délimitateurs spécifiques (voir 1.2.2);

  BD "relationnelles" (E(rel)): les informations sont gérées par des systèmes puissants permettant la lecture, l'écriture et la recherche d'information performantes (voir 1.2.1).

Nous regrouperons les deux premiers cas ensemble, ce qui nous permet la décomposition suivante:

- Point 12.3.1: Les documents multimédia et les structures hiérarchiques.;

- Point 12.3.2: Les *BD* "semi-structurées".;

- Point 12.3.3: Les *BD* "relationnelles"..

Remarque sur la lecture des tableaux qui vont suivre:

Les cas typiques seront décrits dans des tableaux dont les conventions utilisées sont les suivantes:

Chaque ligne représente un critère particulier;

Une colonne correspond à une combinaison particulière des valeurs menant à une ou plusieurs solutions technologiques;

Les valeurs dans chaque case correspondent aux abréviations utilisées dans le tableau des critères (Tableau 1);

Les cases ombrées représentent les valeurs décisives du cas correspondant, c'est-à-dire celles qui imposent un choix particulier;

Les cases avec les valeurs entre parenthèses signifient que ces valeurs découlent des choix ombrés. Ce sont donc les valeurs conseillées pour pouvoir utiliser la technologie correspondante.

En-dessous de chaque colonne, le numéro de la solution correspondante est notée; cette solution sera détaillée par après.

En fonctions des exigences principales, on se référera donc aux colonnes dont les cases ombrées contiennent la valeur voulue. Ensuite, il convient d'adapter le plus possible les autres valeurs en fonction des cases de cette colonne.

12.3.1. Les documents multimédia et les structures hiérarchiques.

Le tableau suivant nous montre les deux technologies adéquates respectivement pour les *"Structures de données" Inexistantes* (I) et *Hiérarchiques* (H). Le choix est donc immédiat en fonction de la première ligne. Les autres valeurs de critères sont alors forcées par la technologies (voir "Première évaluation" dans les chapitres correspondants: 4.8 et 6.6).

Critères informationnels:		
Structure des données.	I	H
Volume.	(P)	(P)
Fréquence des changements.	(F)	(F,M)
Sécurité.	(L)	(L)
Système client:		
Configuration cliente.	(L)	(L)
Interface graphique.	(S,M)	(S)
Interactivité.	(F)	(F)
Système serveur:		
Configuration serveur.	(L)	(L)
Distribution des applications.	(C)	(C)
Distribution des BD.	(I)	(I,E)
Critère de conception:		
Compétences locales.	(F,M,E)	(M,E)
Numéro de solution:	1	2

Tableau 3: Documents multimédia et structures hiérarchiques.

12.3.1.1. Solution 1: Documents multimédia.

La technologie la mieux adaptée aux documents multimédia correspond au ***HTML***. On l'utilisera donc pour créer des sites sans grandes structures et où les pages seront reliées entre elles par des hyperliens.

On notera également que le *HTML* constitue une technologie de support pour la majorité des autres solutions. Un minimum de connaissances dans ce domaine ne sera donc pas superflu.

12.3.1.2. Solution 2: Les structures hiérarchiques.

Lorsque les documents peuvent être décomposés de manière hiérarchique, le ***XML*** constitue une solution appropriée. Ainsi, titres, sous-titres, paragraphes, et autres éléments pourront être définis pour créer une structure unique pour tous les documents. Le *XML* fournira alors un moyen efficace de vérifier que la structure de tout document publié est identique.

12.3.2. Les *BD* "semi-structurées".

Le tableau suivant nous montre les deux technologies adéquates pour les *"Structures de données"* de type *BD* "semi-structurées" [E(ss)]. Il s'agit du ***DHTML*** et du ***XML***. Ces deux technologies fournissent en effet des moyens efficaces de gestion de fichiers structurés.

Certaines valeurs de critères exigent l'utilisation d'une des deux technologies précitées. Par contre, pour les valeurs "minimales" de ces critères, le choix entre les deux n'est pas direct et requiert donc une analyse plus approfondie (voir ci-après).

Les autres valeurs de critères sont alors forcées par la technologie (voir "Première évaluation" dans les chapitres correspondants: 5.3 et 6.6).

Critères informationnels.				
Structure des données.	E(ss)	E(ss)	E(ss)	E(ss)
Volume.	(P)	(P)	(P)	(P)
Fréquence des changements.	(F,M)	(F,M)	(F,M)	(F,M)
Sécurité.	(L)	(L)	(L)	(L)
Système client.				
Configuration cliente.	(L)	(L)	(L)	(L)
Interface graphique.	M	(S,M)	(S)	S
Interactivité.	(F,M,E)	M,E	(F)	F
Système serveur.				
Configuration serveur.	(L)	(L)	(L)	(L)
Distribution des applications.	(C)	(C)	(C)	(C)
Distribution des BD.	(I)	(I)	E	I
Critère de conception				
Compétences locales.	(M,E)	(M,E)	(M,E)	(M,E)
Numéro de solution:	3	3	4	5

Tableau 4: Bases de données "semi-structurées".

12.3.2.1. Solution 3.

Si l'"*Interface graphique*" doit être *Moyenne* ou/et que l'"*Interactivité*" doit être *Moyenne* ou *Elevée*, c'est vers le ***DHTML*** qu'il faudra se tourner.

12.3.2.2. Solution 4.

Si la *"Distribution des BD"* est recherchée, le ***XML*** fournit une solution adéquate. En effet, les requêtes peuvent porter sur des documents présents sur des machines différentes.

12.3.2.3. Solution 5.

Dans les autres cas, les deux solutions peuvent être envisageables.

On conseillera le ***DHTML*** pour les cas où la mise à jour directe est demandée (en effet, le *XML* ne permet que la consultation via Internet, les modifications doivent être faites manuellement). On se souviendra tout de même que cette solution nécessite l'utilisation d'un navigateur spécifique ("Internet Explorer"), ce qui est contraignant vis-à-vis de l'internaute.

Par contre, le ***XML*** constitue une solution adéquate pour des recherches simultanées sur plusieurs documents grâce au langage de requêtes *XML-QL*.

12.3.3. Les *BD* "relationnelles".

La majorité des technologies développées dans ce mémoire permettent la gestion des *BD* "relationnelles". Toutefois, en fonction des exigences, il est possible de proposer une solution plutôt qu'une autre.

La décomposition qui suit correspond aux différentes valeurs du critère *"Distribution des applications"*. En effet, une telle sélection permet de séparer les technologies et de simplifier ainsi la recherche d'un choix pour la gestion des *BD* "relationnelles".

Etant donné qu'aucune solution ne permet de gérer des applications sur plusieurs serveurs (valeur [MM] pour le critère *"Distribution des applications"*) nous utiliserons la décomposition suivante:

- Point 12.3.3.1: Applications "Client-Side". – valeur [C] pour le critère;
- Point 12.3.3.2: Applications "Server-Side". – valeur [S] pour le critère;
- Point 12.3.3.3: Applications "Client/Serveur". – valeur [CS] pour le critère.

12.3.3.1. Applications "Client-Side".

Si les applications doivent être 100% "Client-Side", deux solutions peuvent convenir: **JAVA** et ***DHTML***. Le tableau suivant nous le montre: *"Structures de données"* de type *BD* "relationnelle" (E(ss)) et *"Distribution des applications"* en mode *Client* (C).

Certaines valeurs de critères exigent l'utilisation d'une des deux technologies précitées. Par contre, pour les valeurs "minimales" de ces critères, le choix entre les deux n'est pas direct et requiert donc une analyse plus approfondie (voir ci-après).

Les autres valeurs de critères sont alors forcées par la technologie (voir "Première évaluation" dans les chapitres correspondants: 7.6 et 5.3).

Critères informationnels.							
Structure des données.	E(rel)	E(rel)	E(rel)	E(rel)	E(rel)	E(rel)	E(rel)
Volume.	(P,G)	(P,G)	(P,G)	(P,G)	(P,G)	(P,G)	(P,G)
Fréquence des changements.	(F,M,E)	(F,M,E)	(F,M,E)	(F,M,E)	(F,M,E)	(F,M,E)	(F,M,E)
Sécurité.	E	(L,E)	(L,E)	(L,E)	(L,E)	(L)	L
Système client.							
Configuration cliente.	(L,M,Lo)	M,Lo	(L,M,Lo)	(L,M,Lo)	(L,M,Lo)	(L)	L
Interface graphique.	(S,M,C)	(S,M,C)	C	(S,M,C)	(S,M,C)	(S,M)	S,M
Interactivité.	(F,M,E)	(F,M,E)	(F,M,E)	(F,M,E)	(F,M,E)	(F,M,E)	(F,M,E)
Système serveur.							
Configuration serveur.	(L,M)	(L,M)	(L,M)	M	(L,M)	(L)	L
Distribution des applications.	C	C	C	C	C	C	C
Distribution des BD.	(I,E)	(I,E)	(I,E)	(I,E)	E	(I)	I
Critère de conception.							
Compétences locales.	(E)	(E)	(E)	(E)	(E)	M	E
Numéro de solution.	6	6	6	6	6	7	6

Tableau 5: *BD* "relationnelles" en applications "Client-Side".

12.3.3.1.1. Solution 6.

Cette solution est donc conseillée lorsqu'au moins une des valeurs suivantes est désirée:

- *"Sécurité" élevée*,
- *"Configuration cliente" Moyenne ou Lourde*,
- *"Interface graphique" Complexe*,
- *"Configuration serveur" Moyenne*,
- *"Distribution des BD" Existante*,
- *"Compétences locales" Moyennes*;

La solution 6 correspond au **JAVA**. Cette technologie sera donc utilisée ici dans sa version "Client-Side", c'est-à-dire via les "**Applets Java**", puisque la *"Distribution des application"* doit être 100% *Cliente*.

12.3.3.1.2. Solution 7.

Si les *"Compétences locales"* ne sont pas *Elevées* mais *Moyennes*, le recours au ***DHTML*** devra être envisagé.

12.3.3.1.3. Solution 8.

Dans les autres cas, les deux solutions peuvent être envisageables: **JAVA** et ***DHTML***.

On se rappellera toutefois que le *DHTML* requiert un navigateur spécifique ("Internet Explorer"), ce qui est contraignant vis à vis de l'utilisateur. Par contre, JAVA permet une solution adaptée à n'importe quelle plate-forme

12.3.3.2. Applications "Server-Side".

Si les applications doivent être 100% "Server-Side", quatre solutions peuvent convenir: **JAVA**, ***CGI***, **JAVASCRIPT** et ***ASP***. Le tableau suivant nous le montre: *"Structures de données"* de type *BD* "relationnelle" (E(ss)) et *"Distribution des applications"* en mode *Serveur* (S).

Certaines valeurs de critères exigent l'utilisation d'une des quatre technologies précitées. Par contre, pour les valeurs "minimales" de ces critères, le choix entre les quatre n'est pas direct et requiert donc une analyse plus approfondie (voir ci-après).

Les autres valeurs de critères sont alors forcées par la technologie (voir "Première évaluation" dans les chapitres correspondants: 7.6, 8.8, 9.8 et 10.6).

Critères informationnels:						
Structure des données.	E(rel)	E(rel)	E(rel)	E(rel)	E(rel)	E(rel)
Volume.	(P,G)	(P,G)	(P,G)	(P,G)	(P,G)	(P,G)
Fréquence des changements.	(F,M,E)	(F,M,E)	(F,M,E)	(F,M,E)	(F,M,E)	(F,M,E)
Sécurité.	(L,E)	(L,E)	(L,E)	(L,E)	(L,E)	(L,E)
Système client:						
Configuration cliente.	M,Lo	(L,M,Lo)	(L,M,Lo)	(L,M,Lo)	(L)	L
Interface graphique.	(S,M,C)	C	(S,M,C)	(S,M,C)	(S,M)	S,M
Interactivité.	(F,M,E)	(F,M,E)	(F,M,E)	(F,M,E)	(F,M,E)	(F,M,E)
Système serveur:						
Configuration serveur.	(L,M)	(L,M)	M	(L,M)	(L)	L
Distribution des applications.	S	S	S	S	S	S
Distribution des BD.	(I,E)	(I,E)	(I,E)	E	(I)	I
Critère de conception:						
Compétences locales.	(E)	(E)	(E)	(E)	M	E
Numéro de solution:	9	9	9	10	11	12

Tableau 6: *BD* "relationnelles en applications "Server-Side".

12.3.3.2.1. Solution 9.

Cette solution est donc conseillée lorsqu'au moins une des valeurs suivantes est désirée:

- *"Configuration cliente" Moyenne ou Lourde*,
- *"Interface graphique" Complexe*,
- *"Configuration serveur" Moyenne*.

La solution 9 correspond au **JAVA**. Cette technologie sera donc utilisée ici dans sa version "Server-Side", c'est-à-dire via les "**Sevlets Java**", puisque la *"Distribution des application"* doit être 100% *Serveur*.

12.3.3.2.2. Solution 10.

Si la *"Distribution des BD"* est recherchée, le **JAVA** et le **JAVASCRIPT** peuvent convenir. On notera toutefois que "JavaScript" est plus "limité" pour certains critères:

- *"Configuration cliente" Légère*;
- Pas d'interface graphique *Complexe* possible;
- Une *"Configuration serveur" Légère* mais contraignante (utilisation du serveur de "Netscape").

Ces valeurs permettent donc de départager ces deux technologies. Dans le cas contraire, on choisira entre les deux suivant la discussion qui suit.

A l'avantage de JAVA, on notera son indépendance de la plate-forme. Par contre, le JavaScript en version "Server-Side" nécessitent un serveur Web spécifique ce qui peut être assez contraignant si le système en place utilise déjà un serveur différent. De plus, cela pose généralement des problèmes de "portabilité" des applications vers d'autres machines.

"Java" et "JavaScript" sont assez proches. Toutefois, le "JavaScript" est certainement un peu moins complexe que le "Java".

12.3.3.2.3. Solution 11.

Si les *"Compétences locales"* ne sont pas *Elevées* mais *Moyennes*, le recours au ***CGI*** devra être envisagé.

12.3.3.2.4. Solution 12.

Dans les autres cas, les deux solutions peuvent être envisageables: **JAVA** et ***CGI***. De plus deux autres solutions sont également acceptables: **JAVASCRIPT** et ***ASP***.

A l'avantage de JAVA, on notera son indépendance de la plate-forme. Par contre, le JavaScript en version "Server-Side" et le *ASP* nécessitent un serveur Web spécifique ce qui peut être assez contraignant si le système en place utilise déjà un serveur différent. De plus, cela pose généralement des problèmes de "portabilité" des applications vers d'autres machines.

On notera que le choix entre JavaScript et *ASP* résultera plus d'une préférence personnelle envers les produits "Netscape" ou "Microsoft" plutôt que d'un avantage spécifique d'un des deux produits. Ils sont en effet très similaires de par leurs fonctionnalités et leur utilisation. Ils conviendront tout deux spécialement pour la génération dynamique de pages *HTML* dont seulement quelques éléments proviennent d'une base de données (par exemple un document comprenant le prix d'un article).

La gestion des *BD* "relationnelles" via le *CGI* dépend essentiellement du langage utilisé ce qui veut dire que l'interface est différente d'un *SGBD* à l'autre. Ceci peut donc constituer un

désavantage si on utilise plusieurs types de *BD*. De plus, chaque appel à une application *CGI* résulte par une connexion à la *BD* et à la déconnexion en fin d'application. Comme il n'est pas possible de garder la trace d'un utilisateur, la connexion et la déconnexion sont irrémédiables. Au contraire, JAVA, JAVASCRIPT et ASP fournissent des moyens de conservation des informations concernant une session particulière d'un utilisateur. On peut même gérer plusieurs utilisateurs connectés à la même application.

12.3.3.3. Applications "Client/Serveur".

Si l'on désire profiter au mieux du système client ET du système serveur, deux solutions sont offertes: **JAVA** et **JAVASCRIPT**. Le tableau suivant nous le montre: *"Structures de données"* de type *BD* "relationnelle" (E(ss)) et *"Distribution des applications"* en mode *Client/Serveur* (CS).

Certaines valeurs de critères exigent l'utilisation d'une des deux technologies précitées. Par contre, pour les valeurs "minimales" de ces critères, le choix entre les deux n'est pas direct et requiert donc une analyse plus approfondie (voir ci-après).

Les autres valeurs de critères sont alors forcées par la technologie (voir "Première évaluation" dans les chapitres correspondants: 7.6 et 9.8).

Critères informationnels				
Structure des données.	E(rel)	E(rel)	E(rel)	E(rel)
Volume.	(P,G)	(P,G)	(P,G)	(P,G)
Fréquence des changements.	(F,M,E)	(F,M,E)	(F,M,E)	(F,M,E)
Sécurité.	(L,E)	(L,E)	(L,E)	(L,E)
Système client				
Configuration cliente.	M,Lo	(L,M,Lo)	(L,M,Lo)	L
Interface graphique.	(S,M,C)	C	(S,M,C)	S,M
Interactivité.	(F,M,E)	(F,M,E)	(F,M,E)	(F,M,E)
Système serveur				
Configuration serveur.	(L,M)	(L,M)	M	L
Distribution des applications.	CS	CS	CS	CS
Distribution des BD.	(I,E)	(I,E)	(I,E)	(I,E)
Critère de conception				
Compétences locales.	(E)	(E)	(E)	(E)
Numéro de solution	13	13	13	14

Tableau 7: *BD* "relationnelles" en applications "Client/Serveur".

12.3.3.3.1. Solution 13.

La solution 13 correspond au **JAVA**. Cette technologie sera donc utilisée ici dans sa version "Client-Side", c'est-à-dire via les "**Applets Java**" et dans sa version "Server-Side", c'est-à-dire via les "**Sevlets Java**".

12.3.3.3.2. Solution 14.

Dans les autres cas, deux solutions peuvent être envisageables: **JAVA** et **JAVASCRIPT**.

Ces deux technologies sont assez proches l'une de l'autre. Toutefois, dans sa version "Server-Side", le JavaScript nécessite un serveur Web spécifique ("Netscape"), ce qui peut être contraignant. Par contre, le "Client-Side JavaScript" est plus "léger" pour le navigateur

que les "Applets Java" puisque tous les objets et toutes les méthodes utilisables par "JavaScript" sont déjà installés avec le browser. Ceci constitue également le point faible du "Client-Side JavaScript" puisque ses actions sont ainsi limitées aux objets et aux méthodes prédéfinis.

On notera que la communication entre les deux technologies est possible (avec "LiveConnect"), ce qui constituerait peut-être la solution idéale: JavaScript pour ce qui ne nécessite pas d'objets ou de méthodes spécifiques et Java pour le reste.

Chapitre 13. CONCLUSION.

Si les solutions de mise à disposition de l'information via le Net sont multiples, nous avons pu montrer qu'il était possible de guider le concepteur vers un choix technologique en fonction de critères définis au Chapitre 2. A l'aide de ces critères et de l'analyse des différentes technologies, nous avons ainsi créé un arbre d'aide à la décision.

Les technologies que nous avons décrites dans la Partie II de ce document nous ont montré différentes façons d'utiliser le Web pour connecter des bases de données. Chaque solution a alors été évaluée en fonction des critères définis au préalable. A l'aide du tableau de synthèse ainsi obtenu au Chapitre 11, certains critères nous sont apparus comme étant plus décisifs que d'autres. Ainsi, nous avons décomposé le problème en plus petites parties afin de guider au fur et à mesure l'utilisateur vers une solution appropriée à son projet.

Ce mémoire fournit une synthèse des technologies Internet mais aussi un outil d'aide à la décision pour tout utilisateur désirant mettre de l'information à disposition via le Web. L'intérêt est donc de montrer les différentes possibilités offertes par les technologies tout en guidant le concepteur vers une solution particulière en fonction de ses exigences.

La difficulté principale de cette analyse réside dans la définition des critères de conception. En effet, il est difficile d'être exhaustif quant aux exigences du futur concepteur et celui-ci devra donc se limiter aux critères qui lui sont proposés. Il semble toutefois que la décomposition qui a été faite permet de couvrir la majorité des cas qui pourraient se présenter. S'il s'avérait qu'une dimension particulière venait à manquer, l'ajout d'un nouveau critère pourrait venir compléter l'analyse technologique. Des valeurs seraient proposées, les technologies seraient évaluées par rapport à celles-ci et la synthèse proposerait des solutions supplémentaires. De même, si une nouvelle technologie faisait son apparition, il serait aisé de rajouter un chapitre lui étant consacré et d'insérer cette nouvelle solution parmi la synthèse technologique.

Notons finalement une remarque sur l'ensemble des progiciels qui font leur apparition sur le marché de l'Internet. Je veux parler ici des applications offrant une interface de plus haut niveau pour la conception de sites Internet. On peut citer par exemple "NetDynamics", "Cold Fusion", "PowerSite", "Information Builder", etc. Toutes ces applications permettent l'utilisation de la majorité des technologies décrites dans ce mémoire mais sans en spécifier les avantages ni les inconvénients. Dès lors, l'utilisateur est aidé dans la conception mais les choix technologiques sont toujours laissés à sa propre évaluation. Ou alors, dans certains cas, le choix n'est pas possible et la technologie n'est alors peut être pas la plus adéquate. De tels outils peuvent donc être très utiles pour l'interface qu'ils offrent mais le recours à une aide à la décision, telle que celle qui est offerte ici, sera toujours la bienvenue.

BIBLIOGRAPHIE.

Livres et articles.

[1] George Reese, "*Database Programming with JDBC and JAVA*", O'Reilly & Associates, United States of America, June 1997.

[2] Jean-Luc Hainaut, "*Bases de données et modèles de calcul - Outils et méthodes pour l'utilisateur*", InterEditions, Paris, 1994.

[3] Jason Hunter et William Crawford, "*JAVA Servlet Programming*", O'Reilly & Associates, United States of America, Octobre 1998.

[4] Paul Enfield, "*Implementing a secure site with ASP*", Microsoft Corporation, 17/07/1998

[5] Neil Randall, "*J'utilise HTML*", Simon & Schuster Macmillan, Paris, France, 1996.

[6] David Flanagan, "*JavaScript, The Definitive Guide - Third Edition*", O'Reilly & Associates, United States of America, 1998.

[7] Neil Randall, "*J'utilise HTML*", Simon & Schuster Macmillan, Paris, France, 1996.

[8] Peter Flynn, "Understanding SGML and XML tools", Kluwer Academic Publishers, United States of America, 1998.

[9] George Reese, "*Database Programming with JDBC and JAVA*", O'Reilly & Associates, United States of America, June 1997.

[10] Jason Hunter et William Crawford, "*JAVA Servlet Programming*", O'Reilly & Associates, United States of America, Octobre 1998.

[11] Antoine Mirecourt, "*Le développeur Java 2 - Edition 1999*", Osman Eyrolles Multimedia, Paris, France, 1999.

[12] Graham Hamilton & Rick Cattell, *"JDBC : A Java SQL API – version 1.20"*, Javasoft, January 10, 1997.

[13] Ph. Thiran, O. Demoulin, *"Projet RW n°3062, Inter-DB : Interopérabilité des Systèmes d'Information hétérogènes et distribués – JDBC"*, Facultés Universitaires Notre-Dame de la Paix, Institut d'informatique, Namur, Belgique, 4 février 1998.

[14] Bradley F. Burton and Victor W. Marek, *"Applications of JAVA programming language to database management"*, Department of Computer Science, University of Kentucky, Lexington.

[15] Anup K. Ghosh, "*E-Commerce Security - Weak links, Best defenses*", Wiley Computer Publishing, 1998.

[16] Kristof Vermeire, "*Special Java - Le lien entre serveurs et surfeurs*", Computer Magazine n°94/95, pp. 73-75, Juillet-Août 1999.

[17] Ian Graham, *"Introduction to HTTP and CGI"*, Instructional and Research Computing, University of Toronto.

[18] Shishir Gundavaram, "*CGI Programming on the World Wide Web - First Edition*", O'Reilly & Associates, Mars 1996.

Adresses Internet.

{1} "Un Gouveau Guide Internet - UNGI" – Chap. 49: Quelques "plug-ins" Netscape: http://ungi.cge.net/cyber/

{2} "Server-Side JavaScript" - Writing Server-Side JavaScript Applications: http://developer.netscape.com/docs/manuals/enterprise/wrijsap/index.html

{3} "Un Nouveau Guide Internet" – Pour utilisateurs et concepteurs: http://ungi.cge.net/cyber/ (chap. 41 à 46: "JavaScript")
{4} Source programmes "JavaScript": http://www.developer.com
{5} Source programmes "JavaScript": http://www.essex1.com/people/timothy/js-index.htm
{6} Source programmes "JavaScript": http://www.dynamicdrive.com
{7} Spécifications officielles des "cookies *HTTP*": http://www.netscape.com/newsref/std/cookie_spec.html
{8} "Un Nouveau Guide Internet" – Pour utilisateurs et concepteurs: http://ungi.cge.net/cyber/ (chap. 29: *HTML* 4.0)
{9} "Dynamic HTML" – Tout savoir: http://msdn.microsoft.com/developer/sdk/inetsdk/help/dhtml/dhtml.htm
{10} "Dynamic Drive DHTML (Dynamic HTML) Code Library!" – Répertoire de scripts DHTML: http://www.dynamicdrive.com
{11} "Data Binding" – Le *DHTML* et les *BD*: http://msdn.microsoft.com/developer/sdk/inetsdk/help/dhtml/content/data_binding.htm#dhtml_databind
{12} "Remote Data Service" – Microsoft: http://msdn.microsoft.com/developer/sdk/inetsdk/help/rds/gs01.htm
{13} "The Annotated XML Specification" – site du W3C: http://www.xml.com/axml/testaxml.htm
{14} http://www.tetrasys.fr/
{15} "XML: faq" – Forum aux questions: http://www.ucc.ie/xml/
{16} "The SGML/XML Web Page" – Robin Cover: http://www.oasis-open.org/cover/xml.html
{17} SGML: http://www.sil.org/sgml
{18} "XML DTD for recipes" – Exemple de DTD (pour des recettes de cuisine): http://dessert.home.mindspring.com
{19} "XML-QL: A Query Language for XML": http://www.w3.org/TR/NOTE-xml-ql/
{20} "The Java Development Kit" – Site officiel (SUN): http://java.sun.com/products/jdk/index.html
{21} "JavaMed" – Tout savoir sur Java (en français): http://www.med.univ-rennes1.fr/~courtin/JavaMed/MultimediaJava.html
{22} "Visual J++" – Version 6.0 (Microsoft): http://www.asia.microsoft.com/visualj/featurepeek/
{23} "Academic Program for Java" (IBM): http://www.ibm.com/java/academic/
{24} "The JDBC database access API" – Site officiel (SUN): http://java.sun.com/products/jdbc/
{25} "JDBC documentation" (SUN): http://java.sun.com/products/jdk/1.1/docs/guide/jdbc/index.html
{26} "DevEdge Online" – Site de Netscape: http://developer.netscape.com/docs/technote/index.html?content=database/web/index.html
{27} "The Java Language Environment ; A white paper" (SUN): http://java.sun.com/docs/white/langenv/
{28} "Un Nouveau Guide Internet" – Pour utilisateurs et concepteurs: http://ungi.cge.net/cyber/ (chap. 36 à 40)
{29} "Big Nose Bird" – Introduction au *CGI*: http://www.bignosebird.com/
{30} "Publier sur le World Wide Web" – section *CGI* : http://fp.planete.net/Publier/
{31} "Munica Web Database Solution Provider" – création d'un site avec gestion de *BD* à partir d'un logiciel payant: http://www.munica.com/dp-tutor.htm
{32} "Guide de programmation HTML" – Aide sur *CGI*: http://www.webdeveloppeur.com/Conception/introductionCGI.html

{33} "The Common Gateway Interface" – Page de la *NCSA*:
http://hoohoo.ncsa.uiuc.edu/cgi/overview.html
{34} "CGI Made Really Easy" – *CGI* et formulaires: http://www.jmarshall.com/easy/cgi/
{35} Sites *CGI* de "Yahoo.com":
http://www.yahoo.com/Computers_and_Internet/Internet/World_Wide_Web/CGI___Common_Gateway_Interface/
{36} "Server-Side JavaScript" – Writing Server-Side JavaScript Applications:
http://developer.netscape.com/docs/manuals/enterprise/wrijsap/index.html
{37} "MSDN online - Web Workshop" – An ASP you can grasp: the ABCs of Active Server Pages: http://msdn.microsoft.com/workshop/server/asp/ASPover.asp
{38} "MSDN online - Web Workshop" – ASP from A to Z:
http://msdn.microsoft.com/workshop/server/asp/aspatoz.asp
{39} "MSDN online - Web Workshop" – ASP Technology feature overview:
http://msdn.microsoft.com/workshop/server/asp/aspfeat.asp
{40} "MSDN online - Web Workshop" – ASP Tom: Server Q & A:
http://msdn.microsoft.com/workshop/server/feature/morqa.asp