Web dynamique - interfaçage SGBDR

Application à PHP et Mysql

John GALLET - SaphirTech

http://www.saphirtech.com

**Rédaction initiale : août 2003**
**Dernière révision : juin 2005**



**NB : un document spécifique traitant exclusivement de la sécurisation des applications web est également disponible.**

# 1. Introduction et audience

## *1.1 Présentation*

Ce cours est principalement destiné aux étudiants de la License Professionnelle "Les métiers de l'internet" de l'Université de Paris V Renée Descartes, et s'inscrit à ce titre dans une progression pédagogique maîtrisée.

Les pré-requis pour comprendre ce cours sont :
- langage html basique
- théorie des SGBDR et langage SQL
- notions de POO
- notions de réseaux, en particulier tcp/ip
- algorithmique connue. Il n'y aura pas de rappels de programmation pure.

Les rappels sur les SGBDR présents dans ce cours ne sauraient se substituer à un cours structuré et complet, ils ne sont que des compléments nécessaires pour la mise en pratique.

Ce cours récapitule les principales difficultés auxquelles sont confrontés les développeurs sur un protocole asynchrone non connecté en environnement hautement concurrentiel et propose des solutions appliquées à PHP et MySQL aisément généralisables à d'autres langages (asp, .net, jsp etc...) et d'autres véritables SGBD relationnels (pour lesquels beaucoup de problèmes inhérents aux lacunes de MySQL 3 & 4.0.x disparaissent). Ce cours exclue totalement l'utilisation d'applets java exécutées dans le navigateur.

Un certain nombre de remarques vont à l'encontre de certaines tendances actuelles et pourront faire bondir, à la lecture ou plus tard, certains lecteurs. L'essentiel est de garder un esprit critique et de comprendre les avantages, inconvénients et limites de ce que l'on fait. L'auteur n'entend pas convertir ou endoctriner qui que ce soit mais seulement faire réfléchir. Libre à chacun de choisir la méthode qui lui conviendra le mieux mais surtout qui sera la plus adaptée aux besoins de l'application à réaliser.

Remarque : le html donné dans les exemples pourrait être plus conforme à de soit-disant standards parfois compréhensibles de certains navigateurs. Bien que non "politiquement correct" celui-ci est 100% fonctionnel sur 100% des navigateurs au delà de Mosaïc ou Netscape et IE 2.x (parus en 1997) qui avaient parfois du mal avec les tableaux. Ce cours n'est pas un cours de html, et ne prétend donner aucune leçon dans ce domaine.

## 1.2 Plan général du cours

Le chapitre 2 présente les "acteurs informatiques". Le chapitre 3 recense les difficultés à résoudre dans un environnement classique de web dynamique. Pour aider à la mise en pratique, des rappels de syntaxe de PHP seront brièvement indiqués au chapitre 4. On s'intéressera ensuite au dialogue entre le client et le moteur dynamique au chapitre 5, à la gestion des données externes dans le moteur dynamique (formulaires) dans le chapitre 6. Après un bilan rapide au chapitre 7, le chapitre 8 donne des rappels fondamentaux sur le SGBDR qui permettront d'attaquer les structures de données. Le chapitre 9 détaille le dialogue client-serveur entre le moteur dynamique et le serveur SGBDR. On a alors vu le trajet complet des données pour la génération dynamique de quelques pages que rien ne relie logiquement, et il est temps de voir les sessions dans le chapitre 10. Les chapitres 11 et 12 traitent des accès concurrentiels. Le chapitre 13 aborde la notion de templates (séparation logique/présentation). Une introduction aux regexps est proposée au chapitre 14. Enfin, le chapitre 15 recense les fonctionnalités les plus utilisées de la plateforme PHP.

## 1.3 Objectifs du cours

En fin de module, qui est nécessairement encadré et complété par des TD/TP de mise en application, l'étudiant devrait :
- connaître les principes et problématiques communs à tous les environnements techniques de applications web
- être capable de développer des applications simples sur une plateforme PHP-Mysql ou PHP-Postgresql.

# 2. Les acteurs

## 2.1 Le client

Le plus souvent un navigateur (IE, Netscape, Mozilla, Lynx, etc…) utilisé par un internaute. C'est l'interface de saisie et de présentation.

## 2.2 Le serveur web ou httpd

Il est à l'écoute sur un port bien déterminé (par convention le port 80 ou 8080 si non spécifié dans l'URL). Il accepte des requêtes entrantes et en renvoie le résultat. On peut citer comme daemons httpd les logiciels suivants : apache, Caudium, NES (Netscape Enterprise Server), IIS, Zeus...

## 2.3 Le protocole

Hyper Text Transfert Protocole ou http sur tcp/ip. Il est asynchrone et non connecté. On envoie une requête dans l'hyper-espace virtuel d'internet et on espère recevoir la réponse un jour. De même, la réponse nous est envoyée en espérant qu'on la reçoive. Elle est coupée en autant de petits morceaux que nécessaire si elle est trop volumineuse pour être renvoyée dans une seule trame. Il n'y a pas de liaison réellement établie spécifiquement pour l'échange au même sens que lors d'une communication

téléphonique (ce serait comme si on raccrochait après chaque phrase pour renuméroter). Il n'y a pas de mise à jour automatique possible de la part du serveur en cas d'information périmée, il faut que le client re-demande pour savoir (get et non pas push). Ceci va poser quelques problèmes.

## 2.4 Le SGBDR

Il est lui aussi à l'écoute, sur le réseau local. Le client ne peut pas lui parler directement, pour raisons de sécurité d'une part, de complexité et d'abstraction d'autre part, sauf dans des architectures spécifiques comme le Web Application Server d'Oracle où la limite des différents composants est plus ténue. On peut citer : DB2, Oracle, Sybase, MS-SQLSERVER, RDB, Informix, Ingres, PostgreSQL, Mysql.

## 2.4 Le moteur dynamique

PHP ou PERL ou ASP ou JSP ou .Net etc… Il s'intègre plus ou moins profondément avec le serveur http et permet de faire la liaison entre la demande du client avec ses arguments, l'accès au SGBDR, les traitements nécessaires, et la génération du rendu visuel en retour (présentation). Remarque : CGI n'est pas un langage mais une définition d'interface (on fait le plus souvent du cgi en PERL ou en C ).

## 2.6 Architecture globale

On est donc dans une architecture à quatre tiers. Profitons en pour rappeler que dans ce cadre le terme tiers n'a rien à voir avec la division par 3 mais est lié à la racine "tierce personne" et veut donc dire "nombre de parties". Le passage par le serveur web étant totalement transparent en termes de codage, on peut se ramener à un modèle trois tiers ultra-classique :
- la couche présentation (assurée par le navigateur)
- le stokage rémanent (en SGBDR)
- le "middle tiers" qui comme son nom l'indique fait le lien entre les deux.

Cette vision est importante car elle devrait toujours être conservée dans le codage en séparant, autant que faire se peut, le traitement des données (et en particulier l'interaction avec le SGBDR) et la génération du code html de retour (qui devrait se limiter au remplissage des zones réellement dynamiques). Ce modèle de développement n'est pas si complexe que ça à mettre en oeuvre, et sera abordé en fin de cours. Dans un premier temps, tout sera allègrement mélangé dans un but volontaire de simplification.

## 2.7 VERIFICATION DE LA COMPREHENSION

- Combien de machines différentes peuvent entrer en compte ?
- quels logiciels sont installés dessus ?
- sur quels protocoles communiquent-elles ?

Réponses :

Il y a au moins deux machines en action : celle de l'internaute, cliente, et un serveur web. Celle de l'internaute n'a besoin que d'un navigateur (Internet Explorer, Mozilla, etc...). Le serveur doit disposer d'un serveur http (apache, IIS...) et d'une base de données (Oracle, Sybase, MySQL...). Souvent, on dédie une machine à la base de données, appelée serveur SGBD, ce qui porte à trois le nombre de machines différentes. La machine de l'internaute communique avec le serveur web sur HTTP. Le serveur web et le serveur SGBD communiquent sur TCP/IP dans un protocole propriétaire à la base de données (et qui n'est donc pas le même selon la base). On ajoute souvent à cette architecture deux

firewalls (de marque différente, sinon les failles sont communes !), l'un devant la machine web, lautre entre la machine web et la machine SGBDR.

# 3. Principales définitions, règles et contraintes

## *3.1 Fonctionnement*

Demande d'une page statique :
Le client (navigateur) émet une requête de type GET HTTP page.html
Le serveur httpd, sur réception de cette requête, vérifie qu'il possède bien cette page sur le disque du serveur, et la renvoie en l'état. Le client recevra donc du html. Sans modification physique du fichier sur le serveur web, le flux renvoyé au client restera le même. Cette page peut néanmoins contenir des scripts exécutés côté client (javascript, aspx) qui en modifient l'apparence visuelle au cours de requêtes successives (par exemple : affichage de l'heure locale, ce qui est parfois d'une utilité douteuse, mais un bon exemple).

Demande d'une page dynamique :
Le client (navigateur) émet une requête de type GET HTTP page.php
Le serveur httpd, sur réception de cette requête, vérifie qu'il possède bien cette page sur le disque du serveur, mais en comparant l'extension de la page avec ses tables de configuration internes, s'aperçoit qu'il doit alors passer la main à un programme autre (en l'occurrence, le moteur Zend de PHP). Charge à PHP de construire le flux de retour selon le code du programme PHP. Il s'agit là aussi souvent de construire dynamiquement du flux HTML. Ce flux de retour est alors renvoyé au serveur httpd qui le retransmet au client (navigateur). Le client recevra donc là aussi du html (et éventuellement JS etc...)

Il est donc important de noter que le client (navigateur) n'a absolument aucun moyen de savoir comment a été générée la page qui lui revient. Même une analyse de l'extension du fichier ne saurait être prise pour argent comptant :
- rien n'empêche d'associer l'extension .html à php
- rien n'empêche d'associer l'extension .asp à … php
- rien n'empêche d'associer n'importe quelle extension à n'importe quoi (.toto à php)

Le client (navigateur) se contente d'afficher les données qu'il reçoit (et éventuellement d'exécuter du JS localement).

ATTENTION DANGER : une habitude stupide et dangereuse se répend comme une traînée de poudre au sein des "communautés" de développeurs qui ne comprennent rien à ce qu'ils font, à savoir le ".inc" partout. "config.inc" ou "global.inc" sont des noms courants et DANGEREUX. En effet, il suffit d'appeler ces fichiers directement dans l'URL (http://www.....com/config.inc) et ils seront vus comme des PAGES STATIQUES et donc tout leur code (et les mots de passe qui y sont déclarés...). NE METTEZ JAMAIS L'EXTENSION .inc à un fichier.

## *3.2 Indépendance des requêtes http*

Quand le serveur http reçoit une requête, les trames tpc/ip qui la composent comportent une adresse IP source qui servira à renvoyer la réponse en retour. Il est à noter que cette adresse IP source est écrasée de proche en proche par les différents proxy qu'elles ont traversé. Autrement dit, et ceci est une contrainte imposée par le fonctionnement même du protocole :

- il est impossible de savoir avec certitude quelle est l'adresse IP d'origine d'une requête. Ceci est encore plus vrai si l'on considère les réseaux locaux avec une seule IP publique : il y a un nombre astronomique d'ordinateurs qui ont pour IP du 192.168.xxx.xxx

- en particulier pour cette raison, et parce que le protocole n'a pas prévu de mécanisme complémentaire, il est impossible de détecter deux requêtes successives provenant de la même machine cliente uniquement grâce au protocole http : il faudra utiliser un mécanisme de sessions applicatives.

Sauf en réseau de type intranet/vpn (et encore, avec des restrictions) toute identification reposant sur des adresses IP doit être bannie ou utilisée avec précautions. Il sera nécessaire de gérer applicativement la reconnaissance de deux requêtes successives si l'on veut tracer la navigation de l'internaute.

## *3.3 Péremption de l'information*

Avant même qu'elle ne soit lue par l'internaute, l'information est potentiellement déjà périmée.

Deux cas classiques illustrent ce problème :

- les informations de nature nativement hautement volatile comme les cours de bourse par exemple. Demande client t0. Arrivée sur le serveur http : t1. Interrogation du serveurs 'temps réel' t2. Retour de l'information : t3. Renvoit vers le client : t4 (information déjà potentiellement périmée). Retour au client au travers du réseau : t5, information probablement déjà périmée sur une valeur un peu volatile.

- l'accès concurentiel dans le temps à une même donnée, typiquement une gestion de stock de marchandises vendues en ligne. Le premier client A et le second client B demandent, à quelques secondes d'intervalle ou ensemble, la quantité de produit Z restante. Ils reçoivent tous les deux "10 unités". Le client A en commande 7, valide sa commande. La base de données est mise à jour et il reste donc 3 unités. Mais sur l'écran du client B, il est toujours indiqué "10 unités". Si le client B en commande alors plus de 3, on va arriver à un stock négatif. Ceci peut dans certains cas n'avoir aucune gravité pour des produits de consommation courante, mais est totalement intolérable dans le cas de pièces uniques (par exemple pour un antiquaire : la commode Louis XV, il en a une à vendre, pas deux, et s'il la vend deux fois, les procès en justice ne vont pas tarder !).

## *3.4 Sécurité = confidentialité + intégrité + service*

Confidentialité
On peut chiffrer des données qui transitent (mot de passe par exemple). Mais ceci a un coût en calcul sur le serveur comme le client (20 à 30% de CPU consommé en plus). Il est donc de la responsabilité du développeur de limiter les échanges de données chiffrées. Par exemple, au lieu de systématiquement faire transiter un couple (login/password) on va le demander une seule fois et attribuer un jeton (identifiant de session) avec une durée de vie limitée dans le temps. Celui-ci pourra sans risques transiter en clair s'il n'est pas aisément prévisible.

Intégrité des données de la base
Les SGBDR ont été conçus dans le but de garantir la cohérence des données. C'est leur rôle et leur raison d'être : un SGBD permet de stocker des données de manière ordonnée et cohérente. Tout ce qui peut être assuré par le SGBDR devra donc lui être confié. Par exemple, il est de manière générale anormal de trouver du code applicatif se portant garant de l'unicité des données : c'est le rôle d'une clef unique/primaire. Voir à ce sujet les rappels sur les SGBDR dans ce cours.

Service
Il est des cas où l'accès au site internet doit être garantit aux internautes 24h/24 (en particulier quand ils payent ce service) et avec des temps de réponse bornés. Par exemple: ordres de bourse en ligne. Le développeur doit donc s'assurer que son code n'induira ou ne facilitera pas de DoS (Denial Of Service), crash, surcharges de source interne/externe, et globalement, que ses performances sont correctes.

## 3.5 Portabilité du code

Un code est dit portable si on peut l'installer sans modifications dans une autre configuration similaire.

Il y a trois principales contraintes à la portabilité de code PHP :

- le système d'exploitation (Operating System) (windows, unix etc…). Sauf utilisation de fonctions très spécifiques (et clairement indiquées comme telles dans le manuel de PHP) ou utilisation sauvage de chemins d'accès au file system clairement spécifiques à un OS ("c:\…" par exemple), le code PHP est indépendant de la plateforme sur laquelle il tourne.

- le SGBDR utilisé. Celui-ci pose deux problèmes distincts. D'une part les fonctions PHP qui sont utilisées pour le dialogue avec ce SGBDR. Par exemple, mysql_connect () ou sybase_connect(). Ceci est relativement simple à gérer par une couche d'abstraction, qu'elle soit objet ou reposant sur des systèmes d'include/require. D'autre part les requêtes SQL en elles-mêmes, et en particulier les jointures ou les clauses spécifiques (LIMIT par exemple). Il est complexe d'écrire du code SQL portable si on ne connait pas à l'avance les SGBDR auxquels devra se connecter l'application, et on doit souvant faire de la presque duplication de code. Le respect de standards comme SQL 9.2 n'apporte pas toujours la solution (comme tous les standards ou normes d'ailleurs). Exemple de couche d'abstraction : ADODB (http://adodb.sourceforge.net/)

- le langage lui même ou certaines de ses extensions. Il convient avant de faire un nouveau développement avec un ensemble de fonctions que l'on a jamais utilisées de vérifier l'état du développement. Certaines extensions sont clairement indiquées dans le manuel comme "expérimentales" (et donc sujettes à évolutions futures, par exemple au moment de la rédaction initiale de ce document, Dom-XML, qui était entre les versions 4 et 5 de php). Par ailleurs, le langage lui même évolue, PHP 4 a apporté de nombreuses modifications de comportement par rapport à PHP3 et PHP5 apporte un modèle objet plus complet (ainsi que des modifications dans la gestion des variables par références).

## 3.6 Le projet

Trop de développeurs travaillent en "couches" successives qui sont plus ou moins bien réparties selon les aléas des délais à respecter (et donc de moins en moins complètes au fur et à mesure de l'arrivée de l'échéance), alors qu'il serait simple de prévoir dès le départ le code adapté.
Ainsi, un bon codage prend en compte dès le départ :

- le respect du cahier des charges.
Celui-ci est primordial et à double tranchant. Vous devez développer les fonctionnalités demandées, toutes les fonctionnalités demandées, mais seulement les fonctionnalités demandées. Tout le reste n'a pas été vendu au client, donc ne sera pas facturé et empiêtra sur la marge dégagée, voire sur la proposition commerciale émise pour une deuxième version déjà en négociation. Si vous avez un doute sur un besoin qui vous semble primordial mais n'est pas cité dans le cahier de charges, vous devez poser la question à qui de droit !

- la gestion des erreurs, la reprise sur erreur.
Quand l'internaute reçoit le message d'erreur "Invalid Mysql Index at line 9 in /usr/toto/lib.php" que va-t-il penser de l'applicatif ? Manque de fiabilité. Que va-t-il faire du message d'erreur ? Rien. Que va-t-il faire tout court étant donné qu'il est dans une "impasse de navigation" ? Au mieux recommencer en cliquant sur "retour arrière", au pire… fermer le navigateur et vous avez perdu un client potentiel.
Il convient donc de tester systématiquement le retour de toutes les fonctions afin de détecter et de gérer les erreurs selon leur gravité, en orientant l'internaute et en lui présentant des excuses. Si votre programme s'est planté lamentablement, c'est le minimum que vous lui devez, ou il ira voir ailleurs.

- performances.

Ce n'est pas une fois qu'on a fait tout un schéma de SGBDR et tout un développement qu'on se demande si on aura la réponse dans des temps acceptables. Il faut dès le départ se demander si la solution retenue pourra tenir la charge. De la même manière, le code doit être écrit dès le départ en pensant aux temps de réponse : ne pas faire de traitements inutiles, toujours évacuer les cas d'erreur en premier au lieu de lancer un traitement qui n'a aucune chance de réussir, ne pas parcourir inutilement des tableaux, ne pas transmettre des variables entre le navigateur et le serveur pour rien etc…

## *3.7 Techniques de débugage*

Avant tout : réfléchir. Il ne sert à rien de se jeter sur le code. Ne pas avoir d'a priori, surtout en ce qui concerne les temps d'accès aux SGBDR. Pour débugguer des problèmes de performances, toujours commencer par déterminer réellement et en vérifiant sur plusieurs itérations le temps pris par tous les traitements en cause et s'attaquer, évidement, aux plus longs. Ne pas oublier que les traces peuvent être ajoutées en commentaires HTML, non visibles directement dans le navigateur, au moins temporairement. Le mieux est bien entendu d'avoir une fonction toute prête et d'activer à volonté un niveau de trace incluant nativement des informations sur les temps de traitement.

## *3.8 Méthode de développement*

De nombreuses méthodes de développement existent, il n'est pas possible de toutes les détailler. Celle qui est proposée ici est plus proche de la "checklist" que d'une véritable "méthôôôde", mais elle permet d'avoir un guide général contre l'angoisse de la feuille blanche du développeur :

1) Définir les écrans et leurs fonctionnalités . Cette phase est parfois faite par un infographiste et elle représente l'expression visuelle d'une partie du Cahier des Charges.

2) Définir le schéma du SGBDR permettant de modéliser l'univers lié à l'application. Ceci est parfois fait par un architecte de bases de données.

3) Définir les données qui transitent entre les écrans. Cette phase très importante est trop souvent laissée à la découverte du développeur "au fil de l'eau" ou plutôt "au petit bonheur la chance".

4 ) Définir leur traitement côté serveur (algorithmes).

5) Ecrire les requêtes SQL correspondantes en se servant intelligement des contraintes d'intégrité, et dénormaliser le schéma du SGBDR si nécessaire.

6) implémenter les algorithmes dans le langage choisi, en utilisant de la POO ou de la programmation procédurale, en PHP, en ASP/.NET/C#, en JSP, en PERL...

## *3.9 VERIFICATION DE LA COMPREHENSION*

- comment faire la différence entre les données reçues d'une page statique et les données reçues d'une page dynamique ?
- le protocole http permet-il de différencier les internautes au cours de leur navigation ?
- même sans se connaître, deux internautes peuvent-ils se gêner mutuellement dans leur navigation ?
- appartient-il au code PHP/JSP/ASP/.NET de vérifier si une donnée est déjà présente en SGBD avant de l'y insérer de nouveau ?
- quelle est la première action à mener quand les temps de réponse sont mauvais ?

Réponses:
- On ne peut pas avec certitude faire la différence entre des données générées dynamiquement et des données venant d'une page statique. Par convention, .html est une page statique, mais il ne s'agit que d'une convention et non d'une obligaton technique.
- Non, le protocole http n'offre aucune notion de session utilisateur.
- deux internautes qui accèdent à la même donnée en même temps (que ce soit à la même seconde, ou dans les même 10 minutes) peuvent "se gêner", par exemple s'ils veulent tous les deux acheter une quantité limitée d'une certaine marchandise.
- le rôle premier d'un SGBDR est d'assurer les contraintes d'intégrité, entre autre l'absence de doublons. C'est avant tout grâce aux clefs primaires et clefs uniques que les doublons seront évités. Sauf cas particulier et sur justification exclusiment, ce n'est en rien le travail de qui que ce soit d'autre que le SGBDR d'éviter les doublons.
- Si on a l'impression qu'une application ne répond pas dans des temps corrects, la première chose est de vérifier quel temps exact prennent tous les traitements, afin de se concentrer sur les plus longs.

# 4. Syntaxe de base du langage PHP

**Attention :** il n'est pas question de recopier ici le manuel de PHP qui est et restera la seule référence, ce chapitre est seulement un mémento. Il est disponible en de multiples formats sur http://www.php.net/
Cet aide-mémoire vous permettra de connaître rapidement les principales composantes avec des commentaires quant à leur utilisation. Voir aussi le chapitre 15 sur les fonctions les plus utiles de la plateforme PHP.

## *4.1 Les commentaires*

Tout bloc contenu entre /* et */ ne sera pas interprêté par PHP, sur une ou plusieurs lignes.
Tout ce qui suit // jusqu'au prochain retour à la ligne sera ignoré.
Il n'est pas possible d'imbriquer des commentaires.
Noter qu'il ne faut jamais utiliser // comme commentaire en langage C (à réserver au C++) donc selon les autres langages que vous utilisez, vous pourrez préferrer vous restreindre à /* */ pour éviter des mauvaises surprises.

Une bonne méthode de développement est de commencer à mettre tous les commentaires d'un traitement avant de remplir les blocs de code ainsi définis.

## *4.2 Les constantes*

Par définition une constante ne verra pas sa valeur changer au cours de l'exécution du programme. En revanche, elle reste aisément modifiable selon la configuration locale. Une constante est définie à l'aide de la fonction define(). Elles sont ensuite directement utilisées sans autre signe pour les référencer.
Exemple :
```
define('CONSTANTE_1','une jolie constante');
echo CONSTANTE_1;
```

Une restriction des constantes en PHP est qu'on ne peut pas définir un tableau en tant que constante.

## *4.3 Les variables*

Elles contiennent une valeur à un moment donné. Elles commencent nécessairement par le signe dollar $ et possèdent certaines restriction dans les caractères qui peuvent les composer. Limitez vous à AZaz_09 et vous n'aurez pas de surprises. Ne mixez pas les majuscules et les minuscules et vous en aurez encore moins. Le nom des variables est sensible à la casse (i.e. $TOTO $toto et $Toto sont trois variables qui n'ont rien à voir si ce n'est leur prononciation humaine).

## 4.4 Les variables et constantes prédéfinies

PHP met à disposition un certain nombre de variables et constantes dès le début de votre script, que ce soit TRUE et FALSE ou des tableaux complets comme $_REQUEST[ ] ou $HTTP_ENV[ ]. Nous y reviendrons en temps utile.

## 4.5 Variables variables

Il peut dans certains cas être pratique d'utiliser des variables génériques ($n1 $n2 $n3…. $nCOMPTEUR). PHP permet cette syntaxe en utilisant : ${"n".$compteur} ce qui permet de définir dynamiquement le nom de la variable.

## 4.6 Déclaration, typage

PHP est un langage non déclaratif faiblement typé.

- il est possible d'utiliser une nouvelle variable n'importe où dans un script sans la déclarer préalablement. Ce qui est un piège classique car le script suivant est syntaxiquement correct :
$maVariable=1;
if($mavariable==1) … // raté c'est pas v c'est V. Quand on vous disait de ne pas mixer maj et min…

- il est possible de savoir si le contenu d'une variable est numérique ou alphabétique en utilisant la fonction ad hoc. On peut vérifier si le contenu d'une variable est l'entier 1 ou la chaîne de caractère "1" constituée de l'entier 1. Mais rien n'empêche de stocker à la suite l'un ou l'autre dans la même variable, qui pourra à la ligne d'après recevoir un flottant…

```
$var=1;
if($var===1) // vrai
if($var==="1") // faux
if($var==1) // vrai
if($var=="1") // vrai
$var="abc";
$var=5.67;
```

NB : si les opérateurs triples permettant de tester le type d'une variable ne vous semblent pas clairs, ne vous en souciez pas et considérez que PHP est non typé. Sachez seulement qu'ils existent afin de ne pas tomber des nues si vous les rencontrez dans un script.

## 4.7 Les tableaux

Un tableau en PHP peut contenir n'importe quel types d'éléments, de même nature ou non. Ces éléments peuvent être adressés par un index numérique (notation indexée) ou par un nom (notation associative). Par exemple :

```
$tab[ ] =1;
$tab[ ] =2;
```

```
echo $tab[0]; // 1
echo $tab[1] ; // 2

$tab['z'] =1;
$tab['Z']="chaine";
echo $tab['z']; // 1
echo $tab['Z']; // chaine
```

En interne, PHP gère tous ses tableaux comme des hash tables.

## 4.8 Opérateurs

Tous les opérateurs habituels, mathématiques ou de la logique booléenne, sont existants. Noter l'utilisation de . comme concaténateur de chaîne (comme le + en java).

## 4.9 La syntaxe des chaines et des tableaux associatifs

Une chaine peut être définie entre simples ou doubles quotes. Entre simples quotes, elle ne sera pas interprêtée. Exemple :

```
$var=5;
echo '$var'; // $var
echo "$var"; // 5
```

De manière générale, la syntaxe $tableau[nom] est à bannir, vous devez utiliser $tableau['nom'] ou $tableau["$nom"]. Pour utiliser un tableau associatif dans une chaine de caractères entre doubles quotes, vous avez trois solutions :

- echo 'chaine constante'.$tableau['nom']."autre chaine et $var1 puis $var2";
- echo "chaine constante {$tableau['nom']} autre chaine et $var1 puis $var2";
- echo "chaine constante $tableau[nom]  autre chaine et $var1 puis $var2";

Le dernier cas (tableau associatif contenu dans une chaine entre doubles quotes) est le seul cas syntaxiquement autorisé de nom de la cléf du tableau non entre quotes.

## 4.10 Les variables de type "ressource"

Certaines fonctions, en particulier d'accès aux SGBDR, renvoient des buffers de résultats qui sont le "point d'entrée" des rangs en provenance de la base, et qui eux même passés en argument à une autre fonction, permettent de récupérer un rang précis. Ces buffers font partie des type "ressources" de PHP et seront traités dans le chapitre d'interaction avec les SGBDR. On trouve aussi dans les "ressources" tout ce qui plus généralement sous unix en C est un pointeur de fichier.

## 4.11 Structures de contrôle

Comme dans tous les autres langages :
if(condition) { } else { }

while(condition) { }
Cette construction est utilisée quand on veut exécuter un nombre indéterminé de traitements (tant que … je fais)

do { } while (condition). A noter que le bloc de code est exécuté d'abord et la condition testée ensuite. Ceci a pour effet que le bloc de code est toujours exécuté au moins une fois, ce qui peut ou non être le but recherché. L'emploi de cette instruction doit toujours être accompagné d'un commentaire sur la raison du choix et le traitement des erreurs vérifié.

for(condition d'initialisation ; condition d'arrêt ; instruction à exécuter à chaque boucle ) {}
Cette construction est utilisée quand on connait exactement le nombre d'instructions à exécuter.

continue;
Utilisée dans une boucle for/while/do while cette instruction a pour effet de sauter les instructions restantes et de passer au tour de boucle suivant. Elle doit toujours être accompagnée d'un commentaire en donnant la raison.

break;
Utilisée dans une boucle for/while/do while ou switch (cf ci-dessous) cette instruction a pour effet de sortir immédiatement de la boucle (instruction immédiatement après l'} fermante). Dans le cas du switch, son absence doit être commentée. Pour les autres types de boucles, exactement comme pour le continue, sa présence doit être justifiée en commentaire.

```
switch(variable)
{
case valeur1:
  // instructions
break;
case valeur2:
  // instructions
break;
default :
 // instructions
break;
}
```

L'une des structures les plus intéressantes permettant de faire du code sécurisé grâce à l'instruction default (qui récupère tous les cas non traités en amont). PHP étant un langage non typé, on voit trop souvent :

```
if($booleen) {traitement 1}
else { // traitement 2}
```

Or rien ne garantit que $booleen contiendra une valeur TRUE ou FALSE, surtout s'il provient du monde extérieur…
Par ailleurs, un avantage de PHP est qu'il peut utiliser tout type de valeurs dans les case, des chaînes de caractère par exemple. Ceci évite donc du code illisible du genre :

```
if($var=="chaine1")
{}
else if($var=="chaine2")
      {
       }
      else if(….)
```

le tout agrémenté de strcmp() par exemple en C.

La fonction exit() :
Interrompt le script et tous ses appelants. Aucune autre instruction ne sera exécutée. Admet une chaine de caractères en paramètre optionnel.

## *4.12 Les fonctions*

Une fonction permet d'écrire du code réutilisable (ce qui n'est en rien l'apanage de la programation orientée objet). Elle a de plus pour PHP l'avantage d'être imunisée à l'injection de variables dans toutes

les configurations (voir plus loin le chapitre sur la transmission des données). En PHP3, il fallait impérativement définir le code d'une fonction avant de l'utiliser (et de ce fait, gérer l'ordre des include/require exactement comme les #include "fic.h" en C). Depuis PHP4, l'ordre importe peu.

La syntaxe de déclaration d'une fonction la plus générique est :
```
function nomdelafonction($param1, $param2, $param3=1, $param4="chaine")
{
// instructions de la fonction disposant de $param1 $param2 $param3 et $param4
}
```
Dans l'exemple ci-dessus, $param1 et $param2 sont obligatoires lors de l'appel de la fonction. Si $param3 n'est pas précisé, $param3 et $param4 prennent dans le corps de la fonction leurs valeurs par défaut déclarées (ici 1 et "chaine"). Si $param3 est précisé mais pas $param4, seul $param4 prend sa valeur par défaut. Autrement dit, il y a 4 types d'appels possibles de la fonction ci-dessus :
```
nomdelafonction ($val1, $val2);
nomdelafonction ($vall, $val2, $val3);
nomdelafonction ($val1, $val2, $val3, $val4);
```

Il va de soi que l'appel nomdelafonction ($val1, $val2, $val4); aurait pour effet de mettre $val4 dans le troisième argument de la fonction et de laisser le quatrième prendre sa valeur par défaut, le nom des arguments étant totalement muet.

Renvoyer une valeur

Une fonction peut renvoyer une valeur unique grâce à l'instruction return. Il n'est pas nécessaire de le déclarer dans le prototype de la fonction. Une fonction peut, syntaxiquement, renvoyer une variable dans certains cas et pas dans d'autres, mais ceci est à éviter en termes de développement. En revanche, une fonction peut parfaitement renvoyer un entier dans un cas et un tableau dans un autre, à condition que ce comportement soit parfaitement documenté (car là encore il ne s'agit pas de syntaxe mais de facilité d'utilisation et de maintenance de cette fonction).

Portée des variables dans une fonction

Les variables contenues dans le corps d'une fonction sont locales à cette fonction.
Soit la fonction :
```
function mafonc1($toto)
{
$toto ++;
echo $toto; // NB : echo affiche le contenu de la variable. Si $toto vaut 1, echo $toto renverra 1
}
```
Appelons là dans le script suivant :
```
<?php
$toto=3;
mafonc1($toto):
echo $toto;
?>
```

C'est volontairement que le nom de la variable a été choisit identique pour rappeler qu'il n'a aucune importance.
La variable $toto du script principal prend pour valeur 3 à la première ligne du script. Cette valeur 3 est recopiée dans la variable $toto de la fonction juste avant la première instruction de la fonction. On incrémente alors $toto de la fonction de 1 puis on demande son affichage ce qui affiche donc 4. Après la dernière instruction de la fonction, la copie locale $toto cesse d'exister. Sa valeur 4 est définitivement perdue. De son côté, la variable $toto du script principal n'a pas été modifiée et vaut toujours 3 en sortie de la fonction. On affiche donc ensuite 3.

Soulignons donc à ce sujet de nouveau que les fonctions sont ainsi totalement immunisées à toute tentative d'injection de variables dans un script par un attaquant extérieur : seuls les arguments déclarés dans leur prototype sont disponibles.

Modification de la valeur d'une variable à l'intérieur d'une fonction

En particulier quand la fonction doit modifier plusieurs variables à la fois, il n'est pas toujours possible de se contenter de l'instruction return. Il est donc possible, sur déclaration de ce comportement dans la déclaration de la function, de modifier certaines variables dans le corps de la fonction. L'opérateur utilisé est l'esperluette & devant le nom des variables modifiables dans la fonction.

```
function mafonc2($varcopie, &$varmod1, &$varmod2)
{
$varcopie++; // local, ne sera jamais visible hors de cette fonction
$varmod1++; // affectera la variable hors de la fonction
$varmod2.="dans fonc ".$varmod1;
//Rappel :  l'opérateur . permet la concaténation de chaines comme + en java
}
```

Appel :

```
<?php
$v1=1;
$v2=10;
$v3="dehors";
mafonc2($v1, $v2, $v3);
echo $v1; // $v1 n'a jamais été modifiée, tout comme dans mafonc1(). Affiche 1
echo $v2; // $v2 a été incrémentée dans la fonction. Affiche 11
echo $v3; // $v3 a été modifié en lui ajoutant du texte. Affiche dehors dans fonc 11
?>
```

On remarque que le nom des variables dans le corps de la fonction et celui du script n'ont aucun lien entre elles.

Remarque : vous pourrez aussi trouver la syntaxe suivante :

```
function mafonc3(// arguments)
{
global $toto;
$toto++;
}
```

Script principal :

```
<?php
$toto=4;
mafonc3(// arguments autres):
echo $toto: // affiche 5
?>
```

Le mot clef global indique dans le corps de la fonction qu'il s'agit d'une variable déclarée hors du corps de la fonction. Le nom doit nécessairement être le même. Ce type de programmation est à bannir car impossible à maintenir à une tolérance près : les variables de configuration globales, généralement "rangées" dans un tableau de configuration comme $cfg[ ], situation acceptable car il ne peut être ignoré d'un développeur lisant le code que ce tableau ne devrait pas être modifié, et venant du fait qu'on aurait bien voulu déclarer $cfg[ ] en tant que constante mais PHP ne le permet pas.

## *4.13 Les objets*

Jusqu'en PHP 5, le mécanisme public/privé n'est pas implémenté en PHP. Toutes les propriétés et toutes les méthodes sont publiques. L'héritage est néanmoins fonctionnel depuis les premières version de PHP4 (présent dès PHP3 mais avec des bugs résiduels dans certains cas d'héritage).

La syntaxe de déclaration d'une classe en PHP 4 est :

```
class NOMDECLASSE extends NOMCLASSEMERE {
```

var propriete1;
var propriete2;
function methode1() {…}
…
}

En PHP 5, le mécanisme public/privé existe. On remplace dans la déclaration de la classe le mot clef var par le mot clef public, protected ou private. Les protected sont accessibles seulement aux classes dérivées. Idem pour les méthodes. Le constructeur est une méthode de nom __contructor(), le destructeur a pour nom __destructor()

La même déclaration en PHP5 sera donc :
public class NOMDECLASSE extends NOMCLASSEMERE {
public propriete1;
private propriete2;
public function methode1() {…}
…
}

On accède aux propriétés d'un objet par la classique syntaxe -> (this->propriété1=1; par exemple).

A noter que les objets ne sont pas plus persistents entre deux requêtes HTTP que les autres variables : il faut donc les sérialiser en fin de script, les retrouver et les désérialiser en début du suivant, avec gestion des erreurs …

L'héritage multiple est interdit. L'héritage simple est non seulement possible mais chaudement recommandé dans une vraie programmation orientée objet ! A partir de PHP5, la notion d'interface est disponible, semblable à celle de java.

L'utilisation d'objets peut permettre de présenter des "boîtes à outil noires", ce qui possède avantages et inconvénients : d'un côté l'usine à gaz des variables de statut interne est cachée et restreinte à la classe. D'un autre côté, le code PHP résultant est illisible pour quelqu'un qui ne connait pas le fonctionnement interne de la classe en question. S'il s'agit seulement d'une collection de méthodes en vrac, ce n'est pas de la programmation objet du tout. D'aucuns hésiteront aussi à appeler programmation objet un ensemble de classes interdépendentes mais sans un seul héritage…
Bref, il n'y a pas de solution universelle (sinon tout le monde l'utiliserait depuis longtemps), et empiler les couches n'est pas non plus un but en soi.

## *4.14 SSI - include - require*

Héritier du principe des Server Side Include, PHP peut utiliser des fichiers comme modules de code autonomes (exactement comme des fonctions, modulo la sécurité d'accès aux variables).
L'instruction include('fic.php'); fait rigoureusement **la même chose qu'un copié collé intégral** du fichier fic.php à l'endroit de l'instruction, ce qui a pour conséquence :
- toutes les variables définies au dessus de l'instruction sont définies dans le corps de fic.php au moment de son appel par include. Attention, si on appelle directement fic.php dans l'url elles ne seront pas nécessairement définies.
- toutes les variables modifiées ou ajoutées par fic.php sont modifiées / disponibles après l'appel à include() (hors codes de fonctions bien entendu)
- idem pour une définition de fonction

Le deux principales utilisations de include/require sont donc :

- définition de bibliothèques de fonctions (ou définition de classes). Les fichiers inclus, s'ils étaient appelés directement, ne font rien d'autre que de définir des constantes ou des fonctions mais ne "font" rien.

- traitement spécifique autonome, par exemple de génération d'un formulaire html dans le cas où il manque une données considérée obligatoire.

Il n'est en général pas une bonne idée de mixer ces deux types : préférez la séparation "faire"-"définir" (et on verra plus tard que "faire" possède lui même des sous-catégories de type logique - présentation).

Remarquons au passage que les instructions include/require peuvent créer une faille de sécurité si elles sont mal employées. A retenir : ne faites JAMAIS include($toto) ou require($toto) car si d'une manière ou d'une autre, le contenu de $toto venait à être surchargé par un attaquant, vous exécuteriez le code de l'attaquant sur votre machine.

Différence entre include et require :
En PHP3, require( ) se comportait exactement comme #include du préprocesseur C et faisait systématiquement un "copié-collé" du fichier requis avant l'analyse syntaxique par PHP du fichier. include( ) pouvait être utilisé avec des conditions, et on pouvait inclure tel ou tel fichier selon le contexte. On gagnait ainsi en vitesse d'exécution car PHP analysait seulement les fichiers qu'il traitait. On pouvait donc éviter de 'parser' certaines fonctions en les réservant aux cas où elles étaient nécessaires.

Depuis PHP 4 et le moteur Zend, la méthode de parsing des fichiers a changé et les gains en performance ne nécessitent plus cette gymnastique. Depuis PHP4, la seule différece entre include() et require() est le comportement dans le cas où le fichier à inclure est inaccessible (absent, mauvais droits d'accès etc…). Avec include(), on reçoit une erreur de type warning mais le script continue son exécution. Avec require(), une erreur fatale est générée et tout s'arrête (équivalent d'un appel à la fonction exit()).

include_once() et require_once().

Si vous définissez le même symbole deux fois, tout langage informatique se trouve dans un dilemne impossible à résoudre pour lui : quelle définition employer ? Ainsi, si votre fichier def.php contient la définition de la fonction mafonc() et que par erreur il est inclus deux fois dans le même script appelant, PHP indiquera une erreur. En PHP3, il n'y avait que deux moyens de se prémunir contre ces erreurs :
- faire très attention à ne jamais construire une hiérarchie qui amènerait à cette situation. Ceci faisait de toutes façons partie du processus permettant de séparer les fichiers que l'on allait appeler par require dans tous les cas et ceux par include sous conditions.
- utiliser l'équivalent syntaxique de PHP pour reprendre la très connue "ruse" du langage C avec les .h
En C :

```
#ifndef __NOMFICHIER
#define __NOMFICHIER
/* code du fichier header */
#endif
```

En PHP :

```
if(defined(__NOMFICHIER)==FALSE)
{
define("__NOMFICHIER",1);
// code php du fichier
}
```

En gros, on positionne un flag qui permet d'éviter que le parsing n'ait lieu plus d'une fois.

Depuis PHP4, on peut utiliser include_once() et require_once() qui font rigoureusement la même chose et évitent de gérer manuellement les doublons dans la table des symboles.

## 4.15 VERIFICATION DE LA COMPREHENSION

- Ecrivez un script qui initialise un tableau associant des noms à des prénoms. Parcourez le en affichant chaque prénom en face de chaque nom.

- Ecrivez une fonction ajouter_v1() qui admet entrée deux entiers et renvoie leur somme. Appelez là dans un script et vérifiez son bon fonctionnement.

- Ecrivez une fonction ajouter_v2() qui admet en entrée trois entiers. Le troisième entier devra contenir la somme des deux autres après appel. Vérifiez son fonctionnement dans le même script que ajouter_v1 () et remarquez bien la différence d'appel.

- Ecrivez une fonction afficher( ) qui admet en entrée une chaîne de caractères (message à afficher), un entier (la taille) et une chaîne de caractères (la couleur) dont la valeur par défaut est "green". Cette fonction affichera le message dans des balises FONT avec la taille et la couleur spécifiées. Appeler cette fonction avec les arguments : afficher("en vert", 2); puis avec afficher("en rouge", 3, "red");

# 5. Transmission des données

## 5.1- L'envoi par le client vers le serveur

L'instruction echo permet de renvoyer des données au navigateur, et en particulier du html. On a donc la moitié du flux. Après avoir déterminé ce qui doit transiter entre le client et le serveur, il s'agit donc maintenant de recevoir des paramètres envoyés par le client et de les traiter.

Le protocole HTTP définit les deux moyens de transmettre ces données : GET et POST. Il y a très peu de différence entre les deux, et en particulier, il n'y a pas plus de limitation de taille des données dans un cas que dans l'autre, seuls les navigateurs et les serveurs http imposent ces limitations. Nous regarderons donc le cas de la méthode GET.

Pour demander des données à un serveur http, une chaîne de caractères commençant par HTTP 1.1 GET est envoyée par le navigateur. Ceci peut parfaitement être reproduit manuellement en faisant un telnet sur le port 80 de n'importe quel serveur web
telnet www.saphirtech.com 80
et en tapant :
HTTP 1.1 GET index.html [entrée]
On obtiendra en retour le htlm de la page d'accueil index.html
C'est la même chose que de taper dans la barre de navigation http://www.saphirtech.com/index.html

Passons maintenant un argument directement dans la barre du navigateur. Si vous demandez :
http://www.saphirtech.com/cours.php?arg1=1&arg2=2&arg3=3
vous obtiendrez une page récapitulant tous les arguments reçus par le serveur avec leur nom et leur valeur associée . Il serait aussi possible de faire ceci en telnet sur le port 80.

**La première conclusion importante à retenir est que n'importe qui peut envoyer n'importe quoi à votre serveur web, ajouter, modifier, suprimer des arguments, demander 10000 fois la même chose en boucle** (1) **et rien ni personne ne pourra l'en empêcher.** "It's a wild wild world out there" comme qui dirait.

(1) par exemple :
#!/bin/ksh

while(true)
 wget http://targeturl/toto.php?arg1=1&arg2=2
demandera en boucle la même page avec les mêmes arguments jusqu'à ce que l'arrêt de l'instruction soit provoquée par CTRL-C ou kill -KILL du process local.

La syntaxe imposée par le protocole http pour envoyer des arguments est la suivante :
URL ? liste_d'arguments
où liste_d'arguments est définie comme une liste de taille indéfinie de couples nom=valeur séparés par le signe esperluette &

De la même manière que l'on a précédement tapé directement
http://www.saphirtech.com/cours.php?arg1=1&arg2=2&arg3=3 dans la barre de navigation du navigateur, on pourrait mettre ceci dans l'attribut HREF d'une balise lien html :
<a href="http://www.saphirtech.com/cours.php?arg1=1&arg2=2&arg3=3"> Cliquez ici </a>

Il va de soi que ce lien peut lui même avoir été généré dynamiquement par un script PHP, souvenons nous que pour le client qui reçoit une page html, il n'y a aucun moyen de savoir comment il a été généré.

Néanmoins ceci n'est toujours pas très interactif, on ne va pas demander à l'internaute de saisir tous les arguments sous la forme ? liste de couples séparés par & et avec le bon nom de l'argument SVP sinon ça marchera pas... !

On va donc utiliser les formulaires HTML pour faire saisir des données à l'internaute. Dans tous les cas, c'est toujours l'attribut NAME qui nous intéressera côté serveur. Par exemple, le formulaire ci-dessous enverra exactement les mêmes données que le lien html précédent :
```
<FORM ACTION="http://www.saphirtech.com/cours.php">
<INPUT TYPE="TEXT" NAME="arg1" VALUE="1">
<INPUT TYPE="TEXT" NAME="arg2" VALUE="2">
<INPUT TYPE="TEXT" NAME="arg3" VALUE="3">
<INPUT TYPE="SUBMIT">
</FORM>
```

Notons ici que l'attribut VALUE donne une valeur par défaut. Il n'est pas obligatoire. Bien entendu, l'internaute est libre de la remplacer par la valeur qu'il souhaite.

Cliquer sur le lien précédemment défini ou sur le bouton "submit" de ce formulaire enverra rigoureusement les mêmes données sous la même forme au serveur http. Il s'agit maintenant de récupérer ces données afin de pouvoir les exploiter en PHP.

## *5.2 - Traitement côté serveur*

Dans un premier temps, nous allons voir la version "expurgée", avant d'expliquer les différentes configurations possibles.

Dans les variables prédéfinies, il a été cité le tableau $_REQUEST. C'est lui qui va ici nous intéresser. Souvenez vous aussi qu'un tableau en PHP est une hash table et donc accessible de manière associative. De plus, quelle que soit la forme d'envoi des données, chaque donnée possède un nom, que ce soit dans le schéma liste de couples (nom=valeur) ou attribut NAME. Par conséquent, les données précédemment envoyées avec les noms arg1 arg2 et arg3 se retrouvent logiquement rangées dans les cases du même nom autrement dit le script ci-dessous :
```
<?php
echo $_REQUEST['arg1'];
echo $_REQUEST['arg2'];
echo $_REQUEST['arg3'];
?>
```

renverra au navigateur
123

Ceci est vrai dans toute les configurations de PHP, que les arguments soient reçus par la méthode GET ou POST.

Si pour une raison quelconque on souhaite se restreindre aux arguments reçus par méthode GET, on peut utiliser le tableau prédéfini $_GET à la place de $_REQUEST, ou le tableau $_POST pour se limiter aux arguments reçus par méthode POST. Même s'il n'est pas dit que ce type de programmation apporte vraiment quelque chose, sachez que c'est possible. Pour simplifier, le tableau $_REQUEST est l'association des tableaux $_GET et $_POST (quand l'un dest rempli, l'autre est vide) et de quelques autres données supplémentaires. Notez néanmoins que le contenu d'une variable est beaucoup plus dangereux que sa méthode de transmission...

Plus complexe est la gestion du paramètre de configuration de php register_globals, qui se définit dans le fichier de configuration php.ini et après la version "expurgée", voyons la version complète / complexe. La complexité se situe dans les failles de sécurité introduites / comblées selon la configuration et surtout le style de programmation.

Historiquement, PHP était livré en standard avec register_globals=On ce qui fait qu'il est dans ce cas inutile d'aller interroger $_REQUEST ou $_GET ou $_POST : dans ce cas, une variable PHP du même nom que celui qui est passé en paramètre est immédiatement prédéfinie, à la bonne valeur, dans le script.
Pour reprendre l'exemple de tout à l'heure, il suffit alors d'écrire :

```
<?php
echo "$arg1 $arg2 $arg2";
?>
```

pour obtenir : 1 2 3

Ceci est très pratique mais possède un effet pervers : n'oubliez pas que n'importe qui peut envoyer n'importe quoi à nos scripts PHP. Imaginez le code suivant :

```
<?php
// script avec faille, ne pas faire ceci
if($arg1=='login' && $arg2=='password') $authenticate="ON");
// …
if($authenticate=="ON") { // accès autorisé, faire des choses restreintes }
else include('generer_formulaire.php');
?>
```

Alors il suffit d'ajouter &authenticate=ON dans la liste des paramètres reçus pour court-circuiter le test sur le login et mot de passe (ou un champ HIDDEN dans des formulaires).

Il va de soit que la parade est vite trouvée : il suffit de coder proprement c'est à dire d'initialiser $authenticate = "OFF"; dans un else ou avant le test du login/passwd, de faire une fonction, bref, les solutions ne manquent pas.

Néanmoins, devant la pléthore de scripts présentant ce type de failles liées à une mauvaise connaissance du langage, la (fausse) rumeur montant que "PHP était un langage non sécurisé", le PHP Group a préféré livrer PHP avec par défaut register_globals à Off à partir de la version 4.1.x

Il est donc impossible, dans une configuration en register_gobals à Off, d'ajouter ou donner des valeurs par défaut à des variables dans un script, le développeur devant volontairement aller les chercher dans le tableau $_REQUEST qui sert de zone tampon (ou $_POST ou $_GET).

Le piège est que dans une configuration en register globals=On les tableaux $_REQUEST et autres sont toujours disponibles et le code fonctionne parfaitement. En revanche, on peut toujours modifier les variables comme on veut…

## 5.3 Règles de sécurité

Par conséquent, en tant que développeur, vous devez :

- utiliser $_REQUEST/GET/POST afin que vos applications puissent tourner sur une plateforme où register_globals est réglé à Off
- utiliser un seul point d'accès à ces tableaux (méthode ou fonction) afin de pouvoir valider les données reçues. Vous ne devez pas accéder directement aux données reçues sans les filtrer.
- initialiser correctement vos variables dans tous les cas : de toutes façons ça vous évitera bien des surprises
- forcer le comportement de vos applications à register_globals=Off par une directive de fichier .htaccess à la racine des scripts dans le cas où le serveur http est apache
- préciser dans la documentation utilisateur qu'il est fortement conseillé d'utiliser cette directive avec votre application

Remarque : les tableaux $_REQUEST, $_GET et $_POST sont apparus sous ce nom en PHP 4.0.6. Dans les versions précédentes (y compris php3 donc), il fallait utiliser $HTTP_ENV_GETVARS et $HTTP_ENV_POSTVARS, ce qui explique partiellement le peu d'engouement pour les développeurs à les utiliser vu la taille des noms.

## 5.4 Fonction de filtrage

Un point d'entrée unique aux données venant du monde extérieur est le seul garant d'un début de sécurité contre les injections et attaques de tous types. Il faut impérativement encapsuler l'accès aux données fournies par le monde extérieur.

Au strict minimum, cette fonction devra vérifier que les ' et " sont bien échappées et le faire si nécessaire. Elle devrait aussi aller jusqu'à faire du filtrage de types de données : si vous vous attendez à recevoir un entier, il n'y a pas de raisons que la variable reçue contienne autre chose que des chiffres, éventuellement le signe - en premier caractère. Ce cours ne peut pas détailler toutes les attaques possibles (injections sql sur caractères ou entiers, include/require dynamiques, XSS...)

Un document traitant de la sécurité des applications web est disponible sur www.saphirtech.com et donne un canevas d'une fonction de filtrage qui est reproduit ici sans les explications nécessaires à sa bonne compréhension.

ATTENTION : ceci est un exemple simplifié de fonction de filtrage. Il s'agit là d'un canevas pour donner les principales vérifications, nullement d'une librairie finalisée.

```
// filtrage des données, à utiliser systématiquement pour toute variable externe
function fx_filter($name, $type='STRING', $def='')
{
// si la variable n'a pas ete recue, gerer proprement l'erreur
// on se fichez de savoir comment a ete transmise la donnee.
if(!isset($_REQUEST[$name])) return $def;

$unsafe=trim($_REQUEST[$name]);
// selon le type de variable attendue, traiter.
switch($type)
 {
// on gere ici les entier et les flottants de la meme maniere, ceci est un exemple simplifie
// ceci protege des injections sql sur les entiers et evite toute incohérence.
 case  'INT':
 case  'FLOAT':
```

```
   if(!is_numeric($unsafe)) return $def;
   return $unsafe;
   break; //inutile, pour respecter la syntaxe habituelle
 default :
  // on se protege des injections SQL sur des strings
  // remplacer ces tests pour Sybase, qui echappe les ' par une autre ' et on un \
   if(get_magic_quotes_gpc()==0)
    {
   $unsafe=addslashes($unsafe);
    }
   break;
 }
// il reste les XSS.
// on pourrait aussi utiliser html_entities() à la place de htmlspecial chars
// mais attention aux tailles des champs sgbd a augmenter
// on pourrait aussi utiliser strip_tags()
 $safe=htmlspecialchars($unsafe);
 return trim($safe);
}
```

Appel :

```
$nom_famille=fx_filter ('nom_famille');
$age=fx_filter('age','INT');
$qte=fx_filter('qty','INT',1);
```

### *5.5 VERIFICATION DE LA COMPREHENSION*

- Une vérification de validité en javascript est-elle suffisante pour considérer que la donnée a été vérifiée ?
-Ecrivez un script reception.php qui admettra en entrée trois variables : auteur, titre, texte. Sur réception, il doit afficher ces trois variables avec retour à la ligne.
-Appelez directement ce script par http en passant les valeurs que vous souhaitez.
-Ecrivez un autre fichier envoyer.html qui contiendra un lien html de type A HREF et transmettra ces trois variables.
- modifiez envoyer.html pour ajouter un formulaire à trois champs de type INPUT="TEXT" qui transmettra ces trois même variables à reception.php et testez le résultat.

Réponses :
Non, non et non, aucune vérification quelle qu'elle soit exécutée ailleurs que sur le serveur ne permet de valider une donnée. N'importe qui peut toujours envoyer n'importe quoi à vos scripts, vous ne pourrez rien faire pour l'en empêcher.

## 6. L'importance du flux des données

Aucune méthode classique de modélisation ou de développement ne s'attarde suffisement sur une partie cruciale pour les performances et la sécurité du site : le transit des données entre le client et le serveur. Il est important, pour tout écran de l'interface HTML généré de savoir en permanence ce qui a permis de le générer, ce qu'il transmet et pourquoi.

## 6.1 - quelle est la requête qui l'a engendré, est-elle valide ?

Le cas le plus simple est la page d'accueil. On y arrive quand on demande le site directement, éventuellement quand une erreur 404 (page not found) est produite, ou quand on clique sur un lien interne permettant d'y revenir.
Mettons nous de nouveau dans le cas d'un site marchand avec un panier virtuel. Pour arriver à la page valider_commande.php qui va passer la commande de 'en cours' (l'internaute est en train de remplir son panier) à un statut 'attente paiement' (l'internaute a passé commande et on attend confirmation du paiement) dans la base, et présenter au client internaute le récapitulatif de sa commande, les remerciements ad hoc, le lien pour imprimer un bon de commande à renvoyer avec le chèque et le lien pour paiement par CB, il faut nécessairement que le client ait cliqué sur "finaliser ma commande". Un appel direct du script par l'URL échouera probablement, mais dès lors qu'il ne corrompt pas le SGBD (et en particulier, qu'il ne passe pas deux fois la commande !!!), ce qu'il affichera au client est peu important car il n'a pas à faire d'appel direct, c'est ou un bookmark mal placé, ou un refresh, ou une tentative d'intrusion/corruption de la base. Afficher un message d'erreur et un lien vers la page d'accueil sera donc amplement suffisant si on ne reçoit pas sur cette page valider_commande.php les arguments attendus.

## 6.2 - quels sont les arguments qui ont été reçus de la part du client ?

Sont-ils bien tous utiles ou sommes nous en train de trimballer des tonnes de variables pour le plaisir d'engorger le réseau ? De la même manière qu'une fonction possède un prototype avec un nombre d'arguments bien précis, un script attend certains arguments de manière obligatoire, d'autres sont facultatifs, et les derniers… inutiles donc on ne les transmet pas.

## 6.3 -quelles sont les données renvoyées par le script au client ?

Elles doivent être classées dans les deux catégories suivantes : les données qui seront utiles à la suite de la navigation (par exemple : l'identifiant de session) et celles qui sont uniquement à but d'affichage.

Par exemple, une erreur de conception majeure d'un site marchand serait de faire un SELECT en SGBD à l'écran numéro 1 pour avoir le prix d'un article et de le faire transiter ensuite en HIDDEN pour l'utiliser plus tard pour le calcul de la facture sous prétexte que ça gagne une requête SQL. N'importe qui pourra modifier le prix de l'article ! Ne rigolez pas à la lecture, ce type d'erreur majeure dans l'analyse du flux des données a été constaté réellement. Si on doit en effet afficher le prix de l'article au client, on ne doit pas lui donner la possibilité de le choisir pour autant. Le prix utilisé pour le calcul doit être conservé côté serveur exclusivement.

Cet exemple illustre un principe plus général concernant les données reçues de la part du client (navigateur) par les scripts. Souvenez vous : n'importe qui peut envoyer n'importe quoi. Par conséquent, on ne fait jamais confiance aux données qui arrivent car elles peuvent être erronées de bonne foi (erreur de saisie, par exemple saisir 1 0 [un-espace-zéro] au lieu de 10 [un-zéro], ceci fera probablement une erreur de syntaxe SQL si l'espace n'est pas supprimé) ou par trafficage volontaire (attaque). Notons à ce sujet qu'il y a presque autant d'attaques internes à l'entreprise qu'externes, être sur un intranet n'est en rien une garantie de sécurité. Un commercial va baisser les frais de port d'un de ses clients pour l'aider à passer commande et toucher sa prime, un conseiller boursier va baisser les frais de transaction d'un client à qui il a donné un mauvais tuyau ou à un de ses amis, etc… Et ensuite on va passer des heures à cherche un bug inexistant dans le code après des échanges de mails salés avec le département de comptabilité alors qu'il s'agit d'une erreur de conception.

Notons aussi que l'ajout de contrôles sur le navigateur en JS est totalement inutile en termes de sécurité et ne sert que dans deux cas : d'une part une pré-vérification de la cohérence des données pour le confort de l'internaute qui n'aura pas besoin de lire un message d'erreur perdu au milieu du formulaire et resoumettre (intérêt faible) et d'autre part la vérification de certains champs dont la non validité entraîne une erreur fatale lors du traitement serveur. Pour reprendre l'exemple de la saisie erronée 1 0 on pourra accepter qu'une personne ayant désactivé JS reçoive une "erreur fatale" à la page suivante, à la condition bien entendu que cette erreur ne laisse pas le traitement à moitié fait sur le seveur : si vous avez deux requêtes à exécuter, que la première l'est correctement mais pas la seconde, vous aurez une base incohérente (sauf en cas de transaction bien entendu).

De manière générale, l'auteur déconseille toute validation de formulaires en JS, ça ne sert à rien à part à grossir de 20% la taille du fichier html envoyé, elles doivent impérativement être faites côté serveur de toutes façons.

## *6.4 -Le sempiternel problème des formulaires*

En tant qu'internaute, qu'attendez vous d'un formulaire ? En posant différement la question, quel est le formulaire idéal ? (mis à part celui qui se remplit tout seul sans appuyer sur les touches, ça n'existe pas encore).
A la première lecture : indiquer clairement tous les champs obligatoires et les cohérences attendues.
En reprise sur erreur parce qu'une incohérence / champ invalide a été détecté(e) :
- que tous les champs correctement remplis le soient toujours afin de ne pas avoir à les saisir de nouveau
- éventuellement qu'ils ne soient plus modifiables (discutable comme intérêt, en relisant sa saisie on peut trouver une faute de frappe).
- que toutes les erreurs détectées soient clairement indiquées avec un message compréhensible au bon endroit.

Le cahier des charges est clair et simple, il faut maintenant trouver une méthode générale qui permet de s'en sortir à tous les coups. Un algorithme général simple est le suivant :
- récupérer les variables venant du monde extérieur (on vient de le voir au chapitre précédent)
- faire toutes les vérifications de présence des champs obligatoires, de cohérence entre les champs inhérentes à ce formulaire et cette application.
Si une erreur au moins (ou plus) est détectée : générer le formulaire avec ses champs pré-remplis par ce qui a été reçu et mettre fin au script.
Sinon, lancer le traitement "normal" des données du formulaire qui ont ainsi été validées puis afficher le résultat du traitement.

Outre le fait qu'il est totalement générique, on remarquera deux points importants dans cet algorithme :

Premièrement, le cas où l'internaute arrive sur le formulaire pour la première fois n'est que le cas particulier où tous les champs obligatoires sont absents et toutes les incohérences possibles présentes. Les données pré-remplies seront vides car rien n'a été envoyé. Il est donc strictement inutile de faire un cas particulier pour la première arrivée sur le formulaire si ce n'est éventuellement dans le texte du message affiché en face de chaque champ obligatoire non renseigné. Ceci se gère très facilement en ajoutant une variable disant "déjà vu" quand on re-génère le forumulaire suite à erreur.

Deuxièmement, c'est bien le même script qui gère tout, il n'y a aucun allez-retour inutile entre le client et le serveur sous prétexte qu'un champ est manquant. Pour ceux qui seraient tentés de faire n'importe quoi, on va bien utiliser une commande de type include/require suivie d'un exit() et non pas une bidouille avec la fonction header(). C'est un premier exemple de la séparation de la logique applicative et de la présentation : le script PHP qui va générer le formulaire en pré-remplissant les champs présents ne fait que ça, et est appelé si nécessaire par le script de traitement.

Combinons maintenant les deux chapitres précédents et gérons un formulaire simple qui demande deux informations obligatoires, une information facultative, et les affiche quand tout va bien.

Ci dessous, le code du fichier qui va générer le HTML, generer_form.php
Attention : ce fichier ne fait donc QUE de la génération de html. Ce n'est pas lui qu'on va appeler maintenant dans l'URL, mais le fichier de traitement.

---- début du fichier ----

```
<HTML><HEAD>
<!-- generer_form.php -->
</HEAD><BODY>
Formulaire de saisie<BR><BR>
<FORM ACTION="traiter_form.php">
<TABLE>
<TR>
<TD>
<INPUT TYPE="TEXT" NAME="nom" VALUE="<?php echo $l_nom; ?>">
</TD>
<TD>
<?php if($l_nom=='') echo '<FONT COLOR="red">(Donnée obligatoire)</FONT>'; ?>
</TD>
</TR>
<TR>
<TD>
<INPUT TYPE="TEXT" NAME="prenom" VALUE="<?php echo $l_prenom; ?>">
</TD>
<TD>
<?php if($l_prenom=='') echo '<FONT COLOR="red">(Donnée obligatoire)</FONT>'; ?>
</TD>
</TR>
<TR>
<TD COLSPAN="2">
<INPUT TYPE="TEXT" NAME="deuxieme_prenom"
VALUE="<?php echo $l_deuxieme_prenom; ?>">
</TD>
</TR>
<TR>
<TD COLSPAN="2" ALIGN="RIGHT">
<INPUT TYPE="SUBMIT">
</TD>
</TABLE>
</FORM>
</BODY></HTML>
```

---- fin du fichier ----

On a vu apparaître "magiquement" trois variables $l_nom, $l_prenom et $l_deuxieme_prenom et on note que les données seront envoyées au script traiter_form.php que nous allons donc regarder immédiatement.

---- début du fichier ----

```
<?php
/* ------------
traiter_form.php
Point d'entree unique du traitement du formulaire.
Verifier que le nom et premier prenom ont bien ete remplis.
Si oui, faire le traitement des donnees recues.
 ---------------- */
require('lib.php'); // definition de la fonction fx_filter
$l_nom=fx_filter('nom');  // $_REQUEST[' nom '];
$l_prenom= fx_filter('prenom'); // $_REQUEST[' prenom '];
$l_deuxieme_prenom=fx_filter('deuxieme_prenom'); // $_REQUEST[' deuxieme_prenom '];

if($l_nom=='' || $l_prenom=='') // au moins une donnee obligatoire absente
{
```

```
require(' generer_form.php');
exit();
}
// Toutes les donnees sont la, lancement du traitement.
// ici, le traitement est simpliste et consiste simplement a les afficher
// On pourrait bien entendu les stocker en base de donnees par exemple.
echo '<HTML><HEAD></HEAD><BODY>';
echo "Nom : $l_nom <BR> Pr&eacute;nom : $l_prenom <BR>";
echo "Deuxi&egrave;me pr&eacute; nom : $l_deuxieme_prenom<BR>";
echo '</BODY></HTML>';
?>
---- fin du fichier ----
```

### *6.5 VERIFICATION DE LA COMPREHENSION*

Modifiez reception.php et envoyer.html pour vérifier que les variables auteur et texte sont bien présentes (variables obligatoires).

# 7. Bilan intermédiaire

On sait maintenant recevoir des arguments de la part d'un client distant (navigateur) en se souvenant de l'importance de ce flux sur les performances et la sécurité de l'application, écrire des fonctions pour généraliser les traitements, et renvoyer des données. La boucle est complète en ce qui concerne des données volatiles. Il s'agit donc maintenant de pouvoir stocker des données rémanentes côté serveur et c'est là que les SGBDR entrent en jeu.
Il est bien entendu possible de stocker des données dans des fichiers plats sur le serveur. Sauf dans ces cas extrêmes, cette solution est trop contraignante pour être viable et le besoin d'un SGBD se fait rapidement sentir.

# 8. Rappels sur les SGBDR

Ces notions devraient être acquises, au moins dans les grandes lignes. Néanmoins, ayant régulièrement constaté un manque de mise en pratique des connaissances, l'auteur préfère ajouter ces rappels, qui ne sauraient se substituer à un cours complet et structuré.

### *8.1 Utilité d'un SGBDR*

Un Système de Gestion de Base de Données Relationnelle est utilisé pour stocker des données de manière organisée afin de pouvoir par la suite les retrouver et les modifier. Il gère nativement la notion de contraintes d'intégrité et d'accès concurentiel.

Exemples : RDB, Oracle, Sybase, DB2 (ibm), ms-sqlserver, informix, ingres, postgresql. Le statut de SGBD Relationnel pour Mysql dépend de sa version et de ses options de compilation (prendre la version dite "InnoDb" pour du vrai relationnel). MS-access est plus une application bureautique mono-utilisateur qu'un sgbd.

## 8.2 Choisir le type de la donnée

Il y a en gros quatre types de données à stocker dans une base. Nous allons voir ici les critères qui induisent leur choix mais aussi leur portabilité. Il n'est pas rare que la machine de production soit sous unix mais celle de développement sous windows parce que le PC du coin de la salle (ou celui du développeur) coûte moins cher qu'un serveur Sun ou HP-UX. De plus, la migration des données d'un SGBDR vers un autre (rachat de société, uniformisation entre services etc…) est monnaie courante dans le monde de l'entreprise. La portabilité d'une table est donc non négligeable.

## 8.3 Les quantités

Il n'y a pas beaucoup de pièges fonctionnels sur ce type de données. La seule question à se poser est de savoir si la quantitée concernée est entière ou à virgule.

En revanche, il y a des pièges liés à certains SGBDR. Il est vivement conseillé d'utiliser des NUMERIC à la place des FLOAT sous Sybase, ces derniers étant passablement buggés. Sous Sybase toujours, l'équivalent des autoincrement, les SYB_IDENTITY, peuvent "exploser" en cas d'arrêt trop violent du serveur Sybase (et passer de quelques milliers à 10 puissance 12 ce qui fait "pêter" tous les type de variables C qui y accèdent. N'oublions pas que PHP est écrit… en C).

A noter que certains SGBD (comme mysql) acceptent de "tailler" le nombre maximal de stockage des entiers. Par exemple on peut déclarer un INT(2) ce qui veut dire que l'on peut stocker de -99 à +99. Faire attention à ne pas être trop "avare" de taille de colonnes de ce type, une interruption de service d'un site pendant plusieurs heures à cause d'un compteur mal dimensionné peut coûter beaucoup plus cher à l'entreprise que 10 ans du stockage sur disque des quelques mégaoctets ainsi "économisés".

La migration des données de type quantité d'un SGBDR à un autre pose rarement des problèmes.

## 8.4 - les dates

Il y a deux types de dates natifs : le type humainement compréhensible et le type TIMESTAMP à la mode unix.
Le type humainement compréhensible se découpe parfois en trois sous-types :
DATE au format YYYYMMDD
TIME au format HH:MM:SS
DATETIME qui combine les deux précédents : YYYYMMDD HH:MM:SS
Notons que dans certains cas, il peut être intéressant d'avoir deux colonnes séparées DATE et TIME au lieu d'une seule DATETIME, par exemple si l'on veut qu'une action ne puisse avoir lieu qu'une seule fois par jour il suffit de mettre la colonne de type DATE dans une clef unique ou primaire.
Ces types offrent l'avantage d'être portables inter SGBDR et inter OS, même si leur nom peut varier (par exemple sous postgresql, le datetime s'appelle.... timestamp !)

Le type TIMESTAMP à la mode unix, quand il existe stocke le nombre de secondes écoulées depuis une date arbitraire. Il présente deux gênes possibles : si vous arrivez sans calculatrice à savoir si la date 12938754123 se trouve dans le passé ou le futur, moi pas. De plus, rien ne garantit que la migration d'un SGBDR vers un autre, ou même simplement du même SGBDR mais d'un OS à un autre (windows vers unix, ou l'inverse) fonctionnera sans problèmes. Il présente l'avantage d'être d'une représentation indépendante du "locale" (options de "régionalisation" au format français, US, GB etc…) et d'être automatiquement remplit à l'heure système courante du SGBDR si non précisé dans la requête (est-ce vraiment un avantage…?)

Enfin, on peut aussi choisir de gérer les dates en tant qu'entiers sous la forme 20033112134559 par exemple (13h 45min 59 sec le 31 décembre 2003). Attention, il faudra bien entendu écrire les fonctions utilitaires "qui vont bien" pour gérer ces dates.

A noter que sous PostGresql, le TIMESTAMP est un DATETIME à la ms de précision.

## 8.5 - le texte

Il y a trois types de variables de type texte : taille fixe et variable de faible contenance, forte contenance.

- les CHAR sont de taille fixe. Si vous stockez 'a' dans un CHAR(80) la taille occupée est toujours de 80 caractères (sauf en Sybase <11.5.1 qui considérait la taille réelle… La migration depuis une version antérieure mettait un souk assez intéressant dans toutes les applications avec des procédures stockées en transac-sql).

- les VARCHAR. Ils sont de taille variable. Stocker 'abc' dans un VARCHAR(80) ne prend que 3 caractères. A noter que sous Oracle, ce type, qui existe toujours pour des raisons obscures depuis des années, est à bannir car totalement buggé de l'aveu même d'Oracle. Utiliser à la place du VARCHAR2 i.e. VARCHAR2(80) à la place de VARCHAR(80).

Le gain de performances d'utilisation du CHAR par rapport au VARCHAR ou VARCHAR2 relève de l'économie de bouts de chandelles. Quand vous en serez à optimiser l'application sur ce genre de critères et non dans votre code applicatif, on en reparlera !!
Généralement, les CHAR et VARCHAR sont limités à 255 caractères. Les VARCHAR2 d'Oracle depuis plusieurs années et les VARCHAR les versions postérieures à 12.x de Sybase acceptent des tailles supérieures, de l'ordre de 1000 caractères.

Au delà de ces tailles, on peut soit couper les données en plusieurs colonnes, soit utiliser le type TEXT. Une recherche de clause WHERE sur un type TEXT ne doit pas se faire sans une consultation précise du manuel du SGBDR concerné afin de vérifier le comportement des performances associées à ce type de requêtes.

## 8.6 - les données binaires

Elles sont stockées dans des types de donnée BLOB (Big Large Object). On doit vraiment se poser la question de la pertinence du stockage de données binaires dans un SGBDR. Il est très peu probable de sélectionner les données selon une clause WHERE appliquée à ce type de colonnes. Ce sont des données "mortes" qui sont seulement stockées là. Analysons donc un peu les caractéristiques de ce type de stockage :

- toutes les données sont au même endroit. Ceci évite de faire deux sauvegardes différentes mais peut aussi augmenter la taille des tablespaces à sauvegarder d'un bloc de manière qui devient finalement moins gérable.

- l'applicatif (PHP) doit tout faire passer dans la couche de communication avec SQL. Ceci ne fait qu'ajouter de l'overhead (temps d'accès inutiles) dans le cas où il est sur la même machine que le SGBDR. En revanche, dans le cas où il serait nécessaire de recevoir des fichiers depuis la machine du client, et comme il n'est pas rare que le SGBDR se trouve derrière un firewall, s'il y a plusieurs machines frontales, ou si elles sont en DMZ, il ne sera pas possible de stocker les fichiers ailleurs qu'avec le SGBDR. Ouvrir des ports FTP pour un upload ne sera pas vu d'un bon oeil. On pourra donc utiliser le SGBDR comme serveur de fichiers transparent.

Dans le cas d'Oracle, s'il s'agit de stocker des fichiers complets, le type BFILE propose un bon compromis : le fichier est stocké "dans" le SGBDR mais en tant que fichier autonome et non pas dans le tablespace d'Oracle, et l'overhead est quasi nul localement, seul reste celui de la communication SQL et là les paramètres comme la taille de prefetch peuvent résoudre beaucoup de problèmes de performances.

Notons que le stockage de données binaires pose aussi presque systématiquement des problèmes en cas de migration d'un SGBDR à un autre.

De manière générale, il est aussi simple et plus évolutif de simplement stocker le chemin du fichier, sur le file system et/ou sous forme d'url. Notons d'ailleurs que dans ce cas l'url peut référencer un fichier distant.

## 8.7 MCD/MPD

Le Modèle Conceptuel de Données (MCD) sert à modéliser les données. Il est souvent éclaté en une multitude d'entités et de relations plus ou moins obscures, liées ou non aux contraintes applicatives. Il est rarement utilisable directement par le développeur.
Le Modèle Physique de Données (MPD) se traduit directement en instructions SQL de type CREATE TABLE. Il doit avoir été si nécessaire dénormalisé en passant par une étape intermédiaire (MLD, Modèle Logique de Données) afin que le codage soit rendu possible et avec des performances acceptables. S'il faut faire 4 ou 5 jointures avant d'accéder à la moindre donnée réelle et que ces données ne peuvent pas être mises en cache mémoire, il faudra faire quelque chose ou détourner l'application en simulateur de course de rameurs…

## 8.8 DDL/DML

DDL : Data Definition Language. Tout ce qui commence par CREATE ou DROP et qui agit sur des contenant (table) ou des comportements (index, trigger, procédure…). Une traduction "AllGood Compliant" (1) souvent proposée dans la littérature SQL est "LDD" pour "Langage de Définitions de Données".

DML : Data Manipulation Language. SELECT, INSERT/UPDATE/DELETE. Tout ce qui agit sur du contenu, sur des données. Traduit par "LMD" pour "Langage de Manipulation de Données".

(1) : loi Toubon sur l'utilisation exclusive de la langue française

## 8.9 Opérations ensemblistes

Il est complètement faux de penser qu'une opération SQL impacte nécessairement un rang et un seul : toute requête SQL de DML impacte de zéro à tous les rangs de la ou des tables en jeu (jointures).

Exemples :
SELECT * FROM table1 WHERE 1=2 ne renverra jamais un seul rang. Cet exemple est un peu extrême : on peut aussi considérer un SELECT sur une table vide ou tout simplement qu'aucun des rangs présent dans la table ne correspond à la clause WHERE.
INSERT INTO table2(col1, col2) SELECT col1, col2 FROM table1 insèrera exactement tous les rangs présents dans table2 au moment du SELECT (modulo la compatibilité des colonnes et les clefs uniques bien entendu).

## 8.10 Les index

Un index est une indirection, exactement comme les index des livres. Donc toute opération de type INSERT et DELETE, et potentiellement les opérations d'UPDATE prennent plus de temps à s'exécuter car il faut maintenir l'index. Les améliorations sur des opérations de type SELECT dépendent directement du nombre de rangs renvoyés par la requête. Imaginez que vous deviez accéder

exclusivement à un livre en lisant son index. Si vous avez l'intention de le lire en entier, c'est idiot, autant le lire linéairement, chaque recherche dans l'index vous fait perdre du temps.

Le positionnement d'un index doit donc être justifié par le type d'opérations exécutées sur cette table, en particulier les jointures. Les a priori sont vos ennemis.

Les statistiques du SGBD doivent régulièrement être mises à jour afin que les index soient utiles.

Pour d'obscures raisons d'implémentation, les clef étrangères sous Mysql-InnoDb nécessitent la création manuelle d'index sur la table fille.

## *8.11 Les clefs*

Définition d'une clef primaire selon la théorie des SGBDR : la clef primaire d'une table est le plus petit ensemble de colonnes de cette table permettant d'identifier de manière unique un de ses rangs.
A noter dans cette définition : une clef primaire peut parfaitement être composée de plusieurs colonnes.

Clef muette
(trop) souvent on trouve l'utilisation de clef de type autoincrement. Celles-ci ne sont en rien une solution universelle ! Elle est dite muette car elle n'a aucun lien avec les données stockées dans la table.

Clef pseudo-muette
La clef pseudo-muette ressemble dans sa lecture à une clef muette : elle n'est pas humainement facile à relier aux données de la table. Mais elle garde ce lien. Exemple typique : la signature mathématique md5 de deux champs accolés et passés en majuscules. Attention dans son utilisation : elle ne dispense pas toujours de positionner correctement des clefs uniques exactement comme pour une clef muette afin de prévenir les erreurs en cas de modification manuelle ou plus généralement de modification extérieure aux scripts PHP. Elle peut poser des problèmes de compatibilité en cas de modifications faite par plusieurs programmes différents, surtout s'ils sont écrits dans des langages différents ou tournent sur des OS différents. Elle non plus n'est en rien une solution universelle.

Comment déterminer la clef primaire d'une table ?
Une table correspond à un besoin applicatif précis : on stocke des données selon ce dont on va en faire, pas pour le plaisir d'occuper du disque.
Commencez par regrouper vos données en entités/tables. Vérifiez si une clef primaire, selon la définition ci-dessus, existe déjà ou non. De toutes façons, il vous faudra déterminer les clefs uniques afin d'éviter les doublons. S'il en existe une et que :
- elle fait moins de 3 ou 4 colonnes
- le nombre de caractères ajouté de la taille maximale des colonnes fait moins de 80-100 caractères
alors vous avez la clef primaire de la table.

Sinon, il vous faudra :
- déterminer la ou les clefs uniques de la table.
- vérifier si vous avez besoin d'une clef primaire muette ou pseudo-muette que vous ajouterez donc pour les besoins de l'applicatif (et non pour les besoins du MCD).

Exemples :
Table de publication de reporting d'opérations boursières (ordres d'achat-vente). Rien n'empêche un client d'acheter deux fois 10 alcatel au même prix dans la même journée. Par conséquent, on aura bien la même opération (achat) sur la même valeur, au même prix, pour la même journée comptable sur le même compte titre. La table ne possède pas de clef primaire. En a-t-elle besoin ? A priori non : pour éditer les borderaux à publier, la requête de sélection sera : SELECT col1, col2…,coln FROM table_oper WHERE numero_compte='123456' AND mois='octobre' AND annee =2004 ORDER BY jour

Le remplissage de la table sera fait automatiquement par une procédure d'extraction ou une alimentation de fichiers backoffice. Bref, à part à perdre de l'espace disque, ajouter une clef primaire sur cette table ne servirait à rien.

Table de gestion de contacts commerciaux
Il faut au moins : nom, prénom, entreprise, numéro de téléphone, adresse email (et d'autres choses). Avez vous des candidats à la clef primaire ? Oui, au moins deux : le téléphone et l'adresse email. Il faut alors poser la question aux utilisateurs (ou au cahier des charges) pour savoir si celles-ci sont ou non recevables.
- peut-on avoir plusieurs contacts au sein d'une même entreprise ? Si non, fin de la discussion, on a la clef (email ou téléphone, l'un ou l'autre, comme vous voudrez)
Si oui :
- Indiquer le téléphone en UNIQUE interdit les numéros de standard ou force à inclure l'extension (le poste) soit dans la même colonne soit dans une colonne supplémentaire
- Indiquer l'adresse email comme UNIQUE part du principe que tous les employés ont une adresse nominative et interdit les adresses génériques du genre info@ ou contact@

Bref, c'est une question applicative qui va fortement impacter les possibilités du programme, non une question rhétorique.
Notez qu'ajouter une clef muette (autoincrement) en plus ne répond pas aux questions posées …

Cas le plus complexe : si l'on veut pouvoir gérer plusieurs contacts au sein d'une même entreprise en laissant la possibilité qu'ils aient un même numéro de téléphone ET une même adresse email, il va falloir considérer comme n-uplet unique l'ensemble des colonnes (nom, prénom, téléphone) ou éventuellement (nom, prénom, téléphone, service) où le service sera la Comptabilité, les Ventes, le Support etc… Si une entreprise possède deux employés qui ont le même nom et même prénom dans le même service, elle n'arrivera pas elle même à les différencier, donc on ne sera plus à ça près. Une clef unique à 4 colonnes est parfaitement acceptable aux termes de la théorie des SGBDR. Ensuite, tout dépend de l'application. Dans le cas présent, si l'on veut pouvoir mettre à jour un contact commercial, dans tous les cas, il faudra faire transiter toutes les données car tous les champs seront a priori modifiables : on a rien à gagner en ajouter un identifiant plus court. En revanche, dans les écrans qui feront une association entre le contact commercial et le reste de l'application, il faudra faire transiter 4 données systématiquement. Il n'est donc pas une hérésie d'ajouter une clef muette qui facilitera l'identification d'un contact dans les autres tables et le reste de l'application. On arrivera donc à la structure suivante :

```
CREATE TABLE contact (
id_contact INT AUTOINCREMENT NOT NULL,
nom VARCHAR(50) NOT NULL,
prenom VARCHAR(50) NOT NULL,
service VARCHAR(30) NOT NULL,
telephone VARCHAR(20) NOT NULL,
email VARCHAR (50) NOT NULL
[autres données mortes]
);
ALTER TABLE contact ADD CONSTRAINT PRIMARY KEY(id_contact);
ALTER TABLE contact ADD CONSTRAINT UNIQUE KEY(nom, prenom, service, telephone);
```

Remarquons que l'emploi de la clef autoincrement a été justifié et utilisé en dernier recours, pas en première solution automatique. Et que sans la contrainte UNIQUE, sur le fond, tout est faux.

## *8.12 Pourquoi utiliser des clefs primaires et uniques ?*

Généralement, tout le monde pense à la première utilité : on a très souvent (pas toujours) besoin d'un identifiant unique pour chaque rang. Souvenons nous maintenant d'une autre contrainte vue dans le premier chapitre : n'importe qui peut appeler nos scripts le nombre de fois qu'il voudra avec les mêmes arguments. Rien ni personne ne peut l'en empêcher. Par conséquent, si l'internaute clique sur "rafraîchir/recharger la page" ou sur "retour arrière" et de nouveau sur le bouton SUBMIT d'un

formulaire, le même code PHP sera exécuté une deuxième, troisième, n-ième fois. L'effet pervers de la clef muette auto-increment est que sans la contrainte UNIQUE qui définit la vraie clef primaire fonctionnelle de la table, on aura des doublons, triplets, … n-uplets rigoureusement identiques en données réelles mais différents grâce à l'autoincrement. Quant à purger les tables ainsi corrompues, c'est une opération manuelle pénible et dangereuse. Bref, l'utilité principale d'une clef primaire et des clefs uniques n'est pas seulement d'identifier des rangs de manière unique, mais au même titre d'empêcher les doublons.

A noter donc, une situation gênante qui n'a aucune solution satisfaisante connue de l'auteur à l'heure actuelle : les gestions de votes anonymes. L'utilisation d'adresses IP sources est fausse et trop contraignante. Il n'existe pas de clef primaire acceptable permettant d'interdire à la même personne de voter deux fois deux jours de suite même sans changer d'ordinateur. Un cookie ne sert bien entendu à rien. On peut empêcher le vote automatique par script en affectant un jeton à chaque demande lors de la génération du formulaire et en demandant à l'internaute de saisir la valeur de ce jeton dans une zone de saisie du formulaire. Il va de soit que le rendu visuel de ce jeton ne doit pas être analysable automatiquement par un script, et sera donc probablement une image générée dynamiquement. Comme il s'agira probablement de chiffres et de lettres, il ne serait d'ailleurs pas si difficile que ça d'écrire un petit module de reconnaissance de formes qui analyserait l'image…

Bref, comme toujours pour la sécurité informatique, tout est une question de compromis entre le temps que l'on veut que le "cracker" passe à casser la protection par rapport à l'intérêt de ce qu'il obtient. De manière générale, il faut toujours se prémunir contre les erreurs de bonne foi des internautes, car celles-ci auront lieu nécessairement. Les données centrales de l'application doivent être protégées par le schéma de la base de données. Pour le reste, "faire au mieux".

## *8.13 Clef étrangère*

Définition : une clef étrangère est clef primaire dans une autre table. Elle peut ou non faire partie de la clef primaire de la table courante. Si elle est également clef primaire de cette table, il faudra se poser des questions sur l'utilité et la raison réelle d'avoir deux tables séparées avec la même clef primaire. C'est tout à fait possible mais doit néanmoins être justifié par la situation courante.
Le but d'une clef étrangère est de forcer la cohérence entre des tables. Si on a une table "référentiel client" où la clef primaire est le numéro de client, et une table "commandes", il est logique qu'on ne puisse passer une commande que si on a un numéro de client. Donc une colonne numero de client dans la table commande sera clef étrangère référençant la table "référentiel client".
Un vrai SGBD Relationnel dans cette configuration refusera catégoriquement d'insérer une ligne dans la table "commandes" s'il ne trouve pas le rang correspondant dans le référentiel client. Il refusera aussi de supprimer la ligne dans la table "référentiel client" tant qu'il y aura une ligne restante dans les commandes (car elle deviendrait sinon orpheline).
MySQL accepte syntaxiquement la notion de clef étrangère, mais il l'ignore s'il n'est pas compilé en InnoDB avec index sur la colonne etc… Bref, il faut encore pour le moment ajouter moult (1) restrictions pour parler de SGBD relationnel pour MySQL, mais la situation évolue dans le bon sens.

(1)rappel : moult est un adverbe, donc invariable

## *8.14 Transactions*

MySQL non compilé en InnoDB ne gère pas les transactions. Ceci donnera lieu a un chapitre complet plus loin (voir "accès concurrentiels instantanés").

## *8.15 Le fameux NULL*

Sans rentrer dans le débat "NULL est-il une valeur ou un marqueur" qui relève, pour le développeur moyen, des hautes sphères autorisées à penser, retenez une bonne fois pour toutes ces deux principes :

- NULL n'est PAS ZERO.
- NULL veut dire "la valeur de cet attribut est inconnue" ou éventuellement "n'a pas de sens étant donnée l'étape du cycle de vie à laquelle en est cette instance".

Exemple : on gère un catalogue de pièces d'antiquaire. Chaque pièce est unique et se voit attribuer les notions de "date d'entré au catalogue" et de "date de vente". La date de vente doit être NULLABLE, car tant que l'article n'a pas été vendu, sa valeur est inconnue.

## *8.16 Conclusion : que retenir ?*

1) le schéma du SGBDR est le centre névralgique de l'application. Une application qui fonctionne correctement avec des données erronées renverra toujours n'importe quoi. Ce principe est rappelé par l'adage bien connu "Garbage in, garbage out".

2) ne pas mettre de clef muettes de type autoincrement à tour de bras. Ce n'est pas, contrairement à ce qu'on peut voir trop souvent, la super solution géniale qui résoud tout. C'est parfois la seule solution, mais en rien une panacée universelle.

3) ne jamais oublier de déterminer les clefs uniques nécessaires sur chaque table, en particulier quand elles ont un autoincrement en clef primaire.

# 9. Le dialogue client-serveur entre PHP et le SGBDR

### ATTENTION : dans le chapitre qui va suivre, c'est PHP qui est le client ! Le serveur, c'est le SGBDR.

PHP peut se connecter nativement, c'est à dire en utilisant les librairies spécifiques à ces SGBD qui sont donc optimisées et disposant de plus de fonctionnalités, à la quasi totalité des SGBD existants. Oracle, Sybase, PostGreSQL, MySQL, MS-SQLServer, Ingres, Informix … Au pire, on peut utiliser ODBC (MS-Access). Sauf si vous travaillez sur des gros systèmes assez anciens comme RDB (et encore, on peut la faire se présenter comme un Oracle 7) ou des SBGD "expérimentaux" comme des SGBDOO, il est presque sûr que vous pourrez faire dialoguer votre base de données avec PHP.

Le dialogue entre un client et son SGBD peut en gros se résumer à six phases principales.

## *9.1 Etablissement de la liaison physique.*

Le SGBD est à l'écoute sur une IP et un port bien déterminés (et dépendants de sa configuration interne déterminée par son administrateur). Le dialogue a lieu entièrement par sockets. Il faut donc d'abord ouvrir une socket vers le serveur. Il est donc clair qu'il faudra indiquer à PHP ces informations.

Remarque : dans le cas où le SGBD et PHP sont sur la même machine, l'adrese IP conventionnelle 'localhost' ou '127.0.0.1', si le système d'exploitation le permet, peuvent utiliser des couches de communication plus performantes que les sockets (comme les named pipes/tubes nommés) et ce de manière complètement transparente.

## *9.2 authentification dans le SGBD.*

Tous les SGBD gèrent de manière autonome l'accès à leurs ressources. Il faut disposer d'un login et d'un mot de passe déclarés dans le SGBD pour y accéder.
Ce login et ce mot de passe sont ceux qui vous sont par exemple demandés dans phpmyadmin ou stockés dans le fichier $HOME/.my.cnf de votre compte unix.

Pour MySQL , ces deux premières phases du dialogue sont assurées par une seule fonction PHP : mysql_connect().
Cette fonction accepte trois arguments : l'ip / l'alias du serveur, le login, et le mot de passe. Deux remarques donc :
- sauf localhost ou une entrée définie dans le fichier /etc/hosts de la machine où est PHP, n'utilisez jamais le nom dns de la machine où se trouve le serveur SGBD, toutes vos connections risqueront de perdre du temps à résoudre ce nom en adresse IP auprès du serveur DNS le plus proche.
- où est donc passé le port d'écoute du serveur MySQL ? Celui-ci étant toujours le même par convention, il a été stocké dans le fichier de configuration globale de PHP appelé originalement php.ini dans la section [mysql]. (souvenez-vous que nous avons déjà rencontré php.ini en ce qui concerne le paramètre register_globals)

Erreurs possibles :
1) impossible d'ouvrir la socket. Ceci peut avoir de multiples raisons. Le serveur SGBD n'est pas à l'écoute. Il sature en nombre de connexions. La machine locale ne peut pas voir le serveur SGBD sur le réseau temporairement (routeur HS / saturé etc…). Elle sature en nombre de descripteurs de fichiers pouvant être ouverts.
2) la liaison réseau est bien établie mais il y a incohérence entre le login/pass envoyés et la configuration interne de MySQL. Par exemple, cet utilisateur est déclaré comme pouvant se connecter depuis 'localhost' mais vous avez utilisé une adresse IP réelle qui est interprêtée comme une connexion distante. Le mot de passe est erroné. Etc…

Dans tous les cas, si une erreur, quelle qu'elle soit, est rencontrée, la fonction mysql_connect() renverra FALSE et positionnera les informations nécessaires accessibles par les fonctions mysql_error() et mysql_errno() qui renvoient respectivement un texte humainement compréhensible (pour les initiés en tous cas) et le numéro interne Mysql de l'erreur qui a eut lieu.

Si tout ce passe bien, mysql_connect() renvoie un identifiant de connexion. C'est l'un des types "ressources" cités au début du cours dans les type de variables.

## *9.3 Sélection de la base à utiliser*

Certains SGBD, comme Postgresql, Sybase ou Mysql, regroupent leurs tables dans des 'bases'. Un utilisateur peut avoir plusieurs bases. Une base ne peut contenir qu'une seule table d'un nom donné, mais une table de même nom peut être présente dans plusieurs bases. Lors de la connexion en ligne de commande à ce type de SGBD, il faut utiliser une commande ressemblant souvent à "USE DATABASE nomdelabase;".
Le pendant de cette commande en fonction PHP pour MySQL est la fonction mysql_select_db().
Elle admet deux paramètres en entrée : le nom de la base à utiliser, et l'identifiant de connexion renvoyé par la fonction mysql_connect().
Elle renvoie TRUE en cas de succès et FALSE en cas d'échec. Les raisons d'une erreur sont multiples une fois de plus : la socket vient d'être fermée par l'un ou l'autre des systèmes d'un côté ou de l'autre, par un équipement réseau mal configuré (temps de sessions socket inadaptés) ou en erreur, la base de données demandée n'existe pas ou cet utilisateur n'a pas le droit d'y accéder.

Notons que d'autres SGBD comme Oracle n'ont pas la notion de bases. Chaque table appartient à un login. Chaque login possède donc un ensemble de tables, souvent appelé "schéma". Il ne peut pas avoir plusieurs tables du même nom, mais un autre login peut avoir des tables du même nom.

## 9.4 l'exécution de requêtes SQL

Après ces phases d'établissement du dialogue, on va pouvoir exécuter des requêtes et traiter le résultat en retour. Une requête est envoyée à la base par la fonction mysl_query(). Elle admet en paramètres la requête à exécuter et l'identifiant de connexion renvoyé par mysql_connect().

A la réception de cette requête, le SGBD commence par en vérifier la syntaxe (il commence par vérifier qu'il comprend ce qui lui est demandé). Ensuite, il vérifie si le login qui utilise cette connexion a effectivement le droit de faire les actions qu'il demande. Bien entendu, ces deux vérifications peuvent générer des erreurs. Si tout va bien, la requête est exécutée. Le client SGBD (ici, PHP) reçoit alors une structure que l'on peut ainsi modéliser :
- un champ OK/numéro d'erreur
- un champ "nombre de rangs impactés"
- un buffer contenant les rangs en question (si requête de type SELECT).
Cette représentation est celle d'une structure interne en C et n'est pas visible directement sous PHP, mais correspond aux différentes fonctions qui vont être citées ci-dessous.

Ce retour est présenté en PHP sous la forme d'une resource, parfois appelée $res ou $resbuff pour "result buffer". Il vaut FALSE en cas de problème détecté.

Une instruction de type DDL ne remplira jamais le nombre de rangs impactés ou le buffer de résultats qui n'ont pas de sens. (rappel : DDL = Data Definition Langage = DROP/CREATE). Une instruction de type DDL ne peut qu'être ou ne pas être exécutée. Bien que syntaxiquement correcte et ayant le droit d'être exécutée, une instruction CREATE TABLE peut échouer (table de ce nom déjà existante, espace disque plein par exemple).

Les instructions de type DML se séparent en deux catégories : d'une part le SELECT, d'autre part les INSERT/DELETE/UPDATE.

Toutes ces instructions peuvent échouer dans les vérifications préliminaires (syntaxe, droits d'accès). Mais une instruction exécutée correctement ne va pas nécessairement impacter de rangs. Le champ "nombre de rangs impactés" sera donc systématiquement renseigné en cas de requête exécutée. La fonction mysql_affected_rows() permet de savoir combien de rangs ont été impactés. Ces instructions ne renvoient aucun rangs, donc le buffer reste vide dans tous les cas.

Enfin, le SELECT. Lui aussi peut renvoyer de zéro à tous les rangs de la table (disons de 0 à N rangs où N est le minimum des deux tables en cas de jointure pour généraliser). La fonction mysql_num_rows() permet de connaître le nombre de rangs renvoyés et donc directement la taille du buffer qui cette fois, est rempli (ou reste vide s'il y a zéro rangs).

## 9.5 Traitement du retour

Outre le test de mysql_num_rows() si on fait un SELECT cette fois, il va bien falloir accéder aux données renvoyées.
On peut donc ensuite lire le buffer de résultat rang par rang avec les fonctions mysql_fetch_*(). Sachez aussi pour les rares cas où vous en aurez besoin que la fonction mysql_data_seek() permet de modifier l'ordre de lecture du buffer de résultats.

Deux fonctions principales à connaître dans les mysql_fetch_*( ) : mysql_fetch_row() et mysql_fetch_array(). Citons aussi mysql_fetch_object() pour le principe.

Dans tous les cas, ces fonctions admettent en paramètre le buffer de résultats décrit ci-dessus et renvoyé par la fonction mysql_query(). Fetch_row() renvoie un tableau indexé, fetch_array() un tableau associatif, fetch_object() un objet. Sachant que cet objet n'a aucune méthodes et n'hérite d'aucune autre classe, il n'a d'objet que le nom sauf si on le caste en une classe définie. Ceci n'est possible que si l'objet

a exactement en propriétés les mêmes noms que les colonnes sélectionnées de la table. Bref, ce n'est pas parce qu'on utilise fetch_object() qu'on fait de la POO.

On peut réutiliser une même connexion (identifiant renvoyé par mysql_connect()) autant de fois que l'on veut au sein d'un même programme, et faire autant de séquences mysql_query()/mysql_fetch_* que l'on souhaite.

## *9.6 La rupture de la liaison*

La liaison par socket ne va pas rester ouverte indéfiniment, d'une manière ou d'une autre, elle finira bien par être fermée, que ce soit :
- par appel explicite dans le programme PHP à la fonction mysql_close()
- par appel implicite automatique à cette fonction à la fin du programme dans le cas où on n'utilise pas les connexions persistantes (mysql_pconnect() au lieu de mysql_connect())
- par fin du process httpd courant
- par fermeture de la part du SGBDR qui considère que cette session a été trop longue
- par fermeture de la part d'un équipement réseau pour la même raison.

Même si des erreurs sont possibles lors d'un appel explicité de mysql_close (exemple : connexion déjà fermée, impossible de la fermer une seconde fois) elles sont à ignorer dans le programme courant : que pourrait-on bien y faire ? En revanche, elles doivent être tracées et l'origine trouvée car sont à 50% des cas une erreur de programmation et 50% des cas une mauvaise configuration réseau.

## *9.7 Les fonctions annexes*

mysql_data_seek()
Dans de rares cas, il est utile de gérer le buffer de résultats en le parcourant plusieurs fois ou d'une certaine manière bien spécifique.

mysql_free_result().
Devrait être appelée sur tout buffer de résultats après son traitement, surtout s'il est volumineux. Est implicitement appelée par le moteur Zend en fin de script depuis la 4.0.6.

mysql_pconnect()
Quand cette fonction est appelée, il est inutile d'appeler mysql_close explicitement. C'est le moteur Zend de PHP qui gère les ressources qui gèrera cette connexion. Cette connexion est conservée un certain temps entre les requêtes. Si un autre script (ou le même) accède au même SGBDR, au lieu d'ouvrir une nouvelle socket, celle qui a été conservée sera réutilisée (si elle est disponible).
Il est faux de penser que l'utilisation de mysql_pconnect améliore systématiquement les performances, en particulier à cause des ressources qui doivent être conservée sur le client (httpd+php) et le serveur (mysqld) alors qu'elles sont inutilisées.

Bref, une fois de plus ne pas avoir d'a priori. Un test de charge représentatif sera probablement nécessaire pour choisir l'utilisation de connexions persistentes ou non.

# 10. Les sessions applicatives

Souvenons nous que deux requêtes http successives sont totalement indépendantes pour le serveur http et donc pour PHP. Le protocole http ne donne pas de moyen natif de gérer la continuité de la

navigation. Notons qu'il n'est pas nécessaire de mettre en place ce "pistage" systématiquement, mais le nombre de cas où il est presque incontournable est suffisament important pour devoir le traiter complètement. Un exemple très classique est la restriction d'accès à certaines zones d'un site soumise à vérification de login et mot de passe : on fait transiter une seule fois le login/pass en chiffré (SSL-https), et un jeton est alloué pour un certain temps appelé validité de la session.

Il va donc falloir, d'une manière ou d'une autre, attribuer un identifiant unique à chaque nouvel internaute arrivant sur le site (sous conditions de vérification d'authentification ou non), et être capable de vérifier à chaque nouvelle demande arrivant si ce jeton est référencé côté serveur ou non. Nécessairement donc, il faudra garder trace de l'ensemble des jetons valides côté serveur, et de le faire transiter systématiquement.

La réalisation technique de ces besoins peut ensuite être faite de deux manières principales : "manuellement" ou en utilisant les sessions natives de PHP (apparues à la version 4, inexistantes en PHP3).

## *10.1 Gestion complète ou "manuelle"*

La génération de l'identifiant unique (le jeton), son stockage côté serveur, et son transit entre toutes les pages sera entièrement à la charge du développeur.

Structures de données côté serveur :
Il faut au moins stocker le jeton et son heure d'expiration. Dans le cas où ce jeton est associé à une authentification (login/pass) il sera souvent intéressant de stocker le login associé. Ceci permettrait également aisément de limiter le nombre de sessions actives à une à la fois en cas de besoin (il suffit de mettre une clef unique sur la colonne login). Il va de soi que le jeton est la clef primaire de cette table, il serait sinon impossible de différencier deux internautes qui auraient le même jeton, aussi faible que soit cette probabilité.

Il est également évident que ce jeton ne doit pas être déterministe dans sa génération : si un pirate est capable de deviner le jeton attribué à une personne identifiée par son login/pass, il pourra lui "voler" sa session dès lors que l'utilisateur légitime est loggué. Le plus simple est donc de générer un jeton de taille fixe (30 à 50 caractères selon le temps de session et la fréquentation du site), en piochant aléatoirement dans une liste de caractères autorisés. D'autres méthodes sont bien entendu possibles, l'essentiel étant qu'il ne soit pas prévisible à l'avance par un attaquant extérieur.

La structure de la table sera donc la suivante :
```
CREATE TABLE session (
id_session CHAR(40) NOT NULL,
login VARCHAR(20) NOT NULL,
expiration DATETIME NOT NULL
);
ALTER TABLE session ADD CONSTRAINT PRIMARY KEY(id_session);
```

La colonne login peut-elle être en contrainte UNIQUE ? Tout dépend de l'utilisation des sessions par rapport à la navigation sur le site. Imaginons un site marchand qui permet de consulter un catalogue complet de marchandises, et de remplir un panier virtuel. Pour d'évidentes raisons de marketing, on ne va pas forcer l'internaute à s'enregistrer et obtenir un login/password avant même de savoir si les produits présentés l'intéressent. On peut donc imaginer d'utiliser une table session_anomyme qui possèderait la même structure mais sans la colonne login. On peut aussi tout simpement utiliser un mot clef (par exemple "UNKNOWN" ou "ANONYME") ou la valeur NULL (déclarer login nullable dans ce cas) pour tous les internautes actuellement en train de naviguer de manière anonyme. Ceci évitera d'aller à la pêche dans deux tables différentes pour toutes les actions communes ne nécessitant pas d'être loggué.

Voyons maintenant l'algorithme d'une fonction permettant l'enregistrement d'une nouvelle session.
```
algo generer_session(string login)
```

{
purger toutes les sessions expirées
générer un jeton aléatoire de 40 caractères
insérer le nouveau jeton avec la bonne date d'expiration, gérer les erreurs éventuelles
}
On voit donc se dégager deux constantes et deux requêtes SQL : la taille du jeton en nombre de caractères et le temps de validité de la session (mettons ici 45 minutes). Ces deux constantes seront définies en PHP par la fonction define() déjà citée. Pour les requêtes sur la table session, il y a un DELETE (purge) et un INSERT.

La gestion des dates entre PHP et le SGBDR est toujours un dilemne et, de manière générale, relève des habitudes du développeur et de l'arsenal des fonctions mises à disposition par le SGBD plutôt que d'un raisonnement logique réel. Dans tous les cas on peut avoir des problèmes de portabilité. L'auteur conseille plutôt d'utiliser les fonctions de gestion des dates du SGBDR quand celles-ci permettent de faire ce qui est nécessaire pour une raison d'évolutivité de répartition de charge. Le SGBDR reste central mais rien n'empêche de mettre plusieurs machines frontales avec PHP. Si l'on utilise la fonction du SGBDR pour trouver l'heure courante, quelle que soit la machine source, l'heure sera toujours la même. Si chaque script détermine l'heure système courante, on peut avoir des décalages. Mais comme indiqué au début de ce paragraphe, chaque développeur possède ses habitudes à ce sujet.

Requête de purge
En français : "supprimer toutes les sessions dont l'heure d'expiration est passée // est dans le passé.
En SQL : DELETE FROM session WHERE expiration < NOW()

NOW() est la fonction Mysql qui renvoie l'heure courante, elle est équivalente à SYSDATE de SYBASE etc…

On a donc purgé toutes les sessions anciennes. Notons que ce type de purge permet de garder un système qui s'auto-nettoie. Il est certain que la table des sessions ne sera jamais engorgée par des sessions expirées depuis des lustres, ce qui n'est pas le cas d'une purge lancée par un programme indépendant appelé en crontab ou autre.

Génération de l'identifiant aléatoire :

```
define("TAILLE_ID", 40);
define("PIOCHE", "abcdefghijklmnopqrstuvwxyz0123456789");
define("TAILLE_PIOCHE", 36);

function generer_id()
{
// si on ajoute les majuscules cela fera de 36 à 62
mt_srand();
for($i=0; $i++; $i<TAILLE_ID)
{
$index=mt_rand(0, TAILLE_PIOCHE-1);
$id.=substr(PIOCHE, $index, 1);
}
return $id;
}
```

NB : ce code est modifiable à deux niveaux :
- ne pas recalculer systématiquement la taille du tableau par rapport à l'index et définir TAILLE_PIOCHE comme INDEX_MAX_PIOCHE à 35.
- ne pas utiliser la fonction substr mais aller directement dans une chaîne de caractères considérée comme un tableau. Ceci est possible en PHP mais indiqué dans le manuel comme étant une méthode dépréciée.
On pourrait aussi passer la taille de l'identifiant en paramètre de la fonction, etc…

Il reste alors à insérer la ligne dans la table session par un INSERT.
En français : autoriser un nouvel identifiant pendant les 45 minutes à venir.

En SQL : INSERT INTO session VALUES ('$id','$login',DATE_ADD(NOW(), INTERVAL 45 MINUTE);
La valeur 45 est ici restée "en dur" et non en constante PHP afin de simplifier l'écriture de cet exemple.

La nouvelle session a été enregistrée. L'internaute va revenir. Il faudra alors vérifier qu'il dispose d'une session, qu'elle est valide, et dans beaucoup de cas, lui "en redonner pour 45 minutes" c'est à dire gérer une fenêtre glissante de 45 minutes au fur et à mesure de son activité.

Voyons un algorithme possible.

NB : on oublie pour cet exemple la gestion de l'accès restreint par login dans une zone de navigation pour la clarté.

- SELECT * FROM session WHERE id_session='$id_reçu'
- Vérifier si sbgd[expiration] < heure système courante
Si oui : UPDATE session SETexpiration = heure système+45 minutes.
Si non : traitement d'erreur

Là, on a tout juste sur ce qui doit être fait. On a tout faux sur la réalisation technique. Critique :
1) on ne fait jamais un SELECT * à part en ligne de commande en débug vite fait-mal fait. Si vous demandez le rang (clef primaire !) où l'identifiant de session vaut "ABCD" à votre avis, est-il utile de sélectionner la colonne id_session ? Oui exclusivement si vous allez passer le tableau en retour à une fonction et que vous devriez de toutes façons ajouter l'information à la main.
2) pourquoi faire du tri ou du filtrage dans le code PHP ? Un SGBD est fait pour utiliser des clauses WHERE, utilisons les !

Une deuxième version pourrait donc être :
SELECT login FROM session
WHERE id_session='$id_reçu'
AND expiration >NOW()
Si un rang est reçu, on pourra faire un UPDATE. Sinon, gérer une erreur.

Mais alors bonsangmaisc'estbiensûr ! A quoi sert-il de sélectionner les données pour les mettre à jour ?

L'algorithme et la fonctions sont alors simplistes :

UPDATE session SET expiration=DATEADD(NOW(), INTERVAL 45 MINUTE)
WHERE id_session='$id_reçu'
AND expiration >NOW()

Il suffit de demander combien de rangs ont été mis à jours grâce à la fonction mysql_affected_rows(). Si un rang a été mis à jour, cette session était valide et le sera pour 45 minutes de plus. Sinon, gérer une erreur.

On pourrait bien entendu ajouter à cet requête UPDATE la clause AND login IS NOT NULL ou la clause AND login !='UNKNOWN' selon la convention choisie pour les logins anonymes si la page accédée est dans une zone restreinte.

Avec deux fonctions et trois requêtes SQL on a monté un système de sessions utilisateur utilisable sur tous les SGBDR dans son principe de fonctionnement (modulo l'appel des fonctions donc) et indépendant de la version de PHP (3 ou 4). Ce système utilise un SGBDR qui est presque toujours centralisé et unique même dans le cas où il y a de la répartition de charge sur plusieurs machines frontales. Il s'auto-entretient en fonction de l'activité des internautes (purge).

Un autre avantage est que la table session est parfaitement visible par toutes les requêtes SQL, par exemple d'un outil d'administration.

A partir de cette table de sessions, on pourra selon les besoins applicatifs utiliser une table qui contiendra des données "temporaires". Par exemple, un panier virtuel de commandes en ligne aura pour structure :
- l'identifiant de session
- la clef primaire du catalogue des produits (un code barre par exemple)
- la quantité demandée
- éventuellement un précalcul des prix HT,TTC et TVA ainsi engendrés

Un outil d'administration pourra alors avoir en une seule requête tous les connectés anonymes ou non, avec leur panier en cours. Un refresh toutes les 30 secondes par une balise META dans le header de la page, et on a une console d'activité quasi temps réelle du site. Facile le PHP non ?

On pourrait aussi raffiner la fonction de purge afin qu'elle déclenche une action avant le DELETE pour chaque session expirée, ceci peut parfois être une obligation imposée pour raisons de sécurité ou légales de contestations futures de la part des internautes.

En revanche, ce système de gestion  nécessite systématiquement une requête SGBD à chaque accès de page, ce qui permet par ailleurs d'initialiser dans tous les cas la connexion à la base de données et de l'avoir disponible pour la suite des scripts.

Le plus "pénible" de cette méthode est qu'elle nécessite de faire transiter partout (dans les liens http de type a href ou en champs HIDDEN le fameux identifiant de session attribué plus haut.

## *10.2 Les sessions natives*

Voyons maintenant le fonctionnement des sessions natives PHP4, leurs avantages/inconvénients et comment éventuellement combiner les deux systèmes de sessions.

Pour le développeur, le système de sessions natives permet de stocker côté serveur des variables de tous types simples (entiers, strings, objets, tableaux…) et de les "retrouver" de manière transparente aux requêtes suivantes du même utilisateur dans le tableau originalement appelé $_SESSION. C'est donc une très grande souplesse de développement, avec comme souvent la tentation du "fourre-tout". Pas besoin de requêtes vers un SGBD, on utilise un tableau d'un nom prédéfini, trois fonctions (session_start(),  sesion_register(), session_destroy()) sans plus de complications apparentes. Il est des cas où les sessions natives PHP4 peuvent être très pratiques, en particulier pour éviter de faire transiter dans l'URL des choix globaux de l'utilisateur qu'il détermine au début de sa navigation et qui ne seront pas stockés de manière rémanente sur le serveur. Elles peuvent être couplées à des cookies.

En pratique, les sessions sont stockées sur le disque du serveur web dans un répertoire spécifique qui devra donc être protégé en lecture depuis le monde extérieur (c'est à vous de le vérifier en installant la machine de production). Si plusieurs machines frontales se répartissent la charge, il faudra avoir recours ou à un système de load balancing qui redirigera toujours le même client sur la même machine frontale (aisé si par exemple le load balancing se fait par IP source déclarée de provenance) ou à un montage de disque partagé type NFS, ce qui peut devenir une contrainte de performances et de sécurité.

Les sessions ne se voient pas entre elles et personne ne peut les lire efficacement (elles ne peuvent donc pas par exemple être utilisées par un outil d'administration externe simplement). Le développeur est tenté d'y stocker n'importe quoi : en particulier, des pointeurs de fichiers / sockets ou pire, des buffers de résultats revenant du SGBDR (heureusement, ça ne marche pas). Il est impossible de déclencher une action sur une session expirée.

En revanche, on peut coupler les deux systèmes en gardant comme identifiant de session native PHP4 et de session "manuelle" la même valeur, et en ayant des temps de session sensiblement égaux.

# 11. L'accès concurrentiel instantané

## *11.1 Les transactions*

**Ce chapitre pourrait se résumer à trois mots si tous les SGBD supportaient les transactions : "utilisez les transactions" !!**

Mais certains des SGBD ou assimilés auxquels se connecte PHP ne les connaissent pas. En particulier, MySQL ne supporte la notion de transaction qu'à partir d'une version relativement récente et uniquement compilé en version InnoDB.

Accès concurrentiel : deux scripts différents, ou le même script, lancés par deux utilisateurs différents, accèdent à la même donnée en modification conditionnelle. Exemple classique : sélectionner le maximum d'une colonne, l'augmenter de 1, puis l'utiliser dans une autre table. Si l'on a pas de transactions et que l'on utilise l'algorithme ci-dessous, la catastrophe n'est qu'une question de temps.
SELECT MAX(colonne) FROM table AS max
max=max+1;
UPDATE table SET colonne=max;
INSERT INTO autre_table VALUES (max…)

Le temps passé entre le SELECT et l'UPDATE est faible, mais il n'est pas nul et il n'y a aucun moyen de savoir combien de requêtes seront traitées par le SGBD entre ces deux là. Ce n'est qu'une question de temps avant que la situation suivante n'arrive : l'internaute A et B demandent la page contenant le script faisant ces deux requêtes à peu près en même temps, et l'ordre d'exécution est le suivant.
- A fait le select. Il reçoit max=25.
- B fait aussi le select. Il reçoit donc lui aussi max=25.
- A fait max=max+1 ce qui donne 26
- B fait max=max+1 ce qui lui donne aussi 26.
- B fait son update en premier (pourquoi lui ? Pourquoi pas. L'ordre de passage des requêtes est totalement incontrôlable). Il stocke max=26. Ceci est pour l'instant cohérent.
- A fait son update. Il stocke lui aussi max=26 et là c'est faux.

C'est justement pour éviter ce problème que les clefs de type autoincrement ont été créées.

Un autre cas moins théorique : la gestion des stocks. Le script doit soutirer des stocks actuels une quantité commandée d'un certain article. Si vous utilisez l'algorithme suivant :
SELECT qte FROM table WHERE article=… AS qte_stock
UPDATE table  SET qte=$qte_stock - $qte_commandee WHERE article…
alors tout est faux ! Non seulement le problème est le même que ci-dessus, un jour on perdra un retrait, mais en plus vous faites deux requêtes là où une suffirait :
UPDATE table SET qte=qte-$qte_commandee WHERE article…
Cette requête est atomique. Peu importe que A ou B passe en premier. La base restera cohérente.

## *11.2 Un double impact sur la manière de coder*

Premièrement, on utilisera uniquement des opérations atomiques à chaque fois que possible. Concrètement, à chaque fois que la séquence SELECT / UPDATE se rencontrera, on essaiera de faire seulement l'update avec une clause WHERE adaptée. De toutes façons, en général, cela réduit le nombre de requêtes et donc cela limite le temps passé en transaction quand on peut les utiliser.

Deuxièmement, il y a des cas où seule une transaction permet d'assurer que la base ne sera pas corrompue. Prenons le cas d'un panier virtuel avec gestion des stocks où il est interdit de vendre si le stock n'est pas présent.
Empêcher la requête précédente de donner une quantité négative est très facile :
UPDATE table SET qte=qte-$qte_commandee WHERE qte >= $qte_commandee AND article…
Il suffit de vérifier que mysql_affected_rows()==$qte_commandee et non zéro pour savoir s'il restait en effet assez d'articles.
Donc pour un article, pas de problèmes. Si la quantité commandée est la même pour tous les articles, on pourrait essayer une opération ensembliste sur une liste d'article (WHERE article in ('ref1', 'ref2', … 'refn') ) mais il n'y a pas de raisons de forcer le client à acheter exactement la même quantité de chaque article.
Si on a plusieurs articles, appliquer cette requête posera problème : le premier article fait bien l'update, le deuxième aussi, mais pas le troisième : comme un autre client était lui même en train d'acheter le troisième article, il n'en reste plus assez. On fait quoi ? Rollback me direz-vous. Ah non, justement, on a pas les transactions…

Dans ce type de cas, il n'y a que deux solutions :
- on se voile la face et on espère très fort que ça n'arrivera pas. Cette approche peut parfaitement être acceptable dans certains cas : tout ceci n'est qu'une question de probabilités.
- il est nécessaire de gérer correctement. Alors il faut mettre en place un système de verrouillage d'accès à la table et re-créer un système de transactions/section critique à la main.

Notons que l'absence de transactions amène aussi un autre problème beaucoup plus difficilement détectable : que se passe-t-il si le script est interrompu ? Par exemple, toujours dans notre panier virtuel, après avoir impacté le stock, il faut faire une insertion dans la table COMMANDE mais le script est interrompu enter les deux : on a diminué le stock, mais la commande n'a pas été enregistrée… Là, sans transactions, impossible de garantir la cohérence. On ne pourra que détecter une erreur a posteriori (fichiers de logs, numéro de facture incrémenté mais absent de la base etc…)

# 12. L'accès concurrentiel dans le temps

Nous avons vu au chapitre précédent que même dans un temps très court, l'accès à une même donnée peut poser problème. C'est donc une évidence que si on allonge le temps pendant lequel la collision peut avoir lieu, les risques augmentent. Et là, les transactions ne peuvent rien.

Prenons le cas d'une application de travail collaboratif gérant des articles à publier. Plusieurs personnes différentes, chacune identifiée par un login, peuvent travailler sur un même article. Prenons donc ici deux personnes, travaillant sur la page des sports où l'équipe des Rouge a battu les Vert. Le rédacteur A veut ajouter le score. Il demande à modifier l'article : une requête SELECT va lui chercher le texte dans le SGBD, et lui propose dans une page web en formulaire. Il ajoute le score mais a un doute. C'était 2-0 ou 2-1 ? Il part chercher l'information. L'autre rédacteur s'aperçoit qu'il y a une faute d'orthographe énorme et inversion du nom des capitaines de chaque équipe. Il demande l'article à modifier, corrige les erreurs, clique sur Submit : l'article dans sa version corrigée est stocké par UPDATE. Le premier rédacteur a enfin l'information qui lui manquait : c'était bien 2-0. Il ajoute le score, clique sur Submit et… écrase définitivement les corrections de son collègue qui sont perdues. La faute d'orthographe restera là, le capitaine ne sera pas le bon.

Pour résoudre une telle situation, il y a deux méthodes, applicables ou non selon les cas.

- vérifier avant la mise à jour que personne n'a fait de modifications sur les données depuis le SELECT. Par exemple, faire transiter l'heure de dernière modification au Submit et la comparer à celle en base. Si c'est la même, personne n'a fait de modification. Sinon, il faut refuser celle-ci.

- poser un verrou applicatif sur les données et empêcher quiconque de faire le SELECT (pour éviter de faire le travail pour rien) et surtout de faire l'UPDATE (là est la garantie de l'intégrité). Un verrou applicatif pourrait par exemple se présenter sous la forme d'une table SQL dont la clef primaire est la

même que celle de la table des articles et qui contient d'autres données comme le login de celui qui détient le verrou, une heure d'expiration (attention la gestion en est plus complexe) etc…

Ce type de systèmes d'exclusion mutuelle dans le temps doit être mise en place uniquement dans les cas le nécessitant, et non systématiquement.

# 13. La séparation code - présentation

Contrairement à ce que d'aucuns sous-entendent, le modèle MVC est antérieur à Java et est une méthode de programmation qui n'est donc en rien propre à ce langage. Il est donc parfaitement possible d'en mettre en oeuvre les principes en PHP. Rappelons aussi qu'il n'est en rien nécessaire de faire de la POO pour ce faire. Nous n'allons pas ici rentrer dans le détail de MVC mais se focaliser sur la mise en pratique de l'un de ses concepts les plus utiles : on doit s'efforcer de séparer la présentation du résultat du code qui a permis de générer le résultat lui même.

Les avantages de la méthode sont évidents, en tous cas sur le papier.

- si on veut changer complètement la présentation de l'application, on peut éventuellement confier les fichiers correspondants à une tierce personne (infographiste). Pas de risques qu'il casse quoi que ce soit dans le code (par erreur ou parce que son éditeur favorit gère mal le code php). Il n'a pas besoin de farfouiller partout pour s'y retrouver.

- à partir du même résultat, si on veut multiplier les canaux de présentation (une sortie html, une sortie PDF, une sortie texte brut, une sortie xml par exemple) il n'y a que le code qui va générer le bon format à changer.

Mettons tout de suite en place les limitations de ces beaux principes avant d'en voir la mise en oeuvre concrète. Prenons l'exemple d'un site simple avec un menu en haut et les données au milieu de page, puis un bas de page contextuel. Ajoutons un tableau html dans les données centrales, avec du texte partiellement au dessus et en dessous. On doit découper les données en 5 parties : le menu, le texte du haut de page, la génération du tableau, le texte en dessous, le pied de page. Ceci correspond à une sortie html. Mais pour générer du PDF, on ne va pas nécessairement s'y prendre dans cet ordre là bêtement linéaire ! Donc même la logique de génération de la présentation peut parfois être spécifique à un format. De plus, si l'on change la présentation du site dans sa refonte, il faudra aussi peut-être revoir cette logique de génération.

## *13.1 Le "template du pauvre"*

... ou l'utilisation de PHP selon toute sa puissance. PHP est nativement un moteur de template.

Soit le tableau $data qui contient les variables reference, descriptif, prix_unitaire, qte_stock et qui est le résultat d'une requête vers MySQL. Dans le fichier principal, on peut écrire :

```
<?php
...
echo '<TABLE><TR><TD>R&eacute;f&eacute;rence</TD><TD>Description</TD>';
echo '<TD>Prix TTC unit&eacute;</TD><TD>Quantit&eacute;  disponible</TD></TR>';
while($data=mysql_fetch_array($resbuff))
{
echo "<TR><TD>${data['reference']}</TD> <TD>${data['descriptif']}</TD> ";
echo "<TD>${data['prix_unitaire']}</TD><TD>${data['qte_stock']}</TD></TR>";
}
echo '</TABLE>';
…
?>
```

Mais on peut aussi faire ceci :
```
require('haut_texte.txt');
while($data=mysql_fetch_array($resbuff))
 {require('affiche_ligne.php');}
require('bas texte.txt);
```

haut_texte.txt contient :
```
<TABLE><TR><TD>R&eacute;f&eacute;rence</TD><TD>Description</TD>
<TD>Prix TTC unit&eacute;</TD><TD>Quantit&eacute;  disponible</TD></TR>
```

affiche_ligne.php contient :
```
<?php
echo "<TR><TD>${data['reference']}</TD> <TD>${data['descriptif']}</TD> ";
echo "<TD>${data['prix_unitaire']}</TD><TD>${data['qte_stock']}</TD></TR>";
?>
```
ou même en faisant encore plus fin :
```
<TR><TD><?php echo $data['reference']; ?></TD> <TD><?php echo $data['descriptif']; ?></TD>
<TD><?php echo $data['prix_unitaire']?></TD><TD><?php echo $data['qte_stock']; ?></TD></TR>
```

On pourrait même utiliser la forme courte <?=$data['reference'] ?> au lieu de <?php echo $data ['reference'];?> mais attention à ne pas avoir des mauvaises surprises avec certains éditeurs qui pourraient mal interprêter cette syntaxe raccourcie.

Enfin bas_texte.txt contient simplement <TABLE>

Remarquons un point très important de sécurité : il est hors de question de faire quoi que ce soit dans ce type de fichiers autre que de l'affichage bête et méchant d'informations auxquelles on aura préalablement vérifié le droit d'accès. On peut bien entendu faire un test sur des variables pour modifier contextuellement la présentation mais aucune modification côté serveur ne doit être faite dans ce genre de scripts, sauf s'il est garantit que jamais il ne tournera avec register_globals à On : il est impossible d'initialiser les variables par sécurité dans ce type de cas.

## *13.2 Les moteurs de templates*

On peut le tourner dans tous les sens qu'on voudra : il faudra bien à un moment ou à un autre associer un lieu d'affichage à la valeur d'une variable PHP.  Là, on le fait simplement en afichant une variable de nom prédéterminé qui est en fait le nom des colonnes de la table ou de leur renommage par une clause AS dans le SELECT.

Donc le principe général des templates est en place : dans un fichier qui est lié à la présentation, on marque des emplacements. La syntaxe est souvent {NOM_EMPLACEMENT}.
affiche_ligne.php deviendra affiche_ligne.tpl et contiendra quelque chose ressemblant à :
```
{main}
{haut}
<TABLE><TR><TD>R&eacute;f&eacute;rence</TD><TD>Description</TD>
<TD>Prix TTC unit&eacute;</TD><TD>Quantit&eacute;  disponible</TD></TR>
{/haut}
{ligne}
<TR><TD>{REFERENCE}</TD> <TD>{DESCRIPTIF}</TD>
<TD>{PRIX_UNITAIRE}</TD><TD>{QTE_STOCK}</TD></TR>
{/ligne}
{bas}
</TABLE>
{/bas}
{/main}
```

Le fichier principal deviendra alors quelque chose ressemblant à :

```
$tpl=new template('affiche_ligne.tpl');
$tpl->parse('main.haut');
while($data=mysql_fetch_array($resbuff))
 {
 $tpl->assign($data);
 $tpl->parse('main.ligne');
 }
$tpl->parse('main.bas');
$tpl->parse('main');
```

Ce paragraphe n'entend pas être un cours complet sur les templates, mais vous permettra donc de savoir que la séparation entre la présentation et la logique est parfaitement possible et à quoi vous avez affaire si vous avez un jour à lire ce type de syntaxe. Notons que tous les moteurs de template ont leur propre syntaxe, et qu'elle diffère légèrement de l'un à l'autre. Certains ne parsent que sur appel explicite de la section du template, d'autres au contraire parsent tout par défaut et on doit désactiver les zone à ne pas parser. Presque tous sont dôtés d'un cache interne dont le fonctionnement est plus ou moins clair et intuitif.

### *13.3 Xml-xslt*

Il n'est pas question de traiter cette méthode dans ce cours car elle nécessite un ouvrage complet, mais pour mémo, sachez qu'il est aussi possible d'appliquer des règles xslt à des données xml grâce à un parseur, en l'occurrence Sablotron. Le fichier xml contient les données, elles sont lues en PHP grâce à la lib expat ou dom xml pour application des règles xslt.

## 14. Introduction aux regexps

Ce petit chapitre donne sommairement le principe des regexp et leur syntaxe de base. Des ouvrages entiers sont consacrés aux regexps, qui sont indépendantes du langage et utilisées dans beaucoup d'utilitaires comme grep ou awk par exemple.

regexp : Regular Expression en français dans le texte. La traduction "expression régulière" n'a pas plus de sens, donc utilisons un anglicisme et non un néologisme.

Une regexp est un motif dans le sens d'un motif de tapisserie ou de papier peint mural. Une fois ce motif définit, on peut le rechercher dans une chaîne de caractères de taille arbitraire. En général, une regexp est un assemblage de caractères assez abscon, par exemple ^.+@.+\\..+$ est une regexp. Nous verrons plus tard ce qu'elle représente comme motif.
Syntaxe de base :
a : qui contient le caractère a aa : qui contient deux fois le caractère aa
^a : qui commence par le caractère a a$ : qui se termine par le caractère a
a{1,3} : a ou aa ou aaa (de une à trois occurrences à la suite de a)
a{3,} : aaa ou aaaa ou n'importe quelle nombre de fois l'occurrence de a avec au moins 3
a{,5} : de zéro à 5 occurences de a
a{1,} : au moins une occurrence de a. Cette syntaxe est rigoureusement équivalente à : a+

Le caractère **.** possède un sens spécial : il remplace n'importe quel caractère.

Pour représenter le caractère point lui même, il faut donc l'échapper par \ et échapper \ qui a lui même un sens spécial. Donc \\. est le caractère **.** lui même.

Revenons donc à notre regexp : ^.+@.+\\..+$ et découpons là selon ce qui vient d'être décrit ci-dessus.

On reconnait ^ et $ en début et fin. On reconnait .+ qui est équivalent à .{1,} et veut donc dire "au moins un caractère, n'importe lequel". On reconnait \\. qui est le caractère point.
Cette regexp se lit ainsi : une chaîne qui commence par au moins un caractère n'importe lequel, suivit du caractère @, suivit d'au moins un caractère n'importe lequel, suivit du caractère point, se terminant par au moins un caractère n'importe lequel : qqchose@qqchose.qqchose
Bref, si une chaîne de caractères censée contenir une adresse email ne satisfait pas à cette condition simpliste, il est clair que ce n'en est pas une.
Bien entendu, il serait possible de raffiner cet exemple très simple.

Reste à signaler la notion de listes de valeurs, dont la syntaxe est la suivante :
[abc] : n'importe quel caractère a ou b ou c en une occurrence.
[A-Z] : le segment de caractères A à Z bref, toutes les lettres majuscules, en une occurence
[^a-z] : (noter le ^ en premier char) : tous les caractères SAUF le segment a à z c'est à dire tous les caractères qui ne soient pas une lettre minuscule, en une occurrence.
On peut les combiner : [A-Za-z]+ veut dire "au moins un caractère alphabétique".

Les fonctions PHP permettant d'utiliser les regexp commencent par ereg_ et sont disponibles dans le paragraphe XCI du manuel de PHP Regular Expression Functions (POSIX Extended) ainsi qu'en tant que regexp compatibles PERL pour ceux qui connaissent ce langage (et n'avaient donc pas besoin de lire ce paragraphe de toutes façons !)

# 15. PHP : visite guidée des fonctions et extensions les plus utilisées

La plateforme PHP est l'une des plus riches qui existent tant en fonctionnalités directement intégrées au langage, accessibles en compilant des modules supplémentaires déjà disponibles, ou en facilité d'ajout de modules propriétaires.
Cette partie se lit mieux en ayant en parallèle la table des matières du manuel de PHP disponible sur http://www.php.net/manual/en/ (remplacez /en/ par /fr/ pour la version française, mais vous perdrez l'ordre alphabétique).

N'écrivez jamais de fonctions de manipulation de tableaux, de gestion de chaînes de caractères (incluant les expressions régulières), de gestion de dates ou de tri avant de vérifier celles que propose nativement le langage, qui sont très complètes. La combinaison de deux ou trois fonctions existantes fera probablement directement ce dont vous avez besoin.

Comme déjà indiqué, la quasi totalité des SGBD du marché sont accessibles, ainsi que LDAP. Besoin de générer des images dynamiquement : utilisez la GD-lib. La manipulation du format GIF, longtemps interdite pour sombres raisons de patentes, a été récemment réintégrée. Générer une animation Flash à la volée : voyez la librairie Ming. De générer du PDF ? Plusieurs solutions s'offrent à vous, un comparatif est disponible sur http://www.afup.org/ (forum php 2002, conférence d'Olivier Plathey). De traiter du XML ? Choisissez entre l'extension qui tire partie de la lib-expat et l'extension Dom-XML. Ecrire un service SOAP ? Les classes correspondantes existent. D'envoyer un mail, de transférer un fichier par FTP vers un site distant, d'ouvrir une socket vers un serveur spécifique ? Utilisez directement les fonctions natives disponibles.  Sous Windows, l'extension COM permet de par exemple de générer des fichiers MS-Excel.

Et quand PHP ne vous suffit pas, ajoutez lui des fonctions natives : dès que vous avez une bibliothèque écrite en C (ou C++) vous pouvez l'ajouter au moteur Zend au prix d'un peu de codage en C grâce à l'API de Zend (chapitre V du manuel de PHP). On peut à ce titre citer par exemple l'extension de communication avec SAP http://saprfc.sourceforge.net/

On peut également citer les innombrables bibliothèques de fonctions ou de classes disponibles sur le net, ainsi que les projets complets. Certains sont très complets et fiables, d'une qualité professionnelle (citons par exemple phpMyAdmin). Mais ne vous leurrez pas, il y a aussi beaucoup de choses d'une

qualité plus que douteuse. Il s'agit de garder un esprit critique et de relire attentivement le code avant de l'utiliser en production, en particulier vis à vis de la sécurité/qualité du code.

# 16 Ressources

phpindex : http://www.phpindex.com/
usenet : fr.comp.lang.php et sa FAQ : http://faqfclphp.free.fr/
afup : http://www.afup.org/

# 17 Conclusion

Ingénieur EFREI promo 99, j'ai essayé de condenser six ans d'expérience professionnelle et cinq ans de développement PHP pour couvrir le maximum de cas réels de production bien éloignés de l'image absurde "php=page perso" que j'ai toujours refusée et dénoncée, notamment en étant l'un des acteurs majeurs de la création de l'Association Française des Utilisateurs de PHP (afup). En 2004, PHP est maintenant avec JSP et .NET l'une des trois plateformes les plus utilisées dans le monde du web dynamique.

Merci de me signaler toute erreur (faute de frappe ou autre). Je reste à disposition pour toute question, indiquez "php" quelque part dans l'objet de votre email(1) envoyé à john.gallet@saphirtech.com

(1) Non à la "gagadémie", un bon anglicisme est préférable à un mauvais néologisme. Et pendant que je suis dans les affirmations linguistiques, je rappelle qu'un 's' entre une voyelle et une consonne ne se prononce pas z mais s, on ne prononce donc pas "néologiZZZme" mais néologiSSSme.

# Table des matières