Cours de PHP 5

par Guillaume Rossolini (Tutoriels Web / SEO / PHP) (Blog)

Date de publication : 12 mai 2008

Dernière mise à jour : 4 mars 2009

Ce cours vous apprendra à vous initier à tous les aspects du langage PHP, depuis la syntaxe jusqu'aux meilleures techniques de programmation. De nombreux exemples facilitent la compréhension par l'expérience.

Nous verrons également d'autres aspects comme les méthodes de Test d'applications ainsi qu'une vision globale de la communauté PHP.

Vos commentaires sont les bienvenus : **g-rossolini@redaction-developpez.com**

Je remercie particulièrement **julp** pour avoir pris le temps de m'aider à finaliser ce cours.

PHP est un l'un des langages de script les plus actifs sur le Web. Il permet de créer principalement des pages Web HTML mais aussi d'autres types de contenu comme des images, des animations Flash, des documents PDF, *etc*.

Nous allons voir comment débuter dans ce langage et quels outils, quelles méthodes peuvent nous aider dans notre progression.

PHP peut être utilisé soit comme un langage de script répondant à des besoins simples et à court terme (c'est ce qui l'a fait connaître), soit comme un langage de programmation complexe permettant de mettre en place des applications d'entreprise (**programmation orientée objet**, **design patterns** *etc*.).

Certaines parties de ce cours semblent répéter ce qui est déjà documenté sur le site officiel de PHP. Je ne souhaite pas reformuler le site entier, mais aborder certains concepts à ma manière. En cas de désaccord entre mes explications et la documentation officielle, c'est cette dernière qui fait foi et je vous serais reconnaissant de m'en faire part.

À l'aube de l'an 2008, il n'est plus question de faire sa "**page Web perso**" comme c'était la mode il y a quelques années. Le **HTML** statique est mort depuis longtemps, tandis qu'aujourd'hui le dynamisme a pris son envol avec **AJAX** (nous y reviendrons). Plus personne ne code son site directement en HTML, mais utilise plutôt des scrits qui génèrent des pages HTML selon les informations qui sont en **base de données**.

Un langage de script permet d'utiliser une base de données pour construire un document HTML, qui n'est donc pas nécessairement identique à chaque consultation, sans que le webmestre n'ait eu à intervenir manuellement.

Charge au développeur de choisir son langage de prédilection : ASP, JSP, PHP, RoR... Nous allons nous attacher ici exclusivement à PHP.

PHP est un langage de script, c'est-à-dire que le code est enregistré sous forme de fichier texte sur le disque dur, et qu'il est exécuté à la demande par un programme chargé de l'interpréter. C'est habituellement l'internaute (par l'intermédiaire de son navigateur Web) qui demande l'exécution d'un script lorsqu'il consulte une page Web. La demande est reçue par le serveur Web (par exemple Apache HTTPD), qui se rend compte qu'il doit la sous traiter à PHP.

Notre objectif en tant que développeurs est de ne pas répéter le code source (ni le code HTML ni le code PHP, autant que possible). Nous allons utiliser des techniques permettant de "factoriser" le code source lorsque nous avons besoin d'un même bloc à plusieurs endroits, ainsi que d'autres techniques permettant à un collaborateur de savoir comment est structuré notre code avant même de l'avoir consulté.

PHP est maintenu par une communauté ouverte d'utilisateurs dont l'auteur original, Rasmus Lerdorf, fait encore partie. Tout le monde est invité à aider dans la mesure de ses disponibilités et de ses compétences. Aider le

projet peut prendre diverses formes comme la programmation du core ou d'extensions en langage C, une aide à la documentation, *etc*

.

Les contributeurs de PHP viennent d'horizons très différents. Certains sont de simples utilisateurs du langage qui ont voulu participer à son amélioration, d'autres sont des chefs d'entreprise ou des responsables informatiques désireux de s'assurer que le langage qu'ils utilisent est maintenu ou qu'il correspond à leurs besoins, d'autres encore sont des chercheurs (IBM *etc*.) souhaitant ouvrir de nouveaux horizons au langage...

La communication se fait par divers media. Les utilisateurs finaux peuvent simplement utiliser le **flux Atom des nouveautés**, et ceux qui s'intéressent au fonctionnement interne de PHP peuvent s'inscrire aux **mailing lists**

Une alternative à la mailing list "PHP Internals" est de suive les **Zend Weekly Summaries** rédigés de main de maître par **Steph Fox**

.

Si vous avez besoin d'aide, vous pouvez notamment utiliser **les forums PHP de Developpez** (en français) ou **les listes officielles** (en anglais), mais il en existe d'autres sur Internet. À vous de faire le choix.

Avant de participer ou de poser des questions à une communauté, apprenez-en toujours les habitudes. En effet, chaque communauté a des règles bien spécifiques, un fonctionnement unique. Entraver ces règles, ne pas comprendre ce fonctionnement peut vous placer dans des situations fâcheuses et qui feront perdre leur temps à tout le monde, vous le premier.

Afin de commencer sur de bonnes bases, je vous propose de vous choisir un environnement de développement qui vous convient. Vous ne saurez probablement pas avec certitude que vous avez choisi le plus adapté tant que vous ne connaîtrez pas un peu PHP ; cependant, des sélections et des comparatifs ont déjà été faits, je vous propose donc de vous y reporter.

Pour le moment, le plus important est que votre éditeur sache colorer le code PHP et HTML. Le reste est secondaire, mais qu'il vous donne un accès facile à la documentation sera très appréciable. Vous pourrez changer d'éditeur à tout moment sans aucune conséquence.

Il vous faudra également installer un serveur sur votre machine de développement.

Environnements de Développement Intégré (EDI) :

- **Les meilleurs éditeurs pour PHP** ;
- **Comparatif des meilleurs éditeurs de scripts PHP** ;
- **Tutoriels sur les meilleurs éditeurs de scripts PHP** ;
- **Sondage : Quel est l'éditeur que vous recommandez pour PHP ?**

Serveurs :

- **Les meilleurs serveurs "tout en 1"** (serveur Web + PHP + base de données + administration) ;
- **Tutoriels sur les serveurs pour PHP**

.

Afin d'effectuer les tests proposés au début de ce tutoriel, décompresser l'archive de PHP devrait suffire à vous permettre d'invoquer l'exécutable "php" (ie. php.exe sous Windows) depuis la ligne de commandes en lui donnant le nom du script en paramètre (pas besoin de serveur).

Le site officiel de PHP est extrêmement bien fait. La documentation de n'importe quelle fonction est accessible depuis l'URL : http://php.net/nom_de_la_fonction

N'oubliez pas de consulter les commentaires utilisateurs, car de très nombreux commentaires donnent des conseils très utiles.

Si vous utilisez Firefox, vous pouvez ajouter un mot-clef de recherche en suivant ces instructions :

- Aller sur **php.net** ;
- Cliquez droit **dans** la case de recherche ;
- Sélectionnez "Ajouter un mot clef pour cette recherche..." ;
- Donnez un raccourci à votre convenance, par exemple "p", et validez la boîte de dialogue ;
- Allez dans la barre d'adresse du navigateur et tapez "p echo" : vous êtes renvoyé sur la documentation de la
- function echo (en français).

PHP est un langage de script qui peut être utilisé de diverses manières :

- Pour une interface Web : c'est l'utilisation la plus courante ;
- En ligne de commandes ("CLI" pour *Command Line Interface*) ;
- Pour produire une interface desktop ("GUI" pour *Graphical User Interface*).
-


Certains scripts prévus pour le Web peuvent parfaitement être exécutés en lignes de commandes. En revanche, les scripts GUI ont une structure totalement différente.

Il y a également divers modes d'exécution :

- Mode interprété (compilé à la volée) : c'est l'utilisation la plus courante ;
- Mode pré compilé (nécessite une extension, qui est souvent payante) ;
- Mode compilé (nécessite certaines manipulations, n'est pas adapté au Web).
-


La différence entre les modes d'exécution se voit principalement dans les performances du programme. En tout état de cause, le fonctionnement du programme n'est pas altéré par son exécution dans l'un ou l'autre mode.

```php
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr-FR" lang="fr-FR">
<head>
    <title>Hello World!</title>
    <meta http-equiv="content-type" content="text/html; charset=iso-8859-1" />
</head>
<body>
<?php
if (empty($_GET['user']))
{   echo htmlentities('Bonjour, invité', ENT_QUOTES, 'iso-8859-1');

}
else
{   echo htmlentities('Bonjour, '.$_GET['user'], ENT_QUOTES, 'iso-8859-1');

}
```

```
?>
</body>
</html>
```

```
http://localhost/hello-world.php
```

```
http://localhost/hello-world.php?user=Yogui
```

```
<?php

if ($argc > 1)

{   echo 'Bonjour, '.$argv[1];

}
else

{   echo 'Bonjour, invité';

}
```

```
php C:\Web\hello-world.php
```

```
php C:\Web\hello-world.php Yogui
```

Je ne vais pas m'attarder sur cette utilisation car elle est à la fois complexe et pas encore en version stable.

PHP est combiné au framework GTK pour développer des applications desktop. La dernière version de PHP-GTK est la version 1.0.2, qui utilise GTK1 et PHP4, deux versions complètement obsolètes aujourd'hui. Les bénévoles de PHP travaillent donc à une version plus intéressante, PHP-GTK2+, qui utilise GTK2+ et PHP5.

Les nouveautés promises sont très alléchantes, notamment l'arrivée de la conception objet PHP5 qui est bien plus mature que celle de PHP4. Le projet semble cependant avoir du mal à être finalisé.

Plus d'informations sur le fonctionnement de PHP-GTK dans **le tutoriel par Xaviéra Autissier et Jean-Marc Richon**.

Dans mes exemples, le logiciel serveur utilisé est Zend Core ; le chemin local de DOCUMENT_ROOT est : "C:\Web\online\http".

PHP permet d'exécuter des scripts de deux manières principales : en tant que code incrusté dans une page Web ou bien comme un script système. Je vous recommande pour commencer d'utiliser les lignes de commande (MS-DOS, shell, *etc*.) pour lancer vos premiers scripts, car cela vous évite d'avoir à installer Apache httpd, c'est donc *a priori* plus simple.

Pour exécuter un script PHP depuis les lignes de commande, il faut invoquer le binaire "php" en lui donnant en paramètre le nom du script.

```
cd "C:\Program Files\Zend\Core\bin"
php "C:\Web\online\http\tests\error.php"
```

Si vous avez déjà un serveur Web configuré avec PHP, par exemple en ayant installé l'un des outils "tout-en-un" mentionnés auparavant, alors vous pouvez charger l'URL suivante dans votre navigateur : **http://localhost/tests/error.php** en supposant que votre script se situe ici : {DOCUMENT_ROOT}/tests/error.php

Pour afficher le contenu des variables, j'utiliserai principalement les fonctions **echo** et **print_r()**. S'il s'agit de contenus complexes (tableaux ou objets), en lignes de commandes l'affichage ne posera pas de problème ; en revanche, dans une page Web vous pourriez avoir des surprises, il vous faudra donc utiliser la balise HTML **<pre>** pour forcer la mise en page fixe.

```
<?php
print_r(array('début', 'milieu', 'fin'));
?>
```

```
(
```

```
<pre>
<?php
print_r(array('début', 'milieu', 'fin'));
?>
</pre>
```

```
(
```

Commençons par un exemple, cela nous permettra de nous immerger plus facilement dans la théorie.

```
<?php
echo "Hello world!";
?>
```

Vous pouvez voir certaines similitudes avec le HTML, par exemple l'utilisation d'une balise de début "<?php" et d'une balise de fin "?>". Toute portion de code PHP doit donc être placée entre une balise d'ouverture et une balise de fermeture.

Si votre code n'est pas correctement écrit, PHP arrête l'exécution du script en lançant une erreur :

```
1. <?php
2. echo
3. ?>  ;
```

```
Parse error: syntax error, unexpected ';' in C:\Web\online\http\tests\error.php on line 2
```

L'erreur donne de très nombreuses informations permettant de résoudre le problème. Ici par exemple, on nous informe qu'une erreur de syntaxe se trouve près d'un point virgule, et que ce point virgule n'est pas ce qui était attendu. En effet, "echo" sert à afficher quelque chose, il manque donc le nombre ou le texte à afficher.

Si vous regardez dans le fichier de log de votre serveur Web, vous remarquerez sans doute que cette erreur s'y trouve également :
Sun Nov 25 17:21:08 2007] [error] [client 127.0.0.1] PHP Parse error: syntax error, unexpected ';' in C:\\Web\\online \\http\\tests\\error.php on line 3, referer: http://localhost/tests/

Nous reviendrons plus tard sur ces détails, mais il est important de savoir dès à présent que les erreurs du code sont faciles à retrouver. Votre éditeur de code peut lui aussi (sans doute) vous aider à ce sujet.

L'affichage n'est pas ce qui fait l'intérêt d'un langage de script par rapport au HTML standard, néanmoins c'est un aspect essentiel. PHP permet d'afficher à l'aide des mots clefs "echo" ou bien "print", selon le choix du développeur.

Les boucles sont l'un des attraits des langages de script, car c'est quelque chose que le HTML ne peut pas faire.

```
<?php
for($i=0; $i<1000; ++$i)

{   echo $i;

}
```

Il suffit de mettre n'importe quelle suite d'actions à la place du "echo $i". L'une des applications classiques d'une boucle est d'afficher certains éléments extraits d'une base de données.

Afin de réduire la quantité de code pouvant parasiter la compréhension, dans cette page j'utiliserai principalement des scripts CLI.

Cette partie est nécessaire mais tout lire assidûment dès le départ n'est peut-être pas la meilleure méthode pour apprendre PHP. Pour votre lecture, je vous propose de lire rapidement la syntaxe dans un premier temps, et de revenir à ces paragraphes au fur et à mesure que d'autres concepts apparaissent ou que vous avez des doutes syntaxiques.

Un bloc de code PHP est délimité par une balise d'ouverture <?php et une balise de fermeture ?>.

Un bloc de code PHP peut être le seul bloc de code dans le fichier, ou bien être mélangé à d'autres langages (par exemple HTML ou XML).

```php
<?php
$image = imagecreatetruecolor(100, 50);
$text_color = imagecolorallocate($image, 255, 0, 0);
imagestring($image, 1, 5, 5, 'Hello, world!', $text_color);
header('Content-Type: image/png'); //type MIME
imagepng($image);
```

```php
<?php
/*
  Récupération des variables $title, $charset et $body
  par exemple depuis un formulaire ou une BDD
*/
header('Content-Type: text/html; charset='.$charset); //type MIME
?>
<?xml version="1.0" encoding="<?php echo $charset; ?>"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr-FR" lang="fr-FR">
<head>
    <title><?php echo $title; ?></title>
    <meta http-equiv="content-Type" content="text/html; charset=<?php echo $charset;?>" />
</head>
<body>
<?php echo $body; ?>
</body>
</html>
```

La balise de fermeture de code PHP ?> n'étant pas nécessaire lorsque le script se termine (tel que dans l'exemple d'image ci-dessus), nous l'utiliserons le moins possible. Cela permet d'éviter certaines erreurs, surtout en débutant. Elle est nécessaire uniquement si l'on souhaite alterner PHP et un autre langage.

Vous pouvez varier la présentation du code source (ajouter des espaces, des tabulations et des sauts de ligne) sans que cela affecte l'exécution.

Une "instruction" est une portion de code se terminant par un point virgule (;). Par convention, on la représente sur une ligne de code propre.

```php
<?php
```

```php
echo "Hello World!";
echo "Je m'appelle Guillaume Rossolini";
```

L'**indentation** du code désigne l'utilisation des espaces et des tabulations pour structurer visuellement le code en niveaux. Cela ne change aucunement le résultat de l'exécution, mais facilite la relecture du code par un collaborateur ou par soi-même. Un code très mal indenté peut induire le programmeur en erreur et causer de grands dysfonctionnements lors d'opérations de maintenance.


Adrien Pellegrini vous propose ses conseils : Guide de style pour bien coder .

Le code d'une instruction peut être groupé au moyen de parenthèses "()" afin de faciliter la lecture ou de forcer les priorités. PHP définit des priorités même s'il n'y a pas de parenthèses, mais il est parfois nécessaire de modifier les priorités imposées par le langage. Le principe est exactement le même qu'en mathématiques :

```php
<?php
echo 1+2*3; //affiche "7"
echo (1+2)*3; //affiche "9"
```


Utilisez les parenthèses pour clarifier l'ordre dans lequel les opérations doivent avoir lieu, mais ne surchargez pas votre code pour autant. Ne mettre aucune parenthèse peut porter à confusion (tout le monde ne connaît pas par coeur la priorité de chacun des opérateurs) mais trop de parenthèses rend le code confus. Il faut donc trouver un équilibre.

Les blocs d'instructions (une à plusieurs instructions) peuvent être identifiés par des accolades "{}". Dans d'autres langages, cela permet de limiter la portée (*scope*) d'une variable, mais en PHP cet effet n'est pas pris en compte.

En revanche, les accolades définissent la portée d'une structure de contrôle (*cf.* plus loin) et les limites du corps d'une fonction.

```php
<?php
function additionner($x, $y)

{   //nous sommes dans le corps de la fonction "additionner"
    return $x + $y;

}
//ici, nous ne sommes plus dans le corps de la fonction "additionner"
```


Les accolades sont facultatives dans certaines situations (par exemple s'il n'y a qu'une seule instruction dans un "if") mais je vous recommande de systématiquement les mettre, car cela ne prend pas beaucoup de temps alors que la valeur ajoutée est très grande (lisibilité du code).

Les commentaires en PHP permettent d'introduire des sections de texte qui ne sont pas exécutées. L'utilisation principale est pour la maintenance, l'évolution du code, afin de permettre au développeur d'introduire des commentaires explicites sur ses intentions.

Ils prennent généralement deux formes :

```php
<?php
//Commentaire sur une ligne
```

```php
<?php
/*
Commentaire sur plusieurs lignes
*/
```

Il existe également une forme avec un caractère dièse (#) au lieu du double slash (//), mais cette syntaxe n'est pas conseillée car ce type de commentaire est moins courant parmi les langages de programmation. Sachez simplement qu'elle existe afin de ne pas vous étonner de la voir des scripts écrits par d'autres personnes.

Le commentaire **//** est parfois utilisé en fin d'instruction pour expliquer son utilité.
Le commentaire **/* */** est extensivement utilisé par les générateurs de documentation automatique de code.

```php
<?php

/**
 * Chien
 *
 * @author Yogui
 * @copyright Guillaume Rossolini (c) 2007
 * @version 8.12.2007
 * @access public
 */
class Chien

{   private $âge; //en années

    /**
     * Chien::__construct()
     *
     * @param int $âge
     */
    public function __construct($âge)
    {
        $this->âge = (int)$âge;
    }

}
```

PHP dispose des opérateurs classiques pour effectuer des calculs :

```php
<?php
echo 1+1; //addition
echo 1-1; //soustraction
echo 1*1; //multiplication
echo 1/1; //division
echo 1%1; //modulo (reste de la division)
```

Les opérateurs d'affectation permettent de donner une valeur à une variable.

```php
<?php
```

```php
$x = 5;
$x += 1; //ajoute à la valeur existante
$x -= 2; //soustrait à la valeur existante
$x *= 3; //multiplie la valeur existante
$x /= 4; //divise la valeur existante
echo $x; //affiche "3"
```

```php
<?php
$str = 'texte';
$str .= ' additionnel'; //concatène à la suite de la chaîne existante
echo $str; //affiche "texte additionnel"
```

L'opérateur "**==**" est un opérateur de comparaison de valeurs. Il ne tient pas compte du type de la valeur, puisque PHP est un langage à typage faible. L'opérateur "**!=**" est l'inverse de "**==**".

```php
<?php
echo 1==1; //affiche "1"
echo 1==2; //n'affiche rien puisque c'est faux
echo 1==1.0; //affiche "1" puisque ce sont deux valeurs entières équivalentes
echo '1'==1.0; //affiche "1" puisque la chaîne "1" est équivalente à l'entier numérique "1"
echo '1'=='1.0'; //affiche "1" puisque leur valeur numérique évalue à la même valeur entière
```

L'opérateur "**===**" est le même que le précédent, sauf qu'il n'effectue pas de conversion de type. Il est donc plus rapide que l'opérateur "**==**" et il ne donne pas les mêmes résultats. L'opérateur "**!==**" est l'inverse de "**===**".

```php
<?php
echo 1===1; //affiche "1" puisque les deux valeurs sont identiques en valeur et en type
echo 1===2; //n'affiche rien puisque c'est faux
echo 1===1.0; //n'affiche rien puisque le type diffère (int et float)
echo '1'===1.0; //n'affiche rien puisque le type diffère (string et float)
echo '1'==='1.0'; //n'affiche rien puisque les deux chaînes ne sont pas égales
```

Voir également :

- **Récapitulatif sur la comparaison de types à l'aide de l'opérateur ==**
- **Récapitulatif sur la comparaison de types à l'aide de l'opérateur ===**
- **Récapitulatif sur la comparaison de types à l'aide des fonctions PHP prédéfinies**

Les opérateurs "**<>**" et "**!=**" sont identiques, ils permettent à des développeurs issus de différents horizons de s'adapter sans problème à PHP. Ils fonctionnent aussi bien pour les nombres que pour les chaînes.

Les opérateurs "**<**", "**>**", "**<=**" et "**>=**" sont assez classiques pour ne pas nécessiter d'explications. Ils fonctionnent aussi bien pour les nombres que pour les chaînes.

L'opérateur **instanceof** permet de comparer les classes :

```php
<?php
if ($objet instanceof MaClasse) //attention, ne pas mettre le nom de classe entre guillemets

{   echo 'même classe ou classe fille';
```

```
}
if ($objet_1 instanceof $object_2)

{   echo 'même classe ou classe fille';

}
```


L'opérateur instanceof fonctionne également avec les classes héritées et avec les interfaces.

L'opérateur ternaire "?" est une alternative : "est-ce vrai **?** *valeur si oui* **:** *valeur sinon*".

```
<?php
echo 1==1 ? 'vrai' : 'faux'; //affiche "vrai"
echo 1==2 ? 'vrai' : 'faux'; //affiche "faux"
```

On utilise souvent les parenthèses pour faciliter la lecture de cet opérateur.

```
<?php
echo (1==1) ? 'vrai' : 'faux';
echo (1==2) ? 'vrai' : 'faux';
```


Cet opérateur est plus lent à l'exécution qu'une structure conditionnelle if/else (cf. nos benchmarks). De plus, il peut rendre le code illisible s'il est utilisé trop souvent ou dans des instructions trop longues. Il est donc à utiliser avec précaution, et exclusivement dans des instructions très courtes (notamment sans appel de fonction).

Cet opérateur peut, sous certaines conditions, être utilisé pour affecter des variables. L'exemple qui suit montre un exemple d'affectation (sécurisée et sans erreur) de la variable **$id** :

```
<?php
//la valeur entière de $_GET['id'] ou zéro si la variable n'existe pas
$id = isset($_GET['id']) ? (int)$_GET['id'] : 0;
```

Depuis PHP 5.3 et 6.0, il existe une alternative abrégée :

```
<?php
$id = $_GET['id'] ?: 0; //$_GET['id'] ou zéro si la variable est vide
```

Le problème de cette dernière approche est que dans de très nombreuses situations, elle met le script en danger. Dans l'exemple ci-dessus par exemple, nous n'effectuons aucune vérification sur le type ou le contenu de la variable... Il convient donc d'utiliser la version abrégée de l'opérateur ternaire uniquement dans des situations parfaitement maîtrisées. De plus, écrire par exemple "**echo isset($_GET['id']) ?: 0;**" n'aurait aucun sens, ce qui peut facilement causer des alertes **E_NOTICE**

.

++ : Augmenter de 1 ;
- **--** : Diminuer de 1.
-

```
<?php
$x = 5;
```

```php
echo ++$x; //incrémente puis affiche 6
echo $x; //affiche 6

$x = 5;
echo $x++; //affiche 5 puis incrémente
echo $x; //affiche 6

$x = 5;
echo --$x; //réduit puis affiche 4
echo $x; //affiche 4

$x = 5;
echo $x--; //affiche 5 puis réduit
echo $x; //affiche 4
```

Ces opérateurs fonctionnent également avec les chaînes.

Ces opérateurs permettent d'effectuer des opérations de bits, très utilisées par exemple pour les variables de configuration. Les opérations logiques sont bien plus rapides que les opérations décimales, il peut donc dans certains cas être intéressant de les utiliser.

Les opérateurs ayant un équivalent ensembliste :

- & (and) : Équivalent à une **intersection** d'ensembles (les bits à 1 dans les deux ensembles donnent 1, les autres donnent 0) ;
- | (or) : Équivalent à une **union** d'ensembles ("ou" inclusif : les bits à 1 dans au moins l'un des ensembles donnent 1, les autres donnent 0) ;
- ^ (xor) : Équivalent à une **union** d'ensembles ("ou" exclusif : les bits à 1 dans exactement l'un des ensembles donnent 1, les autres donnent 0) ;

```php
<?php
echo 0&0; //affiche "0"
echo 0&1; //affiche "0"
echo 1&1; //affiche "1"

echo 0|0; //affiche "0"
echo 0|1; //affiche "1"
echo 1|1; //affiche "1"

echo 0^0; //affiche "0"
echo 0^1; //affiche "1"
echo 1^1; //affiche "0"
```

```php
<?php
echo 1&5; //affiche "1"
echo 1|5; //affiche "5"
echo 1^5; //affiche "4"
```

Si on calcule l'union des équivalents binaires de 1 et 4 (soit "un ou quatre inclusif"), on obtient l'équivalent de 5 :

```
001
100 |
101 =
```

```php
<?php
echo 1|4; //affiche "5"
```

L'union de 4 et 5 vaut également 5 :

```
100
101 |
101 =
```

```php
<?php
echo 4|5; //affiche "5"
```

En revanche, par la différence logique de 1 et 4 (soit "un et quatre"), on obtient 0 :

```
001
100 &
000 =
```

```php
<?php
echo 1&4; //affiche "0"
```

Mais l'intersection de 4 et 5 vaut 4 :

```
100
101 &
100 =
```

```php
<?php
echo 4&5; //affiche "4"
```


C'est par exemple cette méthode qui est utilisée pour le niveau d'erreurs géré par PHP (directive error_reporting du fichier php.ini).

Les autres opérateurs de bits :

- $x : Inversion du positionnement des bits dans **$x** (not) ;
- $x **<<** $n : **$x** est multiplié **$n** fois par 2 (décalage à gauche) ;
- $x **>>** $n : **$x** est divisé **$n** fois par 2 (décalage à droite).
-


Une manière optimisée d'effectuer des divisions ou des multiplications par 2 est d'utiliser le décalage de bits. Cependant, cela ne peut fonctionner que sous certaines conditions, par exemple se limiter à des nombres de 32 bits pour les systèmes 32 bits.

Comme dans tout langage de programmation, ces opérateurs permettent de vérifier plusieurs conditions à la fois dans un même test. On peut écrire les opérateurs "**and**" et "**or**" en minuscules ou en majuscules.

```php
<?php
if ($variable > 2 and $variable < 10)

{   echo 'valeur entre 2 et 10 (exclus)';

}
```

```php
<?php
if ($variable > 2 && $variable < 10)

{   echo 'valeur entre 2 et 10 (exclus)';

}
```

```php
<?php
if ($variable > 2 or $variable < 10)

{   echo 'valeur supérieure à 2 ou inférieure à 10 (exclus)';
```

```
}
```

```
<?php
if ($variable > 2 || $variable < 10)

{   echo 'valeur supérieure à 2 ou inférieure à 10 (exclus)';

}
```

- "**&&**" et "**and**" sont identiques sauf pour la priorité qui leur est attribuée ;
- "**||**" et "**or**" sont identiques sauf pour la priorité qui leur est attribuée.
- "

Ma recommandation est d'utiliser "and" et "or" (au détriment de && et ||) puisqu'ils portent la sémantique en eux et qu'ils ne prêtent pas à confusion avec les opérateurs de bits.

Il s'agit d'opérateurs dits "paresseux" : avec l'opérateur "or", une seule condition vraie dans la liste permet d'avérer l'ensemble de l'instruction ; avec "and", une seule condition fausse dans la liste permet de refuser l'ensemble de l'instruction.

```
<?php
//différence entre "&&" et "and"
var_dump(0 and 0 || 1); // FALSE car équivalent à : 0 and (0 or 1)
var_dump(0 && 0 || 1); // TRUE car équivalent à : (0 and 0) or 1

//différence entre "||" et "or"
var_dump(1 or 0 and 0); // TRUE car équivalent à : 1 or (0 and 0)
var_dump(1 || 0 and 0); // FALSE car équivalent à : (1 or 0) and 0
```

C'est le même principe que les opérateurs antérieurs, mais avec les tableaux :

- " : Union de tableaux ;
- "**==**" : Les mêmes paires clef/valeur ;
- "**===**" : Les mêmes paires clef/valeur, dans le même ordre et de même type ;
- "**<>**" ou **!=** : Au moins une paire clef/valeur ne correspond pas ;
- **!==** : Au moins un triplet clef/valeur/type ne correspond pas.
-

L'arobase @ sert à contrôler l'affichage des erreurs. L'utiliser est considéré comme une mauvaise pratique, car cela suppose que votre code n'est pas sécurisé, qu'il ne fait pas les contrôles nécessaires. Par ailleurs, cet opérateur empêche PHP d'enregistrer l'erreur dans un fichier de log (errors.log), ce qui vous oblige à enregistrer vous-même l'erreur dans un log, ce qui finalement ne fait pas gagner de temps et ne simplifie pas le code. De plus, il faut activer une certaine directive de configuration pour avoir accès aux messages d'erreur si on utilise cet opérateur.

Rappel : Vous devriez toujours garder un log des erreurs. Consultez le log de temps en temps pour améliorer la sécurité de votre serveur et de vos applications.

L'opérateur "apostrophe arrière `" (ou "accent grave") permet d'exécuter une commande système (shell). Utiliser cet opérateur revient à utiliser la fonction **shell_exec()**, qui est moins intéressante que la fonction **exec()** de par le manque de paramètres.

Dès l'école primaire, on nous apprend à ne pas additionner des choux avec des carottes lorsque l'on apprend les "unités" en mathématiques. Les "types" sont exactement la même chose en programmation, ils ne sont pas nécessairement compatibles entre eux. Il existe cependant des moyens de convertir un type en un autre, tout comme on peut convertir des grammes en kilogrammes.

Un type définit une valeur, il permet de savoir comment on doit la traiter. PHP ne doit pas, ne peut pas traiter de la même manière un nombre et du texte, tout comme des litres et des grammes ne peuvent pas être utilisés de la même manière. Par exemple, additionner deux nombres est une opération mathématique, tandis qu'additionner des ensembles de valeurs revient à trouver leur union (ensemble des deux). La même opération ne doit pas être effectuée de la même façon sur deux types différents.

Les types sont définis à la fois pour aider le programmeur à connaître les données qu'il manipule (sécurité) et pour optimiser la gestion de la mémoire par l'interpréteur PHP. Il convient donc de connaître les différents types disponibles en PHP. Nous reviendrons plus loin sur leurs utilisations.

Les types PHP sont :

- **boolean** : un contraste "vrai" ou bien "faux", "blanc" ou "noir", "1" ou "0", "yin" ou "yang"... ;
- **integer** : une valeur numérique entière ;
- **double** : une valeur numérique flottante (à virgule) ;
- **string** : une chaîne de caractères (texte) ;
- **array** : un tableau (ensemble de valeurs) ;
- **object** : un objet (instance de classe) ;
- **resource** : une ressource (type abstrait, inutilisable par le programmeur, utilisé uniquement pour des
- fonctions) ;
- **NULL** : un type spécial qui désigne l'absence de valeur.

PHP est un langage à types dynamiques. Cela signifie que l'on ne peut pas définir de manière fixe le type d'une variable pour toute la durée de l'exécution du script ; au contraire, on parle de type juggling, à savoir qu'une variable peut changer de type selon les besoins (cf. exemples plus loin).

La chaîne de caractères est le moyen classique pour désigner du texte en programmation.

Dans de nombreux langages de programmation (ainsi qu'en bases de données), il faut déclarer la longueur de la chaîne (nombre de caractères dans le texte) avant d'affecter du texte à une variable. PHP reste cependant très flexible dans la gestion de la mémoire, cette déclaration n'est donc ni utile ni possible ici.

Une chaîne de caractères peut s'écrire de diverses manières en PHP, chacune utilisant un "délimiteur" bien précis :

```php
<?php
//Délimitation par des guillemets :
echo "Hello World!";

//Délimitation par des apostrophes :
echo 'Hello World!';

//Délimitation par la syntaxe HereDoc :
$string = <<<END
Hello World!
END;
```

```php
echo $string;

//Délimitation par la syntaxe NowDocs :
$string = <<<'END'
Hello World!
END;
echo $string;


//Caractère $ avec la syntaxe HereDoc :
$string = <<<END
Le signe \$ doit être échappé : \$var
END;
echo $string;

//Caractère $ avec la syntaxe NowDocs :
$string = <<<'END'
Le signe $ peut être utilisé : $var
END;
echo $string;
```

Les deux premières formes sont les plus communes. La 3° (HereDoc) est très largement moins utilisée à cause de sa complexité, ce qui est dommage car elle offre certains avantages. La 4° (NowDocs) est encore en discussion pour PHP 5.3.

La syntaxe des guillemets permet d'utiliser sans crainte les apostrophes, mais tout se complique dès que l'on souhaite utiliser des guillemets :

```php
<?php
echo "Voici un exemple d'apostrophe";
echo "Voici un exemple de \"guillemets\"";
```

La syntaxe des apostrophes permet d'utiliser des guillemets dans le texte, mais nous ennuie avec les apostrophes :

```php
<?php
echo 'Voici un exemple de "guillemets"';
echo 'Voici un exemple d\'apostrophe';
```

Si l'on souhaite pouvoir utiliser à la fois des guillemets et des apostrophes dans un même texte, plusieurs solutions s'offrent à nous :

```php
<?php
echo "Voici un exemple d'apostrophe suivi de \"guillemets\"";
```

```php
<?php
echo 'Voici un exemple d\'apostrophe suivi de "guillemets"';
```

```php
<?php
echo <<<EOT
Voici un exemple d'apostrophe suivi de "guillemets"
EOT;
```

```php
<?php
echo "Voici un exemple d'apostrophe" . ' suivi de "guillemets"';
```

Charge à chaque développeur de voir la syntaxe qui l'intéresse le plus pour déclarer une chaîne.

Les différences entre les délimiteurs de chaînes (guillements ou apostrophes) a été évoquée plus en profondeur par Pierre-Baptiste Naigeon : **Apostrophes ou guillemets : lesquels choisir ?**

Le point permet de **concaténer** des chaînes, c'est-à-dire de les ajouter les unes à la suite des autres. L'opérateur de concaténation n'est disponible que pour les chaînes de caractères, il ne l'est pour aucun autre type. Bien entendu, un point situé à l'intérieur des délimiteurs n'est pas un opérateur de concaténation mais un simple point, il faut qu'il soit positionné *entre* deux chaînes pour qu'il joue le rôle d'opérateur.

Il existe également une syntaxe à base d'accolades pour accéder aux caractères d'une chaîne : echo $string{8}; affiche le caractère de $string à la position 8 (c'est-à-dire le 9° caractère).

```php
<?php
$user = 'Yogui';
echo $user; //affiche "Yogui"
echo $user[2]; //affiche "g" : gestion de la variable comme d'un tableau de caractères
echo $user{2}; //affiche "g" : gestion de la variable comme d'une chaîne (string)
echo substr($user, 2, 1); //affiche "g"
```

Les chaînes doivent toujours être traitées de manière spécifique, au cas par cas. Par exemple lors de l'affichage dans une page Web, il faut leur appliquer htmlentities() avant echo. Quelle que soit la destination de la chaîne, prenez soin de toujours appliquer la fonction de conversion adaptée.

En PHP, il n'y a pas toujours de distinction entre "entier" et "flottant". Plus précisément, PHP effectue des changements de type selon la nécessité :

```php
<?php
echo 1.2;
echo 1.0;
```

```
1.2
1
```

Cependant, le type est conservé :

```php
<?php
var_dump(1.2);
var_dump(1.0);
var_dump(1);
```

```
float(1.2)
float(1)
int(1)
```

Il existe quelques subtilités au sujet des nombres à virgule flottante. L'article suivant permet d'éclaircir certains détails : http://docs.sun.com/source/806-3568/ncg_goldberg.html

- **NULL** : Valeur/variable vide ou inexistante ;
- **resource** : Par exemple une variable permettant d'identifier une connexion à une base de données ;
- **object** : Utilisé en Programmation Orientée Objet (POO, *cf.* plus loin).

```php
<?php
$file = fopen(__FILE__, 'r');
echo $file;
var_dump($file);
fclose($file);
```

```
Resource id #3
resource(3) of type (stream)
```

```php
<?php
var_dump(new stdClass());
```

```
object(stdClass)#1 (0) { }
```

La fonction **var_dump()** affiche le type d'une variable et son contenu (ainsi que sa taille si c'est une chaîne). Elle s'applique aussi bien aux variables scalaires qu'aux objets, tableaux, ressources...

```php
<?php
var_dump("Voici du texte...");
```

```
string(17) "Voici du texte..."
```

```php
<?php
var_dump(1.2);
```

```
float(1.2)
```

```php
<?php
var_dump(1.0);
```

```
float(1)
```

```php
<?php
var_dump(1);
```

```
int(1)
```

La fonction **gettype()** permet de déterminer le type d'une valeur, ou **get_resource_type()** si c'est une ressource. Il existe également une fonction **is_*()** spécifique pour chaque type.

```php
<?php
echo gettype($variable); //affiche le type de la variable
echo get_resource_type($variable); //affiche le type de la ressource
```

Valeurs de retour possibles pour gettype() :

- boolean
- integer
- double
- string
- array
- object
- resource
- NULL
-

La fonction **settype()** permet de modifier le type d'une variable pendant l'exécution du programme. Des fonctions spécifiques **intval()**, **floatval()** et **strval()** permettent d'effectuer la même opération, et il est possible d'appliquer des "cast" à la mode du langage C.

```php
<?php
$x = 5.2;
var_dump($x);
settype($x, 'int');
var_dump($x);
```

```
float(5.2)
int(5)
```

```php
<?php
var_dump(intval('5.2 ans'));
var_dump(floatval('5.2 ans'));
var_dump(strval('5.2 ans'));
```

```
int(5)
float(5.2)
string(7) "5.2 ans"
```

On appelle "transtypage" l'opération qui consiste à modifier le type d'une valeur de manière ponctuelle. Cela permet par exemple d'obtenir la valeur entière d'un nombre à virgule, ou bien au contraire de considérer comme un nombre entier comme étant un nombre à virgule.

```php
<?php
var_dump((int) 5.2);
var_dump((bool) 5.2);
var_dump((float) 5);
var_dump((string) 5.2);
var_dump((array) 5.2);
var_dump((object) 5.2);
```

```
int(5)
bool(true)
float(5)
string(3) "5.2"
array(1) {
```

```
  [0]=>
  float(5.2)

}object(stdClass)#1 (1) {
  ["scalar"]=>
  float(5.2)

}
```

Valeurs possibles pour settype() :

- bool
- int
- float
- string
- array
- object
- null

Les paramètres possibles de settype() et les retours possibles de gettype() sont différents pour des raisons historiques de l'évolution du langage.

Une variable PHP est identifiée par le signe dollar ($) suivi d'une lettre puis d'une suite de lettres, chiffres et traits soulignés (_).

Par convention, un nom de variable ne commence pas par une majuscule. S'il faut plusieurs mots pour composer le nom, ils sont habituellement séparés par des soulignés (_).

```php
<?php
$variable; //ok
$variable_2; //ok
$Variable; //ok mais pas conventionnel (majuscule)
$2; //nom incorrect
$2 */-+$% = 'essai'; //nom incorrect
```

```php
<?php
$var = 1;
$Var = 2;

// $var et $Var sont deux variables distinctes :
echo $var; //affiche "1"
echo $Var; //affiche "2"
```

Il existe d'autres manières de construire des noms de variables (en particulier des noms illégaux) mais, lorsqu'un tel besoin se fait sentir, il est souvent préférable d'utiliser des tableaux (nous y reviendrons par la suite).

```php
<?php
//exemple correct mais à éviter :
${'1 */-+$%'} = 'essai'; //ce qui est entre accolades est le nom de la variable
var_dump(${'1 */-+$%'}); //ok

//alternative à éviter aussi :
$index = '1 */-+$%';
${$index} = 'essai'; //le nom de variable correspond ici au contenu de "$index"
```

```php
var_dump(${$index}); //ok
```

Contrairement à d'autres langages, il n'est pas nécessaire de déclarer une variable avant de pouvoir lui affecter une valeur. Cependant, prenez garde à ne pas utiliser la valeur d'une variable avant d'être certain de lui avoir effectivement donné une valeur car cela lancerait un avertissement.

```php
<?php
echo $x; //PHP lance un avertissement puisque $x n'existe pas encore
```

```php
<?php
$x = 3; //la première affectation sert de déclaration
echo $x;
```

Le type d'une variable PHP peut changer au cours de l'exécution d'un même script. Pour cela, il suffit de lui affecter une nouvelle valeur.

```php
<?php
$variable = 1;
var_dump($variable);

$variable = 1.1;
var_dump($variable);

$variable = 'texte';
var_dump($variable);

$variable = new stdClass();
var_dump($variable);
```

```
int(1)
float(1.1)
string(5) "texte"
object(stdClass)#1 (0) {
}
```

Les variables ne sont visibles (scope) que dans le bloc dans lequel elles sont déclarées. Une variable déclarée dans une fonction, classe ou méthode n'est pas visible en-dehors de cette fonction, classe ou méthode.

Les variables superglobales sont mises en place par PHP lors du début du traitement d'une demande par Apache.

Ces variables n'obéissent pas aux limites habituelles des variables en termes de visibilité à l'intérieur d'une fonction. Elles sont accessibles de partout, c'est pourquoi elles portent le nom de "superglobales".

Voici les superglobales :

- **$_GET** : Les valeurs provenant de l'URL ;
- **$_POST** : Les valeurs envoyées par formulaire ;
- **$_FILE** : Les fichiers envoyés par formulaire ;
- **$_SERVER** : Les valeurs mises en place par le serveur Web (elles peuvent donc changer d'une configuration à l'autre) ;
- **$_ENV** : Les variables d'environnement (système d'exploitation) ;
- **$_SESSION** : Les valeurs mises dans le magasin des sessions ;

- **$_COOKIE** : Les valeurs transmises au moyen de cookies par le navigateur ;
- **$GLOBALS** : L'ensemble des variables du script.

Du point de vue de la sécurité, aucune de ces variables ne devrait être introduite telle quelle dans un script PHP, car elles sont toutes potentiellement modifiables par l'internaute. Cela signifie que l'internaute peut pirater nos scripts par l'intermédiaire de toutes ces variables, il faut donc les filtrer et les valider à chaque utilisation. Nous reviendrons plus loin sur les méthodes permettant d'y parvenir.

Transmettre une variable en "entrée-sortie" à une fonction s'apparente à la passer par référence. En PHP5, il n'y a qu'une manière de faire :

```php
<?php
$i = 0;
increment($i);
increment($i);
increment($i);
increment($i);

echo $i; //affiche "4"


function increment(&$nb)

{   ++$nb;

}
```

Cette approche est très efficace en utilisation de mémoire, car elle évite à la fonction de recopier la valeur de ses paramètres transmis par référence. En revanche, il faut faire très attention lorsque l'on modifie ces valeurs, car les changements sont répercutés dans le scope appelant la fonction...
Nous verrons plus loin que les objets PHP5 sont systématiquement transmis par référence.

L'affichage du contenu d'une variable se fait généralement au moyen de la fonction **echo** ou bien **print**, selon le choix du programmeur. Ces deux fonctions agissent presque de la même manière. Pour leur part, les fonctions **var_export()**, **print_r()** et **var_dump()** sont utiles principalement pour le débogage des scripts.

```php
<?php
$x = 5;
echo $x; //affichage de la valeur
print $x; //affichage de la valeur
var_export($x); //affichage de la représentation PHP
print_r ($x); //affichage du contenu
var_dump($x); //affichage du type et du contenu
```

```php
<?php
if isset($variable))
  (
{   //la variable existe

}else

{   //la variable n'existe pas

}
```

```php
<?php
if (empty($variable))
{   //la variable est nulle

}
else
{   //la variable est non nulle

}
```

```php
<?php
unset($variable);
```


Contrairement à echo , les constructs isset() et empty() ne lancent aucun avertissement si la variable n'existe pas.

Utilisez des noms de variable ayant du sens. Évitez les abréviations, les initiales, *etc*

Évitez les noms du style "cas particulier", par exemple pour une couleur il faut éviter de décrire le contenu ($blue, $green...) et préférer **la description de l'utilité** de chaque variable : $row_color, $background_color... Appeler une variable en fonction de sa valeur revient à dupliquer les informations, ce qu'il faut toujours éviter de faire en programmation.

Si vous devez utiliser une syntaxe complexe pour afficher une variable, comme par exemple ce qui est dans la documentation officielle de PHP, préférez la concaténation ou l'utilisation de sprintf() :

```php
echo "Ceci fonctionne : {$arr['foo'][3]}";
```

```php
echo "Ceci fonctionne : " . $arr['foo'][3];
```

```php
echo sprintf("Ceci fonctionne : %s", $arr['foo'][3]); //préciser le type correct à la place de "%s"
```

Une constante est un nom qui permet de donner une sémantique à une valeur. Cette valeur est figée pour toute la durée de l'exécution du script PHP. Par convention, on exclut les lettres minuscules dans le nom d'une constante.

```php
<?php
define('NOM_ADMIN', 'Yogui'); //constante de type string
define('MAX_LIGNES', 5); //constante de type entier
```

L'utilisation des constantes est similaire à des variables. La différence est qu'il ne faut pas utiliser le signe dollar ($) car il s'applique uniquement aux variables. Par ailleurs, il ne faut pas utiliser "=" pour affecter une valeur à une constante, mais uniquement "define". Une fois que la valeur d'une constante est définie, elle ne peut plus être modifiée jusqu'à la fin du script.

```php
<?php
define('NOM_ADMIN', 'Yogui');
echo NOM_ADMIN; //remarquez l'absence de guillemets et de signe dollar
```

Par convention, les noms des constantes sont écrits en lettres majuscules et en chiffres. S'il faut mettre plusieurs mots dans le même nom de constante (comme ci-dessus), on démarque chaque mot par un underscore.

Les constantes sont très utiles pour les valeurs de configuration qui ne doivent pas changer durant toute l'exécution d'un script, car elles nous assurent que la valeur ne sera modifiée à aucun moment de l'exécution.

```php
1. <?php
2. define('MAX_LIGNES', 3);
3. define('MAX_LIGNES', 5);
```

```
Notice: Constant MAX_LIGNES already defined in C:\Web\online\http\tests\error.php on line 3
```

Un aspect intéressant des constantes est que, si l'on utilise une constante qui n'est pas définie, PHP envoie une alerte mais n'arrête pas le script : il utilise le nom de la supposée constante comme sa valeur. Il est évident qu'il faut éviter ce genre de situation, mais cela vaut la peine de le savoir.

Les constantes ne sont pas limitées en termes de visibilité (scope). Une fois déclarée, une constante est visible depuis toutes les fonctions et toutes les classes.

```php
<?php

$user = 'Yogui';
define('USER', 'Yogui');

function test()

{   echo $user; //lance un avertissement PHP "Undefined variable: user"
    echo USER; //ok

}

//lancemant de la démonstration
test();
```

Il existe des "constantes magiques" qui n'obéissent pas totalement aux règles des constantes. Elles existent dans tous les scripts sans qu'il soit nécessaire de les déclarer par programmation. On ne peut bien entendu pas les modifier par programmation, néanmoins leur valeur peut changer au fil de l'exécution du script. De plus, on peut les appeler indifféremment en majuscules ou en minuscules. Chacune de ces constantes contient une valeur qui change selon le contexte dans lequel elle est appelée.

- **__LINE__** : La ligne de code en cours ;
- **__FILE__** : Le nom complet du script en cours ;
- **__DIR__** : Le nom du répertoire du script en cours (depuis les versions 5.3 et 6.0 de PHP) ;
- **__FUNCTION__** : La fonction en cours ;
- **__CLASS__** : La classe en cours, similaire à **get_class($this)** ;
- **__METHOD__** : La méthode en cours ;
- **__NAMESPACE__** : L'espace de noms en cours (depuis les versions 5.3 et 6.0 de PHP).

```php
<?php
echo __LINE__; //affiche "2"
echo __LINE__; //affiche "3"
echo __LINE__; //affiche "4"
```

```php
<?php
define('MAX_LIGNES', 3);
```

```php
<?php
if (defined('MAX_LIGNES'))
{
   echo 'Constante définie';
}
else
{
   echo 'Constante non définie';
}
```

```php
<?php
print_r(get_defined_constants());
```

```php
<?php
define('LOGIN', 'Yogui');

echo LOGIN; //ok
echo constant('LOGIN'); //identique à la ligne précédente, sauf qu'on peut aussi bien mettre une variable contenan
```

Les constantes sont particulièrment utiles pour les valeurs de configuration. Par exemple, si vous avez besoin d'un nom d'utilisateur et d'un mot de passe pour vous connecter à une base de données, vous devriez utiliser des constantes afin d'éviter qu'une portion de votre script ne modifie ces valeurs par inadvertance (une collision de noms est si vite arrivée...).

```php
<?php
define('DB_LOGIN', 'Yogui');
define('DB_PASSWORD', 'pass');
```

Si, dans la suite du script, vous souhaitez utiliser le même nom de constante, PHP vous avertira par une erreur car ce nom existe déjà. Si vous utilisiez une variable, sa valeur serait écrasée sans préavis...

Un tableau est une variable contenant plusieurs valeurs. En PHP, les variables étant faiblement typées, les tableaux sont très simples à manipuler.

On accède au tableau entier en utilisant le nom de la variable, ou bien à un élément concret au moyen des crochets [et]. Ce qui se situe entre les crochets est appelé "index" ou encore "clef" de l'élément du tableau, et un même index est unique dans un tableau.

```php
<?php
$nombres = array(3, 6, 9); //ce tableau contient trois valeurs

echo $nombres[1]; //accès direct à l'élément d'index 1, c'est-à-dire le deuxième élément
print_r ($nombres); //affichage du tableau complet
```

```php
<?php
$nombres = array();
$nombres[] = 3; //ajout à la position suivante = zéro
$nombres[] = 6; //ajout à la position suivante = un
$nombres[] = 9; //ajout à la position suivante = deux

var_dump ($nombres[1]); //accès direct à l'élément en position 1
print_r ($nombres); //affichage complet
```

```
int(6)
Array

(   [0] => 3
    [1] => 6
    [2] => 9

)
```

L'indice permettant d'indexer le tableau peut être numérique ou chaîne. Si on ne spécifie rien (comme ci-dessus), ce sont des valeurs numériques successives. On peut voir ici que le premier élément d'un tableau prend l'index zéro, ce qui est tout à fait habituel dans les langages de programmation.

```php
<?php
$positions = array
    0 => 'début', (
    1 => 'milieu',
    2 => 'fin'
);

echo $positions[1]; //accès direct à l'élément en position 1
print_r ($positions); //affichage complet
```

```php
<?php
$positions = array();
$positions[0] = 'début'; //ajout à la position zéro
$positions[1] = 'milieu'; //ajout à la position un
$positions[2] = 'fin'; //ajout à la position deux

var_dump ($positions[1]); //accès direct à l'élément en position 1
print_r ($positions); //affichage complet
```

```
string(6) "milieu"
Array

(   [0] => début
    [1] => milieu
    [2] => fin

)
```

L'index d'un élément permet d'y accéder aussi bien en lecture qu'en modification ou suppression :

```php
<?php
$nombres = array();
$nombres[] = 3; //ajout à la position suivante = zéro
$nombres[] = 6; //ajout à la position suivante = un
$nombres[] = 9; //ajout à la position suivante = deux

print_r ($nombres); //affichage complet

$nombres[1] = 12; //modification de l'élément en position 1

print_r ($nombres); //affichage complet
```

```
Array
(
    [0] => 3
    [1] => 6
    [2] => 9
)
Array
(
    [0] => 3
    [1] => 12
    [2] => 9
)
```

Puisque chaque case d'un tableau est une variable à part entière, elle peut être de n'importe quel type (nombre, chaîne, ressource, tableau, objet...), ce qui nous permet d'imbriquer les tableaux si nécessaire.

Lorsqu'un tableau est indexé par des chaînes plutôt que par des nombres, on parle de **tableau associatif** :

```php
<?php
$positions = array();
$positions['zéro'] = 'début';
$positions['un'] = 'milieu';
$positions['deux'] = 'fin';

var_dump($positions['un']); //accès direct à l'élément en position (string)"un"
print_r ($positions); //affichage complet
```

```
string(6) "milieu"
Array
(
    [zéro] => début
    [un] => milieu
    [deux] => fin
)
```

Dans le cas d'un tableau associatif, on peut utiliser n'importe quelle chaîne comme index.
Il n'y a pas de restriction sur le nom des indexes comme il y en a pour le nom des variables.
On peut même utiliser le contenu d'une variable pour indexer un élément d'un tableau :

```php
<?php
$positions = array();
$positions['zéro'] = 'début';
$positions['un'] = 'milieu';
$positions['deux'] = 'fin';

$index = 'un';
echo $positions[$index]; //affiche "milieu"
```

Il existe plusieurs manières de parcourir un tableau :

```php
<?php
$nombres = array(3, 6, 9);
foreach($nombres as $nombre)

{   echo $nombre.'<br/>';

}
```

```php
<?php
$nombres = array(3, 6, 9);
foreach($nombres as $i => $nombre)

{   echo $i.' '.$nombre.'<br/>';

}
```

```php
<?php
$nombres = array(3, 6, 9);
for($i=0; $i<count($nombres); ++$i)

{   echo $i.' '.$nombres[$i].'<br/>';

}
```

```php
<?php
$nombres = array(3, 6, 9);
while list ($i, $nombre) = each($nombres))
     (
{   echo $i.' '.$nombre.'<br/>';

}
```


Le pointeur interne d'un tableau est une valeur manipulée uniquement par PHP lui-même, en arrière-plan. C'est la position à laquelle PHP s'est arrêté lorsqu'il a utilisé le tableau pour la dernière fois (dans le cas d'un parcours séquentiel).

Nous y reviendrons par la suite, notamment lors des chapitres sur les boucles (structures de contrôle) et sur les interfaces (programmation orientée objet).

Cela permet par exemple d'avoir un tableau indexé par des numéros mais dont l'ordre des éléments ne suit pas ces numéros, ou bien de réorganiser les clefs d'un tableau associatif, *etc*

.

count() : Compter le nombre d'éléments d'un tableau ;

- ***sort()** : Trier un tableau (nombreuses fonctions disponibles) ;
- **array_*()** : *cf.* la documentation (nombreuses fonctions disponibles) ;
- **list()** : Assigne plusieurs valeurs en une opération (habituellement depuis un tableau) ;
- **current()** : Retourne l'élément du tableau désigné par son pointeur interne ;
- **reset()** : Réinitialise le pointeur interne du tableau ;
- **next()** : Avance le pointeur interne puis agit comme current() ;
- **prev()** : Recule le pointeur interne puis agit comme current().
-

```php
<?php
if (<expression>)

{   <instructions>

}
```

La structure conditionnelle if/else est l'une des plus classiques des langages de programmation. Elle permet d'effectuer des opérations ou d'autres opérations selon certaines conditions.

Elle s'écrit avec le mot clef "**if**" et un paramètre qui peut être une valeur, une variable, une comparaison, bref tout type d'instruction. Si l'instruction est évaluée à TRUE, alors on rentre dans la branche ; sinon, on rentre dans la branche du "**else**" (qui est facultative).

Le code à exécuter (après la condition entre parenthèses) peut être une simple instruction ou bien un bloc d'instructions délimitées par des accolades. L'une des bonnes pratiques de programmation indique la préférence systématique pour les accolades.

```php
<?php
if (2*4 > 5)

{   echo 'strictement supérieur à 5';

else

{   echo 'inférieur ou égal à 5';

}
```

```php
<?php
if (2*4 > 5) echo 'strictement supérieur à 5';
else echo 'inférieur ou égal à 5';
```

```php
<?php
switch(<expression>)

{   case <valeur 1>:
        <instructions>
    break
        ;
    case <valeur 2>:
        <instructions>
    break
        ;
    ...

    default
        <instructions>

}
```

Lorsqu'une variable ou une instruction peut prendre une valeur parmi une liste connue, alors il est préférable d'utiliser le mot clef "**switch**" plutôt que des "if/else".

```php
<?php
switch($variable)

{   case 1:
        echo 'Choix numéro un';
    break
        ;
    case 2:
        echo 'Choix numéro deux';
    break
        ;
    case 3:
        echo 'Choix numéro trois';
    break
        ;
    default
        echo 'Choix invalide';

}
```

```php
<?php
if ($variable == 1)

{   echo 'Choix numéro un';

}else

{   if ($variable == 2)
    {
        echo 'Choix numéro deux';
    }
    else
    {
        if ($variable == 3)
        {
            echo 'Choix numéro trois';
        }
        else
        {
            echo 'Choix invalide';
        }
    }

}
```

Le nombre de **case** peut varier suivant l'utilisation, cela dépend de la situation. Le mot clef "**break**" oblige PHP à sortir du bloc switch.

Certains langages ont des restrictions sur le case, qui ne rendent le mot clef switch utilisable qu'avec des valeurs entières. PHP étant un langage à typage faible, cette restriction ne s'applique pas et chaque case peut contenir une chaîne voire même une instruction complète.

Le mot clef *break* est facultatif : s'il n'est pas précisé, le *case* suivant est évalué à TRUE jusqu'à rencontrer la fin du *switch* ou bien un *break*. Ici par exemple, le même traitement est appliqué à "BrYs", "mathieu" et "Yogui" :

```php
<?php
switch($membre)

{   case 'Marc Lussac':
        echo 'Bonjour ô grand manitou';
```

```
    break
        ;
    case 'BrYs':
    case 'mathieu':
    case 'Yogui':
        echo 'Bonjour '.$membre;
    break
        ;
    default
        echo 'Utilisateur invalide';

}
```

Le mot clef *default* est facultatif, mais toujours le mettre fait partie des bonnes pratiques. Il est le cas par défaut, c'est-à-dire si aucun autre cas n'a été validé. Dans l'exemple ci-dessus, *default* n'est exécuté que si **$membre** ne vaut aucun des quatre noms mentionnés.


L'instruction switch est parfois subtile, mais souvent très intéressante. Sa maîtrise vient avec la pratique, il faut donc faire des exercices en situation réelle.

```
<?php
$i = 4; //modifier pour tester l'effet
switch($i)

{   case 1:
        echo 'un';
    break
        ;
    case 2:
        echo 'deux';
    break
        ;
    default: //valide si $i ne vaut ni 1, ni 2, ni 3
        echo $i;
    break
        ;
    case 3:
        echo 'trois';
    break
        ;
}
```

Ici, le bon sens nous souffle que le cas *default* est mal situé : il devrait être en dernière position car c'est le dernier recours.


```
<?php
for(<initialisation> ; <continuer tant que> ; <incrémentation>)

{   <instructions>

}
```

La boucle "for" permet d'effectuer un groupe d'instructions un nombre déterminé de fois. Elle comprend trois paramètres qui sont habituellement des instructions :

```
<?php
//afficher tous les chiffres de 0 à 9
for($i=0; $i<10; ++$i)

{   echo $i;
```

```
}
//afficher toutes les lettres de l'alphabet français
for($lettre='a'; $lettre<='z', ++$lettre)

{   echo $lettre;

}
```

Voici comment cela se passe dans le 1° exemple :

1. $i est initialisé à "0" ;
2. PHP vérifie si l'instruction "$i<10" est TRUE ;
3. Si c'est TRUE on continue, sinon la boucle est terminée ;
4. PHP exécute le corps de la boucle ;
5. PHP exécute l'instruction d'incrémentation : "++$i" ;
6. Retour au 2° point.

La boucle "for" est à utiliser exclusivement lorsque l'on connaît d'avance le nombre de tours de boucle à effectuer.

Le deuxième paramètre (aka "continuer tant que", qui vaut ici "$i<10") est évalué à chaque tour de boucle. Il est donc préférable que ce soit une valeur fixe afin d'économiser le temps processeur et sous peine de nous exposer à des "boucles infinies".

```
<?php
$users = array('BrYs', 'mathieu', 'Yogui');

//perte de performances puisque count() est recalculé à chaque itération :
for($i=0; $i<count($users); ++$i)

{   echo $users[$i];

}
//ok :
$nb_users = count($users);
for($i=0; $i<$nb_users; ++$i)

{   echo $users[$i];

}
```

```
<?php
for(;;) //aucune condition de fin de boucle

{   //...

}
for($i=0; $i=-1; ++$i) //$i n'atteindra jamais -1

{   //...

}
```

```
<?php
while(<continuer tant que>)

{   <instructions>
```

```
}
```

Une boucle "while" est prévue pour effectuer un groupe d'instructions un nombre indéfini de fois. La condition de fin de boucle dépend souvent du résultat d'une comparaison de valeurs. Un exemple typique d'utilisation est la lecture d'un flux, par exemple un fichier :

```php
<?php
$file = fopen('file.ext', 'r'); //ouverture d'un fichier en lecture
while(!feof($file)) //tant que ce n'est pas la fin du fichier (End Of File)

{   echo fread($file, 8192); //lecture d'une ligne

}
fclose ($file); //fermeture du descripteur de fichier
```

Voici comment cela se passe dans l'exemple :

1. $file est initialisé à l'état du fichier (EOF si le fichier est vide, une valeur non nulle sinon) ;
2. PHP vérifie la condition, donc ici si nous ne sommes pas à la fin du fichier ;
3. Si c'est TRUE on continue, sinon la boucle est terminée ;
4. PHP exécute le corps de la boucle ;
5. Retour au 2° point.

Une autre utilisation classique de la boucle "while" est de mettre une ligne d'affectation comme condition :

```php
//exécution d'une requête définie plus haut dans le script
$db_result = mysql_query($sql);
while($row = mysql_fetch_assoc($db_result)) //tant qu'il y a un résultat

{   print_r ($row);

}
```

Dans ce cas, la condition du while est la valeur de l'expression "$line = fread($file, 8192)", c'est-à-dire le résultat de l'opération, soit la valeur de **$line** après affectation. Cela peut s'exprimer par la question "est-ce que l'affectation a pu se faire ?".

La boucle "while" est à utiliser exclusivement lorsque l'on ne connaît pas d'avance le nombre de tours de boucle à effectuer. Cela restreint beaucoup ses cas d'utilisation, car PHP dispose de très nombreux moyens de compter le nombre d'éléments d'un tableau, de lire des fichiers, etc. sans obliger à l'utilisation de "while".

```php
<?php
while TRUE
     (    )
{   //...

}
```

```php
<?php
do

{   <instructions>
} while(<continuer tant que>);
```

Cette boucle fonctionne sur un principe similaire au "while", mais le premier tour de boucle est effectué avant toute chose.

Dans d'autres langages, une boucle "do...while" est utilisée par exemple pour afficher un menu à l'écran et attendre la réponse de l'utilisateur : on sait d'avance que le menu est affiché au moins une fois, mais on ne sait pas d'avance quand l'utilisateur souhaitera quitter l'application. Cet exemple est malheureusement inefficace en PHP, puisqu'un script se termine sans que l'utilisateur puisse interagir.

```php
<?php
do

{   menu_display(); //définir cette fonction selon le besoin
    $choice = get_user_choice(); //définir cette fonction selon le besoin
    switch($choice)
    {
        //instructions selon la valeur de $choice
    }
} while($choice != CHOICE_LEAVE); //définir cette constante selon le besoin
```


Cette instruction n'est pas vraiment une boucle mais on l'utilise généralement conjointement à un type de boucle. Elle est prévue spécialement pour parcourir les tableaux :

```php
<?php
$membres = array('BrYs', 'mathieu', 'Yogui');
print_r  each($membres));
print_r (each($membres));
print_r (each($membres));
       (
```

Chaque utilisation de la fonction fait avancer le pointeur interne du tableau, c'est-à-dire la position à laquelle PHP s'est arrêté la dernière fois qu'il a consulté le tableau.

Afin de simplifier son utilisation, on la couple souvent avec une boucle "while" :

```php
<?php
$membres = array('BrYs', 'mathieu', 'Yogui');
while list ($i, $membre) = each($membres))
     (
{   echo $membre;

}
```

À mon avis, cette utilisation n'est pas optimale à cause de la sémantique du langage. L'exécution est rapide, mais le code n'est pas très clair. Le construct foreach est souvent plus intéressant de ce point de vue.


```php
<?php
foreach(<tableau> as <element>)

{   <instructions>

}
```

```php
<?php
foreach(<tableau> as <clef> => <element>)

{   <instructions>

}
```

Le mot clef "each" étant assez complexe à utiliser, PHP propose une boucle très pratique : foreach.

```php
<?php
$membres = array('BrYs', 'mathieu', 'Yogui');

//parcours du tableau :
foreach($membres as $membre)

{   echo $membre; //affiche tour à tour "BrYs" puis "mathieu" puis "Yogui"

}
//parcours avec clefs :
foreach($membres as $i => $membre)

{   //affiche tour à tour "0 BrYs" puis "1 mathieu" puis "2 Yogui"
    echo $i . ' ' . $membre;

}
//alternative :
foreach($membres as $i => $membre)

{   //affiche tour à tour "0 BrYs" puis "1 mathieu" puis "2 Yogui"
    echo $i . ' ' . $membres[$i];

}
```

```php
<?php
function <nom de la fonction> (<noms des paramètres>)

{   <instructions>
    return <valeur>; (optionnel)

}
```

Une fonction permet de réutiliser facilement du code PHP. Par exemple si vous devez effectuer la même suite d'opérations à plusieurs endroits de votre code, il est sans doute judicieux de la mettre dans une fonction. Cela vous permet de ne pas dupliquer le code source et d'en faciliter la maintenance.

Une fonction se déclare en utilisant le mot clef "function" suivi du nom de la fonction (mêmes règles que pour les noms de variables), d'un couple de parenthèses et d'un couple d'accolades. Les parenthèses contiennent la liste des paramètres (de zéro à l'infini) et les accolades contiennent les actions de la fonction.

En PHP, le terme "procédure" est très peu utilisé car on part du principe que toutes les fonctions devraient retourner une valeur : au minimum, si la fonction a réussi ou non (booléen). Toutefois, il ne s'agit ici que d'une convention : d'un point de vue syntaxique, il est possible d'avoir une fonction qui n'utilise pas le mot clef "return" (mais ce serait probablement une mauvaise idée).

La valeur de retour d'une fonction est définie par le mot clef "return". Dès que ce mot clef est exécuté, PHP sort de la fonction.

```php
<?php
function ma_fonction()

{   echo 'essai 1';
    return TRUE; //PHP sort de la fonction en renvoyant la valeur "vrai"
```

```
    echo 'essai 2'; //cette instruction ne sera jamais exécutée

}
```

Voyons un exemple d'application :

```
<?php
var_dump(3 * 100 / 8);
var_dump(2 * 100 / 72);
var_dump(5 * 100 / 50);
```

```
<?php
function pourcent($x, $y)

{   return $x * 100 / $y;

}
var_dump(pourcent(3, 8));
var_dump(pourcent(2, 72));
var_dump(pourcent(5, 50));
```

```
float(37.5)
float(2.7777777777778)
int(10)
```

La logique de calcul est ici centralisée dans la fonction **pourcent()**, ce qui facilite la maintenance. On peut par exemple utiliser des fonctions pour la gestion des grands nombres, sans devoir retoucher le code de toute l'application :

```
<?php
var_dump(bcdiv(bcmul(3, 100), 8, 2));
var_dump(bcdiv(bcmul(2, 100), 72, 2));
var_dump(bcdiv(bcmul(5, 100), 50, 2));
```

```
<?php
function pourcent($x, $y)

{   return bcdiv(bcmul($x, 100), $y, 2);

}
var_dump(pourcent(3, 8));
var_dump(pourcent(2, 72));
var_dump(pourcent(5, 50));
```

```
string(5) "37.50"
string(4) "2.77"
string(5) "10.00"
```

Notez que seul le corps de la fonction a été modifié dans le code mis en facteur, mais que tout le programme est affecté.

Il ne faut pas faire des fonctions à tort et à travers, il est par exemple préférable que la fonction retourne une valeur simple plutôt qu'une valeur complexe, et il vaut mieux éviter d'inclure du code HTML dans le retour d'une fonction.

La visibilité (ou "scope") d'une variable dépend de sa première utilisation. Une variable n'est visible que dans la fonction dans laquelle elle a été définie :

```php
<?php
$x = 5;
echo $x; //affiche "5"
test(); //affiche une erreur
echo $x; //affiche "5"


function test()

{   echo $x; //illégal car $x n'est pas défini dans la fonction test()

}
```

```php
<?php
$x = 5;
echo $x; //affiche "5"
test();
echo $x; //affiche "5" car ce $x-ci n'est plus celui de la fonction test()


function test()

{   $x = 7;
    echo $x; //affiche "7" car $x appartient à la fonction test()

}
```

```php
<?php
$x = 5;
echo $x; //affiche "5"
$objet_test = new Test(7);
echo $objet_test->x; //affiche "7"


class Test

{   public $x;
    public function __construct($valeur)
    {
        $this->x = $valeur;
    }

}
```

Il existe cependant un mot clef "global" permettant d'outrepasser cette restriction :

```php
<?php<?php
$x = 5;
echo $x; //affiche "5"
test(); //affiche "5"
echo $x; //affiche "5"


function test()

{   global $x;
    echo $x; //reprend le "$x" du scope global

}
```

```php
<?php
$x = 5;
echo $x; //affiche "5"
test(); //affiche "7"
```

```php
echo $x; //affiche "7"


function test()

{   global $x;
    $x = 7;
    echo $x; //modifie "$x" dans le scope global

}
```

Fonctions :

- **function_exists()** : Est-ce que la fonction existe ?
- **get_defined_functions()** : Liste des fonctions définies ;
- **func_num_args()** : Nombre de paramètres transmis à la fonction courante ;
- **func_get_arg()** : Un paramètre de la fonction courante ;
- **func_get_args()** : Tous les paramètres de la fonction courante ;
- **create_function()** : Créer une fonction utilisateur à partir du code PHP fourni en commentaires.

L'utilisation de create_function() doit se faire uniquement après mûre réflexion (question de sécurité)...

Constantes magiques :

- **__FUNCTION__** : Donne le nom de la fonction en cours.

La programmation orientée objet (**POO**) a fait son apparition dans la version 3 de PHP. C'était alors simplement un moyen d'autoriser la syntaxe OO (par opposition au procédural), mais pas réellement un moyen de programmer efficacement avec des objets.

PHP4 a continué dans la lancée, proposant de nouveaux mots clefs, mais toujours sans proposer une syntaxe proche de langages ayant une plus grande maturité comme C++ ou Java. C'est ce qui a facilité le passage des applications et des hébergements de PHP3 à PHP4, et c'est ce qui retarde la mise en place massive de PHP5 à travers le Web.

PHP5, en revanche, introduit de véritables concepts OO : le constructeur est plus clairement identifié, le destructeur fait son apparition, les objets sont tous pris en charge comme des références, de nouveaux mots clefs font leur apparition (**public**, **protected** et **private**) ainsi que des interfaces et des classes abstraites...

Une **classe** est une représentation abstraite d'un **objet**. C'est la somme des propriétés de l'objet et des traitements dont il est capable. Une Chaise, un Livre, un Humain, une Rivière sont autant d'exemples possibles de classes. Une classe peut généralement être rendu concrète au moyen d'une **instance de classe**, que l'on appelle **objet** : on parle d'*instancier* la classe. Un objet est donc un exemple concret de la définition abstraite qu'est la classe.

On parle d'**héritage** lorsque l'on définit une hiérarchie de classes. On peut par exemple définir la classe Chien mais elle nous semble trop abstraite : si j'instancie Chien par un objet $médor, je ne sais pas de quelle race est mon chien. Je pourrais avoir une propriété "race" dans la classe Chien, mais la solution de l'héritage me donne plus de souplesse : elleme permet de redéfinir des propriétés et des méthodes de la classe Chien, tout en héritant de celles que je ne modifie pas.

Ces mots permettent de déclarer des classes en PHP :

- **class** : Déclaration de classe ;
- **const** : Déclaration de constante de classe ;
- **function** : Déclaration d'une méthode ;
- **public protected private** : Accès (par défaut "public" si aucun accès n'est explicitement défini) ;
- **new** : Création d'objet ;
- **self** : Résolution de portée (la classe elle-même) ;
- **parent** : Résolution de portée (la classe "parent") ;
- **static** : Résolution de portée (appel statique) disponible depuis PHP 5.3 et 6.0 ;
- **extends** : Héritage de classe ;
- **implements** : Implémentation d'une interface (dont il faut redéclarer toutes les méthodes).

Les mots clefs "self" et "parent" sont utiles pour accéder à une propriété ou méthode (statique ou non) de la classe elle-même ou de son parent.

Le mot clef "static" a la même utilité mais il résoud la portée au moment de l'exécution du script (*cf.* plus loin).

Les **méthodes magiques** sont des méthodes qui, si elles sont déclarées dans une classe, ont une fonction déjà prévue par le langage.

Méthodes magiques

- **__construct()** : Constructeur de la classe ;
- **__destruct()** : Destructeur de la classe ;
- **__set()** : Déclenchée lors de l'accès en écriture à une propriété de l'objet ;
- **__get()** : Déclenchée lors de l'accès en lecture à une propriété de l'objet ;
- **__call()** : Déclenchée lors de l'appel d'une méthode inexistante de la classe (appel non statique) ;
- **__callstatic()** : Déclenchée lors de l'appel d'une méthode inexistante de la classe (appel statique) : disponible depuis PHP 5.3 et 6.0 ;
- **__isset()** : Déclenchée si on applique **isset()** à une propriété de l'objet ;
- **__unset()** : Déclenchée si on applique **unset()** à une propriété de l'objet ;
- **__sleep()** : Exécutée si la fonction **serialize()** est appliquée à l'objet ;
- **__wakeup()** : Exécutée si la fonction **unserialize()** est appliquée à l'objet ;
- **__toString()** : Appelée lorsque l'on essaie d'afficher directement l'objet : **echo $object;** ;
- **__set_state()** : Méthode statique lancée lorsque l'on applique la fonction **var_export()** à l'objet ;
- **__clone()** : Appelé lorsque l'on essaie de cloner l'objet ;
- **__autoload()** : Cette fonction n'est pas une méthode, elle est déclarée dans le scope global et permet d'automatiser les "include/require" de classes PHP.

Enfin, la variable **$this** utilisée à l'intérieur d'une classe, vaut toujours une référence vers l'objet lui-même. Pour plus d'informations sur les références, *cf*. plus loin à la section "Références et clônage".

Je ne prétends pas refaire un cours complet sur la POO, car il y a d'excellents cours sur le Net et dans de nombreux livres. Je vais toutefois survoler le sujet afin de rappeler les aspects en rapport avec PHP.

La POO représente la programmation par objets. Un objet est la concrétisation d'une classe, une classe étant un ensemble générique (par opposition à concret) de propriétés et de fonctionnalités. On parle d'**instancier** une **classe** pour en faire un **objet**

.

Une classe s'écrit au moyen du mot "**class**" suivi du nom de la classe et d'accolades (pas de point virgule). Les conventions d'écriture de code recommandent d'écrire ce mot comme un nom propre, à savoir avec une seule majuscule. Certaines recommandations (PEAR, Zend Framework *etc*.) préconisent d'utiliser l'underscore pour identifier l'héritage :

```php
<?php
class Animal{}
class Animal_Chien extends Animal{}
class Animal_Chien_Caniche extends Animal_Chien{}
class Animal_Chien_Labrador extends Animal_Chien{}
```

La création d'une instance de classe se fait au moyen du mot clef "new". L'accès aux propriétés et méthodes de l'objet se fait par la flèche "->", et l'accès statique par l'opérateur de résolution de portée "::". Le point n'est pas utilisé dans la POO de PHP.

Les classes ont des propriétés et méthodes privées, c'est-à-dire internes et qui ne concernent pas l'extérieur. Ces propriétés sont déclarées en tant que "**private**".

```php
<?php
class Caniche
{
    private $nbPattes;

    public function __construct()
    {
        $this->nbPattes = 4;
    }
}

$froufrou = new Caniche();
echo $froufrou->nbPattes; //illégal depuis l'extérieur de la classe
```

```
Fatal error: Cannot access private property Caniche::$nbPattes
    in C:\Web\online\http\tests\error.php on line 8
```

```php
<?php
class Caniche

{   private $nbPattes;

    public function __construct()
    {
        $this->nbPattes = 4;
    }

    public function nbPattes() //voici ce que l'on appelle un "getter"
    {
        return $this->nbPattes; //ok depuis l'intérieur de la classe
    }

}
$froufrou = new Caniche();
echo $froufrou->nbPattes(); //affiche "4"
```

Les propriétés ou méthodes "**protected**" concernent les objets de la même classe ainsi que ses dérivées, mais pas ceux des classes étrangères.

```php
<?php
class Chien

{   protected $presenceQueue;

    public function __construct()
    {
        $this->presenceQueue = TRUE
    }                            ;

}
class Chien_Labrador extends Chien

{   public function aUneQueue()
    {
        if ($this->presenceQueue) //ok depuis une classe fille
        { (
            return 'oui';
        }
        else
        {
            return 'non';
        }
    }

}
```

```php
$médor = new Chien_Labrador();
echo $médor->aUneQueue(); //affiche "oui"
```

Les propriétés ou méthodes " **public**" sont visibles et manipulables par tous les objets, même s'ils sont d'autres classes.

```php
<?php
class Caniche

{   public $âge = 2;

    public function __construct()
    {
        $this->âge = 2;
    }

}
$froufrou = new Caniche();
echo $froufrou->âge; //affiche "2"
```

Les mots clefs "**parent**" et "**self**" combinés à l'opérateur " **::**" résolvent respectivement la classe dont ils héritent et leur propre classe :

```php
<?php
class Chien

{   protected function aboyer()
    {
        return 'Je suis un chien';
    }

}
class Chien_Labrador extends Chien

{   protected function aboyer()
    {
        return 'Je suis un labrador';
    }

    public function identifierParent()
    {
        return parent::aboyer();
    }

    public function identifierSelf()
    {
        return self::aboyer();
    }

}
$médor = new Chien_Labrador();
echo $médor->identifierParent().'<br/>';
echo $médor->identifierSelf().'<br/>';
```

```
Je suis un chien
Je suis un labrador
```

Les variables et méthodes " **static**" sont communes à l'ensemble des objets d'une même classe au moyen de l'opérateur "::". On parle alors de propriétés ou de méthodes "de classe" puisqu'elles n'appartiennent pas à un objet en particulier.

```php
<?php
class Caniche

{   public static $caniches = 0;
    public function __construct()
    {
        ++self::$caniches;
    }

}
$froufrou = new Caniche();
$froufrette = new Caniche();

echo Caniche::$caniches; //affiche "2"
```

L'accès "public/protected/private" s'applique de la même manière que pour les accès non statiques :

```php
<?php
class Caniche
{
    protected static $caniches = 0;
    public function __construct()
    {
        ++self::$caniches;
    }
}

$froufrou = new Caniche();
$froufrette = new Caniche();

echo Caniche::$caniches;
```

```
Fatal error: Cannot access protected property Caniche::$caniches
    in C:\Web\online\http\tests\error.php on line 14
```

Interface

Une **interface** est un ensemble de méthodes que les classes doivent définir si elles veulent l'implémenter.

```php
<?php
interface Joueur
{
    // une classe qui veut implémenter l'interface Joueur
    // doit définir la méthode jouer()
    public function jouer();
}

class Labrador implements Joueur{} //illégal car pas de définition de jouer()

$médor = new Labrador();
$médor->jouer();
```

```
Fatal error: Class Labrador contains 1 abstract method
    and must therefore be declared abstract or implement the remaining methods
    (Joueur::jouer) in C:\Web\online\http\tests\error.php on line 7
```

```php
<?php
interface Joueur

{   // une classe qui veut implémenter l'interface Joueur
    // doit définir la méthode jouer()
    public function jouer();
```

```
}
class Labrador implements Joueur

{   public function jouer()
    {
        echo 'Ouah!';
    }

}
$médor = new Labrador();
$médor->jouer();
```

De nombreuses interfaces intéressantes sont définies dans la **Standard PHP Library** (SPL).

```
<pre>
<?php
print_r(get_declared_interfaces());
```

Depuis PHP 5, les objets sont tous des *références*. Ainsi, copier un objet vers un autre au moyen de l'opérateur "=" ne duplique pas l'objet, au contraire il créé une deuxième référence vers le même objet.

Lorsque l'on créé un objet avec l'opérateur **new**, PHP créé l'objet en mémoire et référence son emplacement en mémoire dans la variable :

```
$object = new stdClass();
```

La variable **$object** contient maintenant une référence vers un objet de la classe **stdClass**, mais elle ne contient pas l'objet lui-même. Pour détruire cet objet en mémoire, il faut détruire toutes les références vers cette instance de la classe.

```
$object = new stdClass();
unset($object);
//toutes les références sont détruites, l'objet n'existe donc plus en mémoire

$object_1 = new stdClass();
$object_2 = $object_1;
unset($object_1); //il reste une référence, l'objet persiste donc en mémoire
unset($object_2); //l'objet n'est plus référencé par aucune variable, il est donc détruit
```

Pour nous en convaincre, essayons le script suivant :

```
<?php
class Test

{   public function __destruct()
    {
        echo 'Objet détruit<br/>';
    }

}
$obj_1 = new Test();
$obj_2 = $obj_1;
$obj_3 = $obj_2;
```

```php
echo 'Marqueur n° 1<br/>';
unset($obj_1);

echo 'Marqueur n° 2<br/>';
unset($obj_2);

echo 'Marqueur n° 3<br/>';
```

```
Marqueur n° 1
Marqueur n° 2
Marqueur n° 3
Objet détruit
```

La phrase "Objet détruit" apparaît uniquement lorsque la fin du script cause la destruction de la 3° et dernière référence (appel automatique du destructeur).

Voyons maintenant avec des clônes de l'objet initial :

```php
<?php
class Test

{   public function __destruct()
    {
        echo 'Objet détruit<br/>';
    }

}
$obj_1 = new Test();
$obj_2 = clone $obj_1;
$obj_3 = clone $obj_2;

echo 'Marqueur n° 1<br/>';
unset($obj_1);

echo 'Marqueur n° 2<br/>';
unset($obj_2);

echo 'Marqueur n° 3<br/>';
```

```
Marqueur n° 1
Objet détruit
Marqueur n° 2
Objet détruit
Marqueur n° 3
Objet détruit
```

Ici, il y a bien une seule référence de chaque objet.

Une innovation de PHP 5.3 et 6.0 est ce que l'on appelle les "Late Static Bindings".

Les LSB sont nouvelle faculté de PHP de résoudre les appels statiques plus tard dans la chronologie des évènements. C'est une question de fonctionnement interne de PHP, et le plus important pour nous est de savoir que les deux exemples suivants agissent différemment :

```php
<?php
class Chien

{   protected function aboyer()
    {
        return 'Je suis un chien';
```

```
    }

    public function identifier()
    {
        return self::aboyer(); //appel à la classe elle-même
    }

}
class Chien_Labrador extends Chien

{   protected function aboyer()
    {
        return 'Je suis un labrador';
    }

}
$médor = new Chien();
$félix = new Chien_Labrador();

echo $médor->identifier().'<br/>';
echo $félix->identifier().'<br/>';
```

```
Je suis un chien
Je suis un chien
```

Ici, la méthode **Chien::identifier()** est bien transmise par héritage à la classe **Chien_Labrador** mais elle n'est pourtant pas appelée. Cela a été perçu comme un problème par de nombreuses personnes, et a donné lieu aux *Late Static Bindings*. De nombreux articles ont été écrits sur le sujet, si vous souhaitez en savoir davantage.

Les Late Static Bindings permettent d'utiliser un nouveau mot clef " **static::**" pour résoudre correctement la portée statique :

```
<?php
class Chien

{   protected function aboyer()
    {
        return 'Je suis un chien';
    }

    public function identifier()
    {
        return static::aboyer(); //appel statique
    }

}
class Chien_Labrador extends Chien

{   protected function aboyer()
    {
        return 'Je suis un labrador';
    }

}
$médor = new Chien();
$félix = new Chien_Labrador();

echo $médor->identifier().'<br/>';
echo $félix->identifier().'<br/>';
```

```
Je suis un chien
Je suis un labrador
```

Dans cet exemple, "**static::**" permet de négocier l'appel de la méthode **aboyer()** au moment de l'exécution, et ainsi d'utiliser la méthode **Chien_Labrador::aboyer()**... Au moment de la compilation du script, on ne sait pas encore quelle classe fournira la méthode : c'est à l'exécution que tout devient clair.

Une application particulièrement intéressante des LSB est dans le cadre de projets ORM (bases de données) :

```php
<?php
class Table

{   static function getByPk($id)
    {
        return 'SELECT * FROM '.get_called_class().' WHERE id = '.(int)$id;
    }

}
class Album extends Table{}
class Artist extends Table{}

echo Album::getByPk(3);
echo Artist::getByPk(6);
```

```sql
SELECT * FROM Album WHERE id = 3
SELECT * FROM Artist WHERE id = 6
```

La fonction **get_called_class()** s'apparente à la constante magique **__CLASS__**, à la différence qu'elle retourne la classe appelée par le développeur plutôt que la classe actuelle dans le code.

Les late static bindings ne sont disponibles qu'à partir de PHP 5.3, qui est actuellement en cours de développement. Par conséquent, rien de ce qui a ici trait aux LSB n'est final (jusqu'à la sortie effective de PHP 5.3). Des mots clefs peuvent notamment apparaître, disparaître ou être renommés.

Il existe déjà de nombreux articles sur ce sujet, je ne vais donc pas m'étendre. Exemple : **Exceptions et PHP5**, par Guillaume Affringue

Les exceptions sont un moyen de gérer les situations marginales mais dont nous savons qu'elles peuvent survenir. Un exemple est l'indisponibilité du serveur pendant la connexion à une base de données : ce n'est pas une situation normale, néanmoins elle est facilement prévisible et identifiable. Un modèle Orienté Objet permet de repérer ces situations et de les gérer de manière personnalisée.

```
try
```

```php
{   throw new Exception('incident');

}catch Exception $e)
     (
{   echo $e->getMessage(); //affiche "incident"

}
```

Lorsqu'une exception est lancée depuis un bloc "try", le bloc "catch" permet de l'attraper et de traiter l'incident. Dans l'exemple ci-dessus, nous avons lancé nous-mêmes volontairement une exception.

PHP permet d'utiliser plusieurs blocs "catch" à la suite, mais ne dispose pas (encore ?) d'instruction "finally" comme dans d'autres langages.

Fonctions :

- **class_parents()** : Retourne un tableau de la classe parent et de tous ses parents ;
- **class_implements()** : Retourne un tableau de toutes les interfaces implémentées par la classe et par tous ses parents ;
- **get_class()** : Retourne la classe de l'objet passé en paramètre ;
- **get_called_class()** : À utiliser dans une classe, retourne la classe appelée explicitement dans le code PHP et non au sein de la classe ;
- **class_exists()** : Vérifie qu'une classe a été définie ;
- **get_class()** : Retourne la classe d'un objet ;
- **get_declared_classes()** : Liste des classes définies ;
- **get_class_methods()** : Liste des méthodes d'une classe ;
- **get_class_vars()** : Liste des propriétés d'une classe.

Constantes magiques :

- **__CLASS__** : Donne le nom de la classe en cours ;
- **__METHOD__** : Donne le nom de la méthode en cours.

get_class($this) et __CLASS__ ont le même effet.

Les espaces de noms, aka**namespace**s, sont un moyen de résoudre les collisions de noms de constantes, fonctions et classes.

Par exemple si j'ai besoin d'une classe pour filtrer mes variables, je peux avoir envie de créer une classe nommée "Filter". Or, ce nom est sans doute déjà utilisé par PHP ou par l'une des bibliothèques incluses dans mes scripts. La solution classique est de préfixer le nom de la classe de manière à le rendre unique : "DVP_Filter".

Ou dans un contexte MVC, j'ai souvent des modèles et des contrôleurs pour le même concept. Selon le framework utilisé, la nomenclature change mais le nom de classe est généralement préfixé lui aussi.

PHP 5.3 introduit le concept des namespaces dans PHP. Cela nous permet de définir des noms de classes, fonctions *etc*. au sein d'un espace de noms, par exemple le namespace "DVP" peut contenir la classe "Filter" alors que l'espace de noms global contient également une classe "Filter". Nous pouvons ainsi avoir autant de classes "Filter" que nous pouvons imaginer de noms de namespaces.

Les espaces de noms ne sont disponibles qu'à partir de PHP 5.3, qui est actuellement en cours de développement. Par conséquent, rien de ce qui a ici trait aux espaces de noms n'est final (jusqu'à la sortie effective de PHP 5.3). Des mots clefs peuvent notamment apparaître, disparaître ou être renommés.

```
<?php
namespace <Nom>;
```

```
...
```

```
<?php
use <Nom> as <Alias>;
$object = new Models::Member(); //avec le nom complet
$object = new M::Member(); //avec l'alias
```

Les espaces de noms composés peuvent être importés tels quels ou avec un alias :

```
<?php
namespace Cours::Models;
...
```

```
<?php
use Cours::Models;
use Cours::Models as M;
...
```

En revanche, les noms qui ne sont pas composés doivent être importés avec un alias :

```
<?php
namespace Models;
...
```

```
<?php
use Models as M;
...
```

Erreur si on ne définit pas d'alias :

```
<?php
use Models;
...
```

```
Warning: The use statement with non-compound name 'Models' has
    no effect in C:\Web\online\http\cours-php\namespaces\index.php on line 6
```

Bien entendu, il est possible d'imbriquer les espaces de noms :

```
<?php
namespace Offline::Sites::Application::Models;
```

```
<?php
use Offline::Sites::Application::Models as M;
$object = new M::Member();
```

Si vous êtes dans un espace de noms (c'est-à-dire dans un script contenant par exemple "**namespace Cours;**"), vous pouvez vous abstenir d'utiliser le préfixe "Cours::" devant les symboles de cet espace de noms. En effet, PHP suppose que vous utilisez des symboles du même espace de noms, ou bien un symbole du langage. Si vous voulez utiliser un symbole du scope global (mais pas interne à PHP), préfixez le nom du symbole par "::". Si vous utilisez un symbole d'un autre espace de noms, il faut bien entendu préfixer votre symbole de l'espace de noms complet.

Si vous êtes hors d'un espace de noms, il faut systématiquement préfixer soit de l'alias, soit du nom complet.


N'avez-vous jamais rêvé d'utiliser un nom de fonction déjà réservé par le langage ? Les espaces de noms vous permettent d'utiliser les noms déjà réservés pour des constantes, fonctions ou classes internes de PHP.


Il était initialement prévu d'utiliser le mot clef "import" comme en Java, mais cela a finalement été changé pour le mot clef "use". Voir les archives de php.internals pour plus d'informations.

Reprenons un contexte MVC assez simple pour illustrer une utilité possible des namespaces.

```
<Directory "C:/Web/online/http/cours-php/namespaces">
    AllowOverride None
    php_value include_path ".;C:/Web/offline/sites/cours-php/namespaces"
    SetEnv HTTP_ROOT /cours-php/namespaces/
    RewriteEngine on
    RewriteCond %{REQUEST_URI} !\.(js|css|jpg|png|gif)$
    RewriteRule .* index.php
</Directory>
```

```
<?php
namespace Cours::Controllers;
class Member

{   public static function whoAmI()
    {
        return 'Je suis un Contrôleur';
    }

}
```

```
<?php
namespace Cours::Models;
class Member

{   public static function whoAmI()
    {
        return 'Je suis un Modèle';
    }

}
```

```
<?php
namespace Cours::Views;
class Member

{   public static function whoAmI()
    {
        return 'Je suis une Vue';
    }

}
```

```
<?php
require 'models/member.php';
require 'views/member.php';
require 'controllers/member.php';
```

```php
header('Content-Type: text/html; charset=utf-8');
echo Cours::Models::Member::whoAmI().'<br/>';
echo Cours::Views::Member::whoAmI().'<br/>';
echo Cours::Controllers::Member::whoAmI();
```

```php
<?php
require 'models/member.php';
require 'views/member.php';
require 'controllers/member.php';

use Cours::Models;
use Cours::Views;
use Cours::Controllers;

header('Content-Type: text/html; charset=utf-8');
echo Models::Member::whoAmI().'<br/>';
echo Views::Member::whoAmI().'<br/>';
echo Controllers::Member::whoAmI();
```

```php
<?php
require 'models/member.php';
require 'views/member.php';
require 'controllers/member.php';

use Cours::Models as M;
use Cours::Views as V;
use Cours::Controllers as C;

header('Content-Type: text/html; charset=utf-8');
echo M::Member::whoAmI().'<br/>';
echo V::Member::whoAmI().'<br/>';
echo C::Member::whoAmI();
```

```
Je suis un Modèle
Je suis une Vue
Je suis un Contrôleur
```

Le comportement de PHP est dicté par sa configuration, établie dans le fichier **php.ini**. Ce fichier standard de configuration est habituellement placé dans le même répertoire que PHP, et nous pouvons le modifier à l'aide de n'importe quel éditeur de texte.

Le fichier php.ini est donc utile à la configuration de PHP, mais aussi de toutes ses extensions. En effet, chacune des extensions de PHP peut avoir sa propre section du *php.ini* et ses propres directives.

Nous allons voir ici les directives les plus courantes du fichier *php.ini*, et la valeur recommandée dans chaque situation (*ie*. **développement** ou **production**).

Les directives peuvent être de plusieurs types :

- **booléen** : Prend les valeurs "On/Off" ou bien "1/0" (les majuscules n'ont pas d'influence) ;
- **numérique** : Le temps se compte en secondes, et certaines valeurs acceptent des unités (K, M ou G) ;
- **chaîne** : Souvent entre guillemets.

De très nombreuses directives de ce fichier peuvent être modifiées pour la durée d'une requête par le fichier httpd.conf, par un fichier .htaccess ou au moment de l'exécution du script par la fonction ini_set()

Ce comportement est attribué dans le code source de PHP par l'une des constantes suivantes :

- **PHP_INI_USER** : Valeurs définies dans un script PHP ou dans le registre de Windows (aucune directive du core n'utilise PHP_INI_USER) ;
- **PHP_INI_PERDIR** : Valeurs définies dans *php.ini*, *httpd.conf* ou *.htaccess* ;
- **PHP_INI_SYSTEM** : Valeurs définies dans *php.ini* ou *http.conf* ;
- **PHP_INI_ALL** : Valeurs définies dans *php.ini*, *httpd.conf*, *.htaccess*, le registre de Windows ou un script PHP.

La documentation officielle se trouve ici : http://php.net/ini.core

Active ou désactive le tag "<?" (par opposition à "<?php") pour ouvrir un bloc PHP dans le script.

Activer cette directive est utile pour conserver la compatibilité avec d'anciens scripts (notamment de l'époque de PHP3), mais une bonne pratique est plutôt de réécrire les scripts de cette époque à cause de leur ancienneté. Les pratiques de sécurité ont changé depuis l'époque de PHP3. Ce tag apporte l'inconvénient de générer des erreurs de syntaxe lorsque l'on veut servir des documents XML sur le même serveur, puisque c'est le même type d'ouverture de balise : "<?xml", reconnu par php.ini comme une ouverture de bloc PHP, ce qui est bien sûr incorrect.

Désactiver cette directive est considéré comme une bonne pratique. Il en va de même pour **asp_tags**, qui devrait être réservé au langage ASP.

Configuration recommandée :

- **Développement** : Off ;

•**Production** : Off.

PHP_INI_PERDIR

Cette directive est habituellement désactivée, car la majorité des applications n'en ont pas besoin. De plus, celles qui utilisent ce type de buffer ont généralement un besoin très particulier qu'il est préférable de manipuler dans le script lui-même par la fonction **ob_start()**

Configuration recommandée :

- **Développement** : Off ;
- **Production** : Off.

PHP_INI_PERDIR

Le Safe Mode a beaucoup fait parler de lui, à tel point que le PHP Group prévoit de totalement le supprimer de la configuration.

Le problème de cette directive est que son nom promet un niveau de sécurité qu'elle ne peut fournir. Elle ne rend pas l'exécution de scripts "sûre", en fait c'est seulement un ensemble prédéfini de directives de configuration.

Il est préférable de ne pas utiliser cette option, et de sécuriser les scripts pendant leur développement.

Les options **safe_mode_*** ne fonctionnent pas si **safe_mode** est désactivé.

Configuration recommandée :

- **Développement** : Off ;
- **Production** : Off.

PHP_INI_SYSTEM

disable_functions permet de désactiver certaines fonctions dans l'ensemble des scripts PHP. De nombreux hébergeurs mutualisés désactivent par exemple "exec" et "dl", pour ne citer que deux exemples.

disable_classes fonctionne sur le même principe mais pour des classes.

Uniquement dans le php.ini

Bien que 30 secondes soit un temps extrêmement long pour un script PHP (dont le temps moyen d'exécution ne dépasse pas 1 seconde), l'option **max_execution_time** est souvent laissée à sa valeur par défaut de 30 secondes. Cela n'influe pas véritablement sur les performances des scripts, mais réduire cette valeur permet de savoir plus

rapidement si un script tourne en rond. La très large majorité des applications n'arrivent pas à *timeout*, et celles qui y arrivent ont probablement besoin d'une valeur spécifique précisée au niveau du script.

La fonction PHP **set_time_limit()** permet de modifier cette option pendant l'exécution d'un script.

On peut noter que le temps d'exécution d'un script ne compte que l'exécution du script : tout envoi de requête à un SGBD par exemple, ne compte pas dans l'intervalle défini par cette option. Seul le code PHP est pris en compte.

L'option **max_input_time** est son équivalent pour le traitement de données par formulaire. La valeur par défaut de 60 secondes me semble exagérée, mais là aussi elle n'a que peu d'influence sur le comportement de PHP.

max_execution_time : PHP_INI_ALL
max_input_time : PHP_INI_PERDIR

Cette option est importante en fonction des extensions que votre serveur utilise. Plus vous utilisez de traitements complexes, plus vous aurez besoin d'augmenter cette valeur. Par exemple, si vous utilisez des traitements graphiques avec la bibliothèque GD ou similaire, il est probable que vous ayez besoin de l'augmenter.

Personne ne peut vous dire quelle est la configuration recommandée. Tout dépend des capacités de votre serveur, de la consommation de vos scripts, des données d'entrée, *etc*

.

PHP_INI_ALL

Définit quelles erreurs doivent être rapportées par PHP. Les erreurs sont habituellement affichées dans la sortie standard et enregistrées dans un fichier de log. Comme nous le verrons ailleurs dans ce cours, PHP dispose de divers niveaux d'erreur : c'est ce niveau d'erreur qui est choisi par l'option **error_reporting**. Un niveau élevé permet de connaître plus facilement les erreurs des scripts, donc les bugs ou encore les failles de sécurité.

Depuis PHP 5, E_STRICT permet de connaître certaines bonnes pratiques de programmation. Il faut donc utiliser **E_ALL | E_STRICT** pour avoir toutes les erreurs.

E_STRICT est intégré à E_ALL à partir de PHP 5.3, mais un nouveau niveau d'erreurs E_DEPRECATED est apparu dans le même temps. Depuis PHP 5.3, pour afficher toutes les erreurs il faut donc utiliser **E_ALL | E_DEPRECATED**

.

Voici les niveaux d'erreur fréquemment utilisés :

```
error_reporting = E_ALL | E_STRICT
```

```
error_reporting = E_ALL | E_DEPRECATED
```

Configuration recommandée :

- **Développement** : le maximum possible ;
- **Production** : le maximum possible.

PHP_INI_ALL

Cette option est malheureusement très souvent méprise pour la précédente, **error_reporting**. Ces deux directives ont pourtant des objectifs bien différents : **error_reporting** définit le niveau d'erreur qui doit être filtré par PHP (donc également dans les fichiers de log du serveur), tandis que**display_errors** définit si PHP doit afficher les erreurs dans la sortie standard, indépendamment du niveau d'**error_reporting**

.

Afficher les erreurs internes de vos scripts à l'utilisateur final lui donne des informations préciseuses sur le fonctionnement et les bugs, donc les failles, de vos scripts. Afficher les erreurs est une invitation à se faire pirater. PHP offre un moyen d'enregistrer les erreurs sans les afficher : le fichier de log.


Il est donc préférable d'activer error_reporting au maximum dans toutes les situations, mais d'activer ou désactiver display_errors selon le cas.

Configuration recommandée :

- **Développement** : On ;
- **Production** : Off.

PHP_INI_ALL

Comme le dit le fichier **php.ini** lui-même, cette option n'est utile que pour le débogage (donc dans de très rares cas), d'autant que de toute manière le fichier de log standard d'Apache indique les erreurs de chargement des extensions PHP.

Configuration recommandée :

- **Développement** : On ;
- **Production** : Off.

PHP_INI_ALL

Il est évident que cette option devrait être laissée active dans toutes les situations. Sans fichier de log, vous n'auriez aucune idée des problèmes qui surviennent sur votre serveur, et vous n'auriez aucun moyen d'éviter leur récurrence.

Configuration recommandée :

- **Développement** : On ;
- **Production** : On.

PHP_INI_ALL

Dans certaines situations, il peut être pratique d'ajouter de manière automatique une chaîne au début du message d'erreur. Par exemple pour les exceptions qui ne sont pas attrapées dans le code, j'ai tendance à demander l'affichage de "<pre>" afin d'améliorer la lisibilité du message.

Configuration recommandée :

- **Développement** : "<pre>" ;
- **Production** : Off.

PHP_INI_ALL

Si vous ne souhaitez pas mixer les erreurs Apache et les erreurs PHP, cette option est pour vous. Attention aux droits d'accès au fichier de destination. Personnellement, j'ai tendance à ne pas activer cette option (ce qui est donc la configuration par défaut).

PHP_INI_ALL

Voici une autre directive qui était sensée apporter un confort d'utilisation, mais qui a été mal jugée et qui finalement a conduit à davantage de problèmes qu'elle aurait pu en régler.

Elle permet d'importer les variables autoglobales dans le scope du script, par exemple **$_GET['page']** devient automatiquement **$page**. Le problème est que les programmeurs ne filtraient pas du tout ces variables, puisqu'elles étaient déjà dans le scope de leur application, et de nombreux problèmes de sécurité en ont découlé.

Certaines applications ont encore besoin de cette option. Si vous ne pouvez vraiment pas éviter d'utiliser l'une de ces applications, je vous recommande de ne pas activer **register_autoglobals** dans votre *php.ini* pour autant, mais plutôt d'utiliser **httpd.conf** ou **.htaccess** pour définir une configuration particulière pour le script qui a besoin des autoglobales.

Configuration recommandée :

- **Développement** : Off ;
- **Production** : Off.

PHP_INI_PERDIR

Taille maximum des données que PHP accepte depuis un formulaire POST.

PHP_INI_PERDIR

Apparues dans la même période que **safe_mode**, les directives **magic_quotes_*** sont tout aussi dangereuses. Elles donnent un faux sentiment de sécurité, car les données ne sont en rien protégées contre les attaques. Cette directive devrait disparaître de *php.ini* d'ici peu, il est donc fortement recommandé de ne plus l'utiliser et de réécrire les scripts qui en ont besoin.

Configuration recommandée :

- **Développement** : Off ;

•**Production** : Off.

magic_quotes_gpc : PHP_INI_PERDIR

magic_quotes_runtime : PHP_INI_ALL
magic_quotes_sybase : PHP_INI_ALL

Le type de document par défaut envoyé par Apache au navigateur Web. Il est fortement recommandé de laisser cela à la valeur proposée, puisque la quasi totalité des scripts du marché se reposent sur cette configuration. Modifier cette valeur peut vous poser bien des ennuis.

Configuration recommandée :

- **Développement** : "text/html" ;
- **Production** : "text/html".

PHP_INI_ALL


L'encodage par défaut transmis par le serveur Web.

Là aussi, je vous recommande de ne pas modifier la configuration par défaut (option désactivée), car cela peut causer des problèmes avec certains scripts existants. Par contre, comme nous le verrons plus loin, il est souhaitable de modifier cette valeur dans tous les scripts PHP.

PHP_INI_ALL


Utile pour simplifier vos appels à **require()** ou **include()** dans vos scripts.

PHP_INI_ALL


Le répertoire vers vos extensions PHP. Préférez un chemin absolu et sans caractères spéciaux (espaces compris).

PHP_INI_SYSTEM


Autorise le chargement dynamique d'extensions PHP au moyen de la fonction **dl()**. Peut être très dangereux si vous autorisez l'exécution de code par d'autres développeurs puisque cette option vous retire la maîtrise des applications binaires disponibles sur votre propre machine.

Configuration recommandée :

- **Développement** : Off ;

•**Production** : Off.

PHP_INI_SYSTEM

Taille maximum des fichiers que PHP accepte depuis un formulaire POST. Cette valeur correspond à ce qui devrait apparaître dans le champ MAX_INPUT_SIZE de chacun de ces formulaires. Comme toujours, réduisez cette valeur au minimum, puis augmentez au niveau du répertoire (grâce au *httpd.conf* ou à un *.htaccess*) s'il y a besoin de plus.

PHP_INI_PERDIR

allow_url_fopen permet d'utiliser simplement les fichiers situés sur d'autres serveurs Web, mais il faut bien entendu faire attention à traiter ce qu'ils contiennent avec beaucoup de prudence.

allow_url_include permet d'inclure du code provenant d'un autre serveur, comme si c'était un script local. Cette technique est vivement déconseillée, car vous n'avez pas la maîtrise du contenu de scripts distants, ce qui est donc très dangereux pour la sécurité de votre serveur. De plus, inclure un script distant a un effet dramatique sur les performances.

allow_url_fopen : PHP_INI_SYSTEM
allow_url_include : PHP_INI_SYSTEM

La documentation officielle se trouve ici : http://php.net/ini

- **date.timezone** : Cette option est utilisée pour localiser le serveur dans le monde et pour adapter la valeur de retour de certaines fonctions de dates. Cette valeur devrait être mise par défaut selon votre fuseau horaire, par exemple à "Europe/Paris" si votre serveur est en France, et adaptée au moment de l'exécution de chaque script selon la localisation géographique de l'internaute.

Windows :

- **SMTP** (PHP_INI_ALL) : L'adresse du serveur SMTP à utiliser, typiquement votre hébergeur ou votre machine.
- **smtp_port** (PHP_INI_ALL) : 25 par défaut ;
- **sendmail_from** (facultatif - PHP_INI_ALL) : Adresse par défaut en tant qu'émetteur des e-mails.
-

Unix :

- **sendmail_path** (facultatif - PHP_INI_SYSTEM) : Chemin jusqu'au programme d'envoi d'e-mails, peut contenir des paramètres (par défaut : "sendmail -t -i").

- **session.save_handler** (PHP_INI_ALL) : Par défaut "files", peut être modifié dans le script avec un appel à **session_set_save_handler()** ;
- **session.save_path** (PHP_INI_ALL) : Chemin vers le magasin des sessions, *cf.* ci-dessous pour des exemples ;
- **session.use_cookies** (PHP_INI_ALL) : Doit être activé pour utiliser la transmission par cookie, ce qui est la méthode conseillée ;
- **session.name** (PHP_INI_ALL) : Le nom de la variable de session, qui est aussi le nom du cookie, par défaut "PHPSESSID" ;
- **session.auto_start** (PHP_INI_ALL) : Démarrer la session à chaque requête ;
- **session.cookie_lifetime** (PHP_INI_ALL) : Habituellement laissé à sa valeur par défaut, c'est-à-dire zéro ;
- **session.cookie_path** (PHP_INI_ALL) : Généralement laissée à la valeur par défaut, qui correspond à la racine "/" du site Web ;
- **session.cookie_domain** (PHP_INI_ALL) : Doit correspondre au domaine hébergeant l'application, par exemple **exemple.com** ou **.exemple.com** pour inclure les sous domaines ;
- **session.cookie_httponly** (PHP_INI_ALL) : Activer ce flag augmente la sécurité mais interdit l'utilisation de la session par AJAX ;
- **session.hash_function** (PHP_INI_ALL) : "0" pour MD5, ou "1" pour SHA-1 (conseillé) ;
- **url_rewriter.tags** (PHP_INI_ALL) : Utilisé si vous transmettez l'ID de session par URL.
-

Exemples pour **session.save_path** :

```
session.save_path = "/path"
```

```
session.save_path = "N;/path"
```

```
session.save_path = "N;MODE;/path"
```

- **tidy.clean_output** (PHP_INI_PERDIR) : Activer ou désactiver le nettoyage automatique : ne pas utiliser cette option si vous n'avez pas que du contenu HTML (par exemple des images, PDF, *etc*.) ; il peut être intéressant de l'activer au niveau de chaque script.

Chaque extension PHP dispose de ses propres directives dans le fichier *php.ini*, il est donc à votre charge de reporter ces directives au moment de l'activation de toute extension.

Les scripts montrés dans cette page ne sont plus à exécuter en CLI mais à partir d'un navigateur Web.

Le principe d'exécution d'un script est le suivant : PHP est utilisé par le serveur Apache uniquement pour le code entre balises PHP. Le reste du source est géré directement par Apache ou bien par un autre langage (dotNET, Ruby...) selon le cas.

Ainsi, afficher du code HTML au moyen d'un echo en PHP est une mauvaise solution car cela oblige PHP à traiter quelque chose qu'Apache pourrait faire tout seul :

```
<?php
/*
  Faire ici la récupération des variables $title, $charset et $body
  par exemple depuis un formulaire ou une BDD
*/
echo '<?xml version="1.0" encoding="'.$charset.'"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr-FR" lang="fr-FR">
<head>
    <title>'.$title.'</title>
    <meta
        http-equiv="content-type"
        content="text/html; charset='.$charset.'" />
</head>
<body>
'.$body.'
</body>
</html>';
```

Il est habituellement préférable d'utiliser PHP le moins possible, et de laisser Apache traiter directement un maximum d'affichage :

```
<?php
/*
  Faire ici la récupération des variables $title, $charset et $body
  par exemple depuis un formulaire ou une BDD
*/
?>
<?xml version="1.0" encoding="<?php echo $charset; ?>"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr-FR" lang="fr-FR">
<head>
    <title><?php echo $title; ?></title>
    <meta
        http-equiv="content-type"
        content="text/html; charset=<?php echo $charset; ?>" />
</head>
<body>
<?php echo $body; ?>
</body>
</html>
```

Ici, PHP est appelé cinq fois (dont quatre pour des tâches très courtes), tandis qu'Apache s'occupe de la majorité de l'affichage.

Un script PHP s'exécute dans un intervalle de temps défini par la variable **max_execution_time** du fichier de configuration **php.ini**, dont la valeur vaut 30 secondes par défaut.

```
max_execution_time = 30
```

Cette valeur peut être modifiée au cours de l'exécution du script par la fonction **set_time_limit()**. Il est recommandé de ne jamais l'augmenter dans la configuration globale (php.ini) mais plutôt dans les scripts individuels qui en font la demande.

Pour forcer l'arrêt d'un script en cours d'exécution, on peut utiliser les mots clefs **die()** et **exit()**. Ce sont des alias, on peut donc utiliser indifféremment l'un ou l'autre. Tous deux acceptent un paramètre optionnel : le message d'erreur à afficher.

Toutes les erreurs lancées par PHP dépendent du niveau d'erreur configuré dans le fichier **php.ini** à la valeur **error_reporting**.

Voici le niveau d'erreur recommandé en fonction de votre version de PHP :

```
error_reporting = E_ALL | E_STRICT
```

```
error_reporting = E_ALL | E_DEPRECATED
```

L'affichage des messages d'erreurs est indépendant du niveau de reporting, il est contrôlé par la directive **display_errors** du **php.ini** :

```
display_errors = On
```

```
display_errors = Off
```

Quelle que soit votre configurationdisplay_errors , il est fortement recommandé de laisser le paramètre log_errors à "On".

Avec la configuration par défaut, PHP envoie les données au fur et à mesure qu'il les calcule. Par exemple avec une boucle qui récupère des informations d'une BDD et qui les affiche immédiatement, le navigateur les reçoit petit à petit. C'est parfois pour cela que les pages se chargent par paquets, surtout si la base de données met du temps à répondre à PHP.

Cependant, il est parfois utile de conserver toutes ces informations afin d'effectuer un traitement global avant de les transmettre au navigateur. Cela peut être effectué très simplement au moyen des fonctions de contrôle de flux. Ces fonctions demandent à PHP de conserver dans une variable interne (mémoire tampon) tout ce qu'il veut envoyer à la sortie standard, et d'autres fonctions nous permettent de manipuler ce flux.

Ces fonctions sont rarement utilisées, car cela dénote souvent un problème dans la conception de l'application, c'est une sorte de rustine sur du code mal conçu. C'est le cas de **Tidy**, qui peut corriger le code HTML avant de l'envoyer au navigateur.

Cependant, il y a des utilisations intéressantes. Le module **iconv** et l'extension **mbstring** (que nous verrons plus loin) proposent des fonctions permettant de convertir le charset du texte de la page, en une seule passe au moment d'envoyer le document au navigateur : cela évite d'appeler des fonctions de conversion tout au long du code de l'application.

PHP est souvent décrit comme un moteur de templates. Cela signifie qu'il sert à récupérer des informations et à produire des documents en fonction de ces informations. Dans l'exemple ci-dessus, un document HTML est produit à partir de trois variables. En situation réelle, nous aurions probablement plusieurs accès à une base de données et quelques boucles pour afficher le contenu, filtré et organisé comme il se doit.

Structure conseillée pour un script PHP :

1. Récupération des informations (+ validation et filtres) ;
2. Construction du document (cela peut parfois être fait au moment de l'affichage) ;
3. Envoi des en-têtes HTTP (facultatif s'ils sont correctement envoyés par Apache) ;
4. Affichage du document.

Retarder l'envoi des en-têtes et l'affichage des informations jusqu'à la fin du script permet de choisir le type de document à afficher et de simplifier le débogage.

Par exemple, avec la même variable **$message**, on peut produire de nombreux documents totalement différents :

```php
<?php
//récupération des données
$message = empty($_GET['message']) ? 'Hello, world!' : $_GET['message'];

//affichage
header('Content-Type: text/html; charset=iso-8859-1');
?>
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr-FR" lang="fr-FR">
<head>
    <title><?php
        echo htmlentities($message, ENT_QUOTES, 'iso-8859-1');
        ?></title>
    <meta http-equiv="content-Type" content="text/html; charset=iso-8859-1" />
</head>
<body>
<h1><?php echo htmlentities($message, ENT_QUOTES, 'iso-8859-1'); ?></h1>
</body>
</html>
```

```php
<?php
//récupération des données
$message = empty($_GET['message']) ? 'Hello, world!' : $_GET['message'];
```

```php
//construction du document
$image = imagecreatetruecolor(100, 50);
$text_color = imagecolorallocate($image, 255, 0, 0);
imagestring($image, 1, 5, 5, $message, $text_color);

//affichage
header('Content-Type: image/png');
imagepng($image);
```

```php
<?php
//récupération des données
$message = empty($_GET['message']) ? 'Hello, world!' : $_GET['message'];

//construction du document
$t = new SWFTextField();
$t->setFont(new SWFFont('arial.ttf'));
$t->setColor(255, 0, 0);
$t->addString($message);
$movie = new SWFMovie();
$movie->add($t);

//affichage
header('Content-Type: application/x-shockwave-flash');
$movie->output();
```

```php
<?php
//récupération des données
$message = empty($_GET['message']) ? 'Hello, world!' : $_GET['message'];

//construction du document
require 'fpdf.php';
$document = new FPDF();
$document->AddPage();
$document->SetFont('Arial', 'B', 16);
$document->Cell(40, 10, $message);

//affichage
header('Content-Type: application/pdf');
$document->Output();
```

Imaginons maintenant que la variable **$message** soit remplie au moyen d'un formulaire ou bien d'une base de données. Avec différentes sources de données (formulaire, BDD,*etc.*), PHP nous permet d'obtenir des documents de types différents (HTML, image, SWF...). Nous pouvons obtenir une sortie en image, HTML, SWF ou d'autres formats tout en conservant le même bloc de code pour la récupération des données. De même, nous pouvons modifier la récupération des données (formulaire, XML, BDD...) tout en conservant le même format de sortie. C'est ce que l'on appelle la modularité, la séparation des couches ou encore l'architecture **MVC**.

```php
<?php
if (strtolower($_SERVER['REQUEST_METHOD']) == 'post')
{
    //récupération des données
    $message = $_POST['message'];

    //construction du document
    $image = imagecreatetruecolor(100, 50);
    $text_color = imagecolorallocate($image, 255, 0, 0);
    imagestring($image, 1, 5, 5, $message, $text_color);

    //affichage
    header('Content-Type: image/png');
```

```
    imagepng($image);

}
else

{   ?>
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr-FR" lang="fr-FR">
<head>
    <title>New document</title>
    <meta http-equiv="content-type" content="text/html; charset=utf-8" />
</head>
<body>
    <form method="post" action="<?php echo basename(__FILE__); ?>">
        <input type="text" name="message"/>
        <input type="submit" value="Générer le bouton"/>
    </form>
</body>
</html>
    <?php

}
```

```
<?php
//récupération des données
mysql_connect('host', 'login', 'password');
mysql_select_db('developpez');
$sql = 'SELECT text FROM message HAVING id = max(id) GROUP BY text';
$db_message = mysql_query($sql);

if ($tmp = mysql_fetch_assoc($db_message))

{   $message = $tmp['text'];

}
else

{   $message = 'Aucun message';

}
//construction du document
$image = imagecreatetruecolor(100, 50);
$text_color = imagecolorallocate($image, 255, 0, 0);
imagestring($image, 1, 5, 5, $message, $text_color);

//affichage
header('Content-Type: image/png');
imagepng($image);
```

La tâche principale de PHP est donc de récupérer des informations depuis une source externe au script, puis de les afficher dans un format défini. Dans ce cours, nous allons voir comment nous pouvons programmer la récupération des informations depuis différentes sources de données, ainsi que diverses manières de les afficher.

L'un des aspects qui font de PHP un langage dynamique, est la capacité d'inclure des scripts dans d'autres scripts. Cela permet de "mettre en facteur" des portions de code qui se répètent d'une page à l'autre.

Un exemple simple : **index.php**, **products.php**, **links.php** sont 3 pages du site ayant un en-tête de page et un pied de page identiques.

```php
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en-US" lang="en-US">
<head>
    <title><?php echo htmlentities($title, ENT_QUOTES, 'iso-8859-1'); ?></title>
    <meta http-equiv="content-type" content="text/html; charset=iso-8859-1" />
</head>
<body>
<h1><?php echo htmlentities($title, ENT_QUOTES, 'iso-8859-1'); ?></h1>
```

```php
<span class="copyright">Auteur : <?php echo $author; ?>,
    <?php echo $date; ?></span>
</body>
</html>
```

```php
<?php
$title = 'Accueil';
$author = 'Yogui';
$date = date('Y', filemtime(__FILE__));

include 'header.php';
?>
<p>Bienvenue sur notre site</p>
<p>Notre entreprise fut fondée en...</p>
<?php
include 'footer.php';
```

```php
<?php
$title = 'Nos produits';
$author = 'Yogui';
$date = date('Y', filemtime(__FILE__));

include 'header.php';
?>
<ul>
    <li>...</li>
    <li>...</li>
    <li>...</li>
    <li>...</li>
</ul>
<?php
include 'footer.php';
```

```php
<?php
$title = 'Nos partenaires';
$author = 'Yogui';
$date = date('Y', filemtime(__FILE__));

include 'header.php';
?>
<ul>
    <li><a href="..." title="...">...</a></li>
    <li><a href="..." title="...">...</a></li>
    <li><a href="..." title="...">...</a></li>
</ul>
<?php
include 'footer.php';
```

Une autre utilisation classique des inclusions est de séparer les fonctions, les classes, *etc.* : chaque script regroupe ces éléments par thème. Les conventions préconisent notamment de ne déclarer qu'une seule classe PHP par script.

Exemple :

```php
<?php
function html($string)

{   return utf8_encode htm lspecialchars($string, ENT_QUOTES));
                    (
}
```

```php
<?php
define('DB_TYPE', 'mysql');
define('DB_HOST', 'localhost');
define('DB_USER', 'utilisateur');
define('DB_PASSWORD', 'motdepasse');
define('DB_NAME', 'developpez');
```

PHP dispose de 2 instructions pour inclure un script :

- **include()** : Inclure le code du script indiqué, lancer un avertissement si le fichier est introuvable ;
- **require()** : Inclure le code du script indiqué, lancer une erreur fatale si le fichier est introuvable ;

Chacune de ces instructions se décline en une instruction ***_once()** qui oblige PHP à vérifier si le script demandé a déjà été inclus au cours de la requête actuelle (très pratique pour les déclarations de fonctions et de classes). Les fonctions *_once() ne sont pas nécessairement plus lentes à l'exécution que leurs grandes soeurs, mais elles consomment légèrement plus de mémoire. La différence étant dérisoire, il est inutile de s'en soucier.

Ainsi, ce script inclut deux fois "header.php" :

```php
<?php
include 'header.php';
include 'header.php';
```

Tandis que celui-ci ne l'inclut qu'une seule fois :

```php
<?php
include_once 'header.php';
include_once 'header.php';
include_once 'header.php';
include_once 'header.php';
include_once 'header.php';
```

Une fonction particulièrement utile pour les classes est **__autoload()**. Si cette fonction magique est déclarée dans vos scripts, alors toute classe utilisée mais n'ayant pas été chargée jusque-là, est chargée à l'aide de cette fonction.

```php
<?php
function __autoload($class)

{   require_once $class.'.php';

}
$object = new MyClass(); //chargement automatique de "MyClass.php"
```

```php
<?php
function __autoload($class)

{
```

```
    require_once str_replace ('_', '/', $class).'.php';
}
//chargement automatique de "My/Special/Class.php"
$object = new My_Special_Class();
```

Puisque nous avons fréquemment besoin de plusieurs bibliothèques dans un même projet, et que chaque bibliothèque a toutes les chances de définir son propre **__autoload()**, nous avons besoin d'un moyen de faire fonctionner tous ces __autoload() en même temps. C'est la SPL qui nous donne la fonction à utiliser : **spl_autoload_register()**

```
<?php
spl_autoload_register('basic_autoload');
spl_autoload_register('dotted_autoload');
spl_autoload_register('pear_autoload');

$object = new My_Special_Class();

function basic_autoload($class)

{   $file = $class.'.php';
    if file_exists ($file))
    { (
        require_once $file;
    }
    //echo $file.'<br/>';

}
function dotted_autoload($class)

{   $file = str_replace ('_', '.', $class).'.php';
    if file_exists ($file))
    { (
        require_once $file;
    }
    //echo $file.'<br/>';

}
function pear_autoload($class)

{   $file = str_replace ('_', '/', $class).'.php';
    if file_exists ($file))
    { (
        require_once $file;
    }
    //echo $file.'<br/>';

}
```

Enlevez les commentaires pour savoir ce qu'il se passe pendant l'exécution du script.

Le chargement automatique de classes ralentit l'exécution du code . Pour améliorer les performances, il faut utiliser un optimiseur de code ainsi qu'un cache d'opcode. Nous reviendrons sur ces notions par la suite.

Faites attention lorsque vous utilisez des scripts écrits par d'autres développeurs, qu'ils n'introduisent pas des failles de sécurité dans votre application. De même, évitez d'inclure des scripts situés sur d'autres serveurs car vous n'en avez pas le contrôle, ils peuvent être dangereux.

Il faut faire très attention à ce qui est mis à disposition sur le Web, car il est parfaitement impossible de faire oublier quelque chose à Internet. Dès lors qu'un document est disponible sur Internet, il est potentiellement déjà repris par Google (et mis en cache), des bots spammeurs, WebArchive.com... Il est futile de croire que l'on peut "supprimer" un document qui a été mis en ligne (même pour très peu de temps), et par conséquent il faut apporter un soin minutieux à ce qui peut être mis *en ligne* et à ce qui doit rester *hors ligne*

.

À cet effet, les scripts destinés à être inclus ne doivent jamais être placés dans un répertoire du serveur Web accessible depuis Internet, même si vous n'en diffusez jamais l'adresse. Ne croyez jamais que les utilisateurs ne devineront pas l'URL puisque vous ne l'avez pas devinée, ce serait sous estimer le hasard, les pirates, ou simplement notre capacité inhérente à faire des erreurs... La solution la plus sûre est toujours de ne pas tenter le diable, et ici cela se traduit par "ne pas mettre en ligne des fichiers qui n'ont pas besoin de l'être". Apprenez à maîtriser votre serveur Web (le système hôte comme le daemon HTTP), à inclure depuis PHP des fichiers qui ne sont accessibles que par le système de fichiers et non par Internet, *etc*

.

Personnellement, j'ai plusieurs répertoires :

- **/web/offline-shared** : Les bibliothèques PHP (PEAR, Zend Framework, FPDF...) ;
- **/web/offline-sites** : Les classes métier spécifiques à chaque application (classes dérivées, scripts communs...) ;
- **/web/online-http** : Scripts PHP, images, scripts JS *etc*. accessibles en ligne.

Bien entendu, dans le même ordre d'idées, le SGBD doit être configuré pour que seul votre réseau local, voire quelques machines précises, puissent s'y connecter.

Apprenez à utiliser la directive " **include_path**" de votre fichier **php.ini** afin de ne pas surcharger vos scripts de chemins absolus :

```php
<?php
set_include_path('.'
    . PATH_SEPARATOR . '/web/offline-shared'
    . PATH_SEPARATOR . '/web/offline-sites/test'
    . PATH_SEPARATOR . get_include_path());

//situé dans "/web/offline-sites/test/config.php"
include 'config.php';

//situé dans "/web/offline-shared/session-start.php"
include 'session-start.php';

//situé dans "/web/offline-sites/test/header.php"
include 'header.php';
```

Prenez garde aux noms des scripts (collisions) et à l'ordre des inclusions.

La fonction **__autoload()** est utile uniquement si **spl_autoload_register()** n'est pas appelée. Dès lors que spl_autoload_register() est appelée dans le code, __autoload() perd sa propriété magique et il faut appeler **spl_autoload_register('__autoload')**, ce qui est sémantiquement absurde. Par conséquent, il est préférable d'éviter d'utiliser __autoload() seul, au profit de **spl_autoload_register()**. Cela vous évitera des surprises en utilisant des bibliothèques développées par d'autres personnes ou en distribuant vos propres bibliothèques.

Les déclinaisons *_once() sont prévues pour les scripts qui ne sont nécessaires qu'une fois, par exemple un script de déclarations de fonctions ou de classe. C'est le besoin le plus fréquent, require() et include() sont donc moins souvent utilisables que leurs équivalents *_once().

Il faut utiliser les instructions **include** et **include_once** lorsque le script à inclure n'est pas primordial pour le bon fonctionnement du reste du programme, et les instructions **require** et **require_once** dans le cas contraire.

```php
<?php
require_once 'config.php';
require_once 'functions.php';

include_once 'header.php';
...
include_once 'footer.php';
```

La majorité des utilisateurs de notre application sont légitimes, ils utilisent simplement l'interface que nous leur proposons. Cependant, certains d'entre eux font des erreurs de manipulation innocentes et d'autres cherchent à s'approprier des droits d'accès plus élevés (piratage). Ces situations peuvent être désastreuses pour notre application, nos données et nos utilisateurs.

En aucun cas il ne faut faire confiance à une donnée provenant d'un utilisateur. Cela implique simplement de vérifier ces données avant de les utiliser, mais il faut le faire **systématiquement**.

Par exemple, si nous utilisons une variable non validée dans une requête SQL, nous sommes exposés à une faille d'**injection SQL**.

La sécurité des données s'applique à deux moments :

1. Lors de la récupération des données, il convient de les valider et/ou de les filtrer (*input validation*) ;
2. Lors de leur utilisation, il faut les convertir dans le format de sortie (*output escaping*).

Si ces deux principes ne sont pas scrupuleusement respectés, vous avez toutes les chances de vous faire pirater

Valider des informations revient à s'assurer que les variables contiennent ce qu'elles devraient contenir (leur valeur correspond à un schéma défini). La validation d'un schéma XML à l'aide d'un schéma XSD est un parfait exemple.

Filtrer les données, c'est éliminer le danger dans les variables qui ne passent pas la validation. Le filtrage des données survient donc en cas d'échec (ou à la place) de la validation.

Il faut toujours vérifier le type d'une variable. Bien sûr, le protocole HTTP transmet uniquement du texte et PHP est un langage faiblement typé. Nous avons vu ce genre de code dans la partie sur les types de données :

```php
<?php
var_dump(is_int("1")); //bool(false)
```

On ne peut pas compter sur les fonctions **is_*()** car elles sont trop restrictives. Voici des exemples plus exacts :

```php
<?php
if(!empty($_GET['id']) and ctype_digit($_GET['id']))

{   //"id" est de type numérique entier

}
else

{
```

```
    //nombre invalide, agir en conséquence
}
```

```
<?php
if (!empty($_POST['password']) and ctype_print($_POST['password']))
{   //"password" contient uniquement des caractères imprimables
}
else
{   //mot de passe impossible, agir en conséquence :
    //par exemple proposer un mot de passe aléatoire
}
```

Les fonctions ctype_*() sont un très bon moyen de vérifier le type d'une donnée. Si le type attendu est une chaîne, alors il faut souvent effectuer un contrôle plus poussé au moyen d' expressions rationnelles .

La validation doit être appliquée à toutes les données provenant de l'extérieur du programme, que ce soit d'un formulaire, d'une variable superglobale, d'un fichier (même s'il est placé sur le serveur), d'une base de données, *etc* Toute variable utilisée sans avoir été validée **dans le script** est un risque pour le reste de l'application, pour votre base de données, pour vos données client... .

Les fichiers transmis par des utilisateurs sont à considérer avec autant de vigilence. La validation d'un fichier se fait raisonablement par son type. Ne vous fiez ni à l'extension du fichier, ni au type fourni fans le tableau $_FILES puisque ce sont des informations transmises par le navigateur : s'il s'agit d'un piratage, ces informations sont sans doute falsifiées pour vous induire en erreur. Le type MIME d'un fichier peut être déterminé grâce à l'extension **Fileinfo** .

Si la validation échoue, il faut filtrer les données afin qu'elles correspondent au type et au contenu attendus. Cela se fait au moyen du transtypage pour les types autres que les chaînes, ou avec des expressions rationnelles pour les chaînes.

Dans le cas de nombres entiers (qui constituent la majorité des valeurs transmises par l'utilisateur), le plus simple est le transtypage systématique :

```
<?php
if (empty($_GET['id']))
{   $id = 0;
}
else
{   $id = (int)$_GET['id']; //on transtype "id" au type numérique entier
}
```

Pour certains types de données complexes et pour les fichiers, si la validation échoue, alors il peut être judicieux d'arrêter le script plutôt que d'essayer de filtrer la valeur. Par exemple pour un document XML, il est parfois impossible (ou trop complexe) de reconstruire un document qui puisse remplacer le document invalide.

Chris Shiflett, expert sécurité, propose de mettre les variables filtrées dans un tableau PHP afin de mettre en valeur le fait qu'elles sont filtrées :

```
<?php
$clean = array();
if (empty($_GET['id']))
{
```

```
    $clean['id'] = 0;
}
else
{   //on transtype "id" au type numérique entier
    $clean['id'] = (int)$_GET['id'];
}
```

Une variable filtrée ne doit pas pour autant être utilisée sans précautions. On sait simplement que sa valeur correspond à ce que l'on en attend, par exemple ce n'est pas un mot de passe à la place d'un identifiant numérique. L'utilisation de la variable est une autre histoire, elle dépend du contexte de destination.

Lors de l'utilisation d'une donnée (affichage, envoi dans une requête SQL ou dans une commande shell...), il faut systématiquement protéger la valeur, la convertir dans le format attendu par le destinataire.

Si l'on souhaite afficher la valeur dans une page Web, il faut utiliser une des fonctions suivantes :

- **utf8_encode()** : Pour afficher au format UTF-8 (approche recommandée pour les chaînes UTF-8) ;
- **htmlentities()** : Pour convertir tous les caractères en leur entité HTML correspondante, attention à bien utiliser les **deux** paramètres optionnels ;
- **htmlspecialchars()** : Pour convertir uniquement les entités HTML fondamentales (fonction insuffisante si elle est utilisée seule).

```
<?php
$string = "Developpez.com, le club des développeurs.";

header('Content-Type: text/html; charset=iso-8859-1');
echo htmlentities($string, ENT_QUOTES, 'iso-8859-1');
```

```
Developpez.com, le club des d&eacute;veloppeurs.
```

Une notion fondamentale en sécurité Web est la faille **XSS** (*cross-site scripting*). Il s'agit simplement de tromper l'utilisateur et de lui faire exécuter du code destiné à un autre site. C'est possible si une variable utilisateur est affichée sans protection, par exemple avec **echo $_GET['login']** on peut facilement produire une faille XSS.

Comment se protéger de la faille XSS ?

La seconde notion fondamentale est la faille **CSRF** (*cross-site request forgery*), légèrement plus complexe à mettre en place que XSS du point de vue du pirate, mais elle est également très facile à éviter pour le développeur du site. Nous y reviendrons en parlant des formulaires.

Dans le cas d'attaques XSS ou CSRF, les utilisateurs de votre site sont des victimes à 100%. Ils ne savent pas ce qu'il se passe avant qu'il soit trop tard (en supposant qu'ils le sachent un jour).

Les autres menaces principales au moment de l'utilisation des variables sont des "failles d'injection".

De nombreux articles décrivent les bonnes méthodes pour se protéger des injections, notamment :

- **Comment se protéger des failles d'injection ?**
- **Développement web : Généralités sur la sécurité**, par Julien Pauli

Dans le cas d'une requête SQL, le meilleur moyen est d'utiliser des **requêtes préparées** : on ne traite alors plus des chaînes mais leur représentation hexadécimale (donc inoffensive).

C'est possible avec les APIs OO, notamment :

- **MySQLi** : $db->prepare()
- **PDO** : $db->prepare()

Faites bien attention à ne jamais utiliser une variable non filtrée ou non validée. La majorité des failles de sécurité sont dues à cette erreur.

Plusieurs approches peuvent vous aider :

- Une approche de type "**poka-yoké**" vous évite d'utiliser une donnée sans utiliser une méthode explicite de filtrage ;
- Une extension PHP comme **GRASP** ou **celle proposée par Wietse Venema**.

Comme le dit très bien M. Wietse, aucune extension ne pourra vous dire dans 100% des cas et avec 100% de certitude, que vous avez ou n'avez pas protégé correctement vos données. Les outils sus mentionnés sont des **aides** pour développer avec le moins d'erreurs possibles, mais c'est aussi votre rôle de programmeur d'auditer votre code.

Dans un monde idéal, ne stockez jamais une valeur "untainted", par exemple le résultat de htmlentities() . Il est préférable de l'utiliser directement dans la fonction de destination, par exemple echo ou mysql_query() . Cela permet de rester conscient du fait qu'une donnée est convertir selon son contexte.

```php
<?php
/*
 * connexion au SGBD...
 */

$clean = array();
$clean['login'] = htmlspecialchars($_POST['login'], ENT_QUOTES);

//possibilité d'injection SQL
$db->query(sprintf(
    "SELECT id, name FROM user WHERE name='%s'",
    $clean['login']));
```

Le code ci-dessus présente deux erreurs : d'une part la valeur n'est pas protégée au format SQL ("escape" ou requête préparée), et d'autre part ce qui est enregistré dans la BDD est vraisemblablement un format HTML plutôt qu'une représentation brute. Ce dernier point est une erreur fondamentale d'analyse des besoins : n'enregistrez jamais la conversion d'un texte, à moins d'enregistrer également la version brute.

Lorsqu'un client demande une page Web, c'est-à-dire lorsqu'un internaute clique sur un lien ou valide une adresse dans son navigateur, il envoie une requête HTTP au serveur Web. Le protocole HTTP est décliné en plusieurs

versions : celles qui sont le plus couramment utilisées sur le Web sont la 1.0 et la 1.1, et **sont décrites en détail par Mathieu Lemoine**
.

```
GET /cours/ HTTP/1.1
Host: example.org
User-Agent: Mozilla/1.4
Accept: text/xml, image/png, image/jpeg, image/gif, */*
```

Lorsque le serveur Web reçoit une telle demande, il évalue comment il peut y répondre (quel code HTTP renvoyer), puis il construit la réponse et il envoie le tout.

```
HTTP/1.1 200 OK
Content-Length: 61

<html>
<body>
<img src="http://example.org/image.png" />
</body>
</html>
```

On voit très bien ici que la réponse HTTP et la page demandée font partie du même envoi. Le serveur Web envoie le code réponse HTTP et le document à la suite l'un de l'autre, comme s'il s'agissait d'un seul document. Les en-têtes HTTP ne peuvent pas être dissociés du document de réponse, ils ne peuvent être envoyés ni au milieu du document ni en plusieurs parties. L'ensemble des en-têtes HTTP doivent donc être prêts avant que le serveur commence à envoyer le document au navigateur qui en a fait la demande.

C'est pour cela qu'utiliser la fonction**header()** après avoir commencé à envoyer du contenu, mène à l'erreur "headers already sent". Tout ce qui est echo, print, var_dump, print_r *etc*. sert à transmettre des informations au client, il faut donc terminer l'envoi des en-têtes avant de les utiliser.

C'est aussi pour cela qu'il est impossible d'envoyer deux documents différents en une seule réponse. Une réponse HTTP a un type unique. Lorsqu'une page Web contient des images, des feuilles de style, des animations Flash, des scripts Javascript *etc*., le serveur envoie chaque élément dans une réponse HTTP séparée, chacun à la demande du navigateur. Le navigateur détermine les éléments à demander au serveur en fonction du document HTML qu'il reçoit ainsi que des préférences de l'utilisateur (*ie*. désactiver JavaScript ou ne pas afficher d'images), et chaque demande se fait sous la forme d'une requête HTTP.

Pour rappel, voici l'ordre que j'ai proposé dans un paragraphe précédent :

1. Construction du document demandé ;
2. Envoi des en-têtes HTTP ;
3. Envoi du document.

Sous Apache, vous pouvez voir l'ensemble des transactions HTTP entre un navigateur et votre serveur Web dans le fichier apache/logs/access.log . C'est une excellente source d'informations pour un administrateur consciencieux, et vous devriez fréquemment analyser ce fichier log.

Des en-têtes doivent être envoyés à chaque transaction. Le serveur Web est généralement configuré pour envoyer certains en-têtes par défaut, mais le développeur peut les remplacer ou les compléter s'il le juge nécessaire.

Le fichier *php.ini* dispose par exemple de deux directives **default_mimetype** et **default_charset**, qui permettent de définir un type par défaut de contenu et son jeu de caractères. La première est habituellement laissée à "text/html",

tandis que la deuxième est généralement mise de côté. En effet, la majorité des requêtes concernent des pages Web au format texte/html, mais pour les autres requêtes le jeu de caractères n'a probablement pas de sens.

L'envoi explicite d'en-têtes depuis le code PHP n'est donc pas systématiquement nécessaire. Le serveur Web se charge de transmettre l'en-tête suivant dans la majorité des cas :

```
HTTP/1.1 200 OK
Content-Type: text/html
```

Le document doit alors correspondre à cet en-tête afin que le navigateur puisse l'interpréter. Par exemple, il serait mal venu d'envoyer une image avec ce Content-Type... C'est ici qu'intervient la fonction **header()** en PHP.

La fonction header() permet d'envoyer un en-tête HTTP brut, en remplaçant un en-tête précédent similaire. Nous savons que le serveur Web envoie systématiquement le Content-Type text/html, mais si nous avons le code suivant, cet en-tête sera remplacé par le nôtre :

```
<?php
header('Content-Type: image/png');
```

```
HTTP/1.1 200 OK
Content-Type: image/png
```

Nous pouvons ainsi dire au navigateur de l'internaute quel est le type exact du contenu que nous lui envoyons. C'est de cette manière que l'on évite les problèmes d'accents, force le téléchargement d'un fichier, évite la mise en cache du document, *etc*

.

À titre d'illustration, voici une requête complète sur une page générant une erreur PHP :

```
<?php
header('HTTP/1.1 500 Internal Server Error');
```

```
GET /tests/error.php HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.8.1.11)
    Gecko/20071127 Firefox/2.0.0.11
Accept: text/xml,application/xml,application/xhtml+xml,text/html;
    q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Cache-Control: max-age=0
```

```
HTTP/1.x 500 Internal Server Error
Date: Mon, 31 Dec 2007 22:41:19 GMT
Server: Apache/2.2.4 (Win32)
Content-Length: 535
Connection: close
Content-Type: text/html; charset=iso-8859-1
```

```
HTTP/1.x 500 Internal Server Error
Date: Mon, 31 Dec 2007 22:39:42 GMT
Server: Apache/2.2.4 (Win32) PHP/5.3.0-dev
X-Powered-By: PHP/5.3.0-dev
Content-Length: 0
```

```
Connection: close
Content-Type: text/html
```

```
HTTP/1.x 500 Internal Server Error
Date: Mon, 31 Dec 2007 22:40:04 GMT
Server: Apache/1.3.35 (Win32) PHP/5.2.5
X-Powered-By: PHP/5.2.5
Connection: close
Transfer-Encoding: chunked
Content-Type: text/html
```

Prenez garde aux injections de headers si vous utilisez des variables utilisateur dans vos en-têtes. Il est très facile d'injecter un en-tête dans une application qui ne se protège pas, et cela peut avoir des conséquences dramatiques pour vos utilisateurs. Cette mise en garde est valable aussi bien pour un en-tête HTTP que pour l'en-tête d'un e-mail.

Si vous êtes sûr de la configuration de votre serveur, n'envoyer que les en-têtes HTTP nécessaires vous fera gagner du temps de développement. Sinon, leur envoi explicite est un bon moyen de vous assurer que le navigateur traite correctement le document qui lui est envoyé dans la suite de la réponse. Le développeur peut s'appuyer sur la configuration du serveur Web, mais il est conseillé d'envoyer des en-têtes personnalisés à chaque réponse HTTP. Renseignez-vous sur le protocole HTTP, sur les particularités des navigateurs Web à ce sujet et souvenez-vous qu'un en-tête HTTP n'est pas perçu comme un *ordre* par le navigateur, mais plutôt comme un *conseil*

Live HTTP Headers est une excellente extension pour Firefox, elle vous permet de voir quels en-têtes sont envoyés ou reçus par votre navigateur.

Les liens sont la plus commune des manières de transmettre une demande de l'utilisateur, et d'introduire du dynamisme dans un site Web.

Sans doute connaissez-vous ce lien :

```
http://www.google.com/search?hl=en&q=php
```

Sa signification est transparente : *charger la page "search" sur le site "google.com", avec les paramètres "hl" en anglais et la question "php"*

C'est grâce aux paramètres (à savoir ce qui suit le "?") que Google sait que je veux mes résultats en anglais et pour le mot clef "php". Si je modifie l'un de ces paramètres, Google me répond autre chose.

Les paramètres GET sont mis dans le tableau superglobal **$_GET** de PHP. C'est un tableau associatif, donc l'exemple ci-dessus se traduit par :

```
Array
(
    [hl] => en
```

```
    [q] => php
)
```

URL est l'acronyme d'Uniform Resource Locator : cela signifie que, d'un chargement à l'autre de la même URL, le document résultant doit être sensiblement équivalent. On considère les modifications entre deux chargements comme des "mises à jour" et non comme des "contenus différents". L'URL est l'identifiant privilégié pour avoir accès à une ressource. Une URL pointe vers un seul document, et un document n'est accessible que par une seule URL active.

Si vous avez plusieurs URLs pour une même ressource, vous devriez mettre en place au plus vite des redirections HTTP.

La superglobale **$_SERVER** permet de retrouver la requête originale du navigateur sous diverses formes. Le contenu exact de ce tableau dépend de votre configuration, mais certaines variables se retrouvent, en particulier REQUEST_URI et QUERY_STRING :

```
http://localhost/tests/error.php?hl=en&q=php
```

```
Array
(
    [ZendEnablerConfig] => C:/Program Files/Zend/Core/etc/fastcgi.conf
    [PHP_FCGI_MAX_REQUESTS] => 10000
    [PHP_FCGI_CHILDREN] => 1
    [PATH] => /* ma variable système PATH */
    [TEMP] => C:\Program Files\Zend\Core\temp
    [OS] => Windows_NT
    [SystemRoot] => C:\WINDOWS
    [ComSpec] => C:\WINDOWS\system32\cmd.exe
    [_FCGI_MUTEX_] => 1888
    [_FCGI_SHUTDOWN_EVENT_] => 1892
    [_FCGI_NTAUTH_IMPERSONATE_] => 1
    [FCGI_ROLE] => RESPONDER
    [HTTP_HOST] => localhost
    [HTTP_USER_AGENT] => Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US;
        rv:1.8.1.11) Gecko/20071127 Firefox/2.0.0.11
    [HTTP_ACCEPT] => text/xml,application/xml,application/xhtml+xml,text/html;
        q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
    [HTTP_ACCEPT_LANGUAGE] => en-us,en;q=0.5
    [HTTP_ACCEPT_ENCODING] => gzip,deflate
    [HTTP_ACCEPT_CHARSET] => ISO-8859-1,utf-8;q=0.7,*;q=0.7
    [HTTP_KEEP_ALIVE] => 300
    [HTTP_CONNECTION] => keep-alive
    [HTTP_CACHE_CONTROL] => max-age=0
    [COMSPEC] => C:\WINDOWS\system32\cmd.exe
    [PATHEXT] => .COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH
    [WINDIR] => C:\WINDOWS
    [SERVER_SIGNATURE] =>
    [SERVER_SOFTWARE] => Apache/2.2.4 (Win32)
    [SERVER_NAME] => localhost
    [SERVER_ADDR] => 127.0.0.1
    [SERVER_PORT] => 80
    [REMOTE_ADDR] => 127.0.0.1
    [DOCUMENT_ROOT] => C:/Web/online/http
    [SERVER_ADMIN] => @@ServerAdmin@@
    [SCRIPT_FILENAME] => C:\Web\online\http\tests\error.php
    [REMOTE_PORT] => 1125
    [GATEWAY_INTERFACE] => CGI/1.1
    [SERVER_PROTOCOL] => HTTP/1.1
    [REQUEST_METHOD] => GET
    [QUERY_STRING] => hl=en&q=php
    [REQUEST_URI] => /tests/error.php?hl=en&q=php
    [SCRIPT_NAME] => /tests/error.php
    [ORIG_SCRIPT_FILENAME] => C:/Web/online/http/tests/error.php
```

```
    [PHP_SELF] => /tests/error.php
    [REQUEST_TIME] => 1199154411
    [argv] => Array
        (
            [0] => hl=en&q=php
        )

    [argc] => 1

)
```

Afin de simplifier la résolution des problèmes de conversion et de sécurité posés par la transmission de chaînes de caractères, il est préférable de transmettre autant que possible des valeurs numériques. Ces valeurs sont transmises en tant que texte pendant les échanges HTTP, mais elles n'ont aucun encodage spécial et nos scripts peuvent les filtrer sans complications.

Prendez garde à choisir des noms de paramètres utilisant uniquement l'alphabet anglais, des chiffres et pas d'espace, cela simplifie les opérations.

Il reste à parler de l'encodage, mais c'est un sujet qui mérite son propre chapitre.

Par défaut, le protocole HTTP permet de faire des liens paramétrés que tout langage de script sait interpréter.

```
http://www.exemple.com/products.php?family=97&manufacturer=46
```

L'un des problèmes de cette approche est que l'URL est relativement laide pour un oeil humain, et par conséquent les humains la lisent rarement. Cela peut même conduire à certaines techniques de **phishing**, dans la mesure où personne ne fait plus attention au lien sur lequel il clique. Par ailleurs, les moteurs de recherche ont tendance à ne pas référencer toutes les URLs qui leurs semblent dynamiques, car ils savent qu'il y a trop de combinaisons possibles. Enfin, ces mêmes moteurs ne peuvent pas toujours trouver de mots clefs dans une URL dynamique, à cause de la majorité d'identifiants utilisés. Par exemple ci-dessus, "manufacturer" et "family" sont tout à fait inutiles pour le référencement de mon site, alors que le nom de la famille et du constructeur seraient d'une grande aide à la place de leurs identifiants.

Accessoirement, le mot clef "php" est enregistré ici pour une page dans laquelle il n'est aucunement question de PHP ou même de programmation.

Voici un exemple de conséquence : cherchez un podcast ou une conférence PHP sur eMule simplement en tapant "php" avec un type "audio". Vous trouverez de nombreux fichiers nommés "*.php.mp3" car ils ont probablement été téléchargés depuis un script PHP. L'apparition de "php" dans les noms des morceaux n'a aucune valeur mercatique, pourtant il apparaît et cela produit de très nombreux faux résultats dans nos recherches.

Un remède à certains de ces élément pourrait être une URL de ce type :

```
http://www.exemple.com/products/97/46/
```

Nous n'affichons pas les noms de variables (inutiles pour le référencement), l'URL est plus courte, on n'a pas l'impression d'avoir affaire à des paramètres mais à une arborescence de fichiers HTML statiques. Les avantages semblent être nombreux, mais l'inconvénient majeur est que nous avons peu de mots clefs. L'URL est propre, mais

elle est peu efficace : "47" et "96" ne reflètent pas le contenu de ma base de données, ils n'aident pas au référencement de mon site.

Voici une solution vraisemblablement idéale :

```
http://www.exemple.com/products/processeurs/amd/
```

Si le serveur Web sait interpréter cette URL avec les mêmes paramètres que dans l'exemple précédent, alors elle est bien plus efficace.

Malheureusement, l'un des problèmes de cette approche est que les identifiants sont numériques dans la BDD (contrairement aux chaînes "processeurs" et "amd"), que le nom du fabricant peut être modifié à tout moment (ce qui aurait pour fâcheuse tendance de produire des erreurs 404 partout dans votre site) et que rien n'indique qu'il utilise exclusivement l'alphabet anglais en lettres minuscules (rappelez-vous que ce sont des impératifs pour une bonne URL). Et il y a aussi le problème de l'internationalisation (*i18n*).

Ainsi, une alternative acceptable (parmi d'autres) peut être :

```
http://fr.exemple.com/produits/97-processeurs/46-amd/
```

Mais bien sûr, de telles URLs ne correspondent pas à des fichiers sur le serveur Web car c'est impraticable, prend trop de place sur le disque ou bien augmente trop les accès disques. Il faut voir ces URLs comme un moyen virtuel d'avoir accès à une véritable ressource, mais c'est le serveur Web qui s'occupe de l'aspect virtuel (de manière totalement transparente pour l'internaute). Tous les grands serveurs Web du marché savent gérer ce que l'on appelle "URL Rewriting", ou encore "réécriture de liens".

Pour plus de renseignements : **Tutoriel d'URL Rewriting**

Construire un lien ne se fait pas à la légère. Renseignez-vous sur le **Webmarketing** si vous souhaitez approfondir (ce que je vous recommande), mais sachez déjà que le plus important est d'avoir des **URLs courtes** et d'y mettre des **mots clefs**. Un lien efficace est uniquement construit par un travail particulièrement soigné.

L'un des aspects passe par le respect des standards. Lisez la documentation concernant le *doctype* que vous utilisez, que ce soit *HTML 4 Transitionnal* ou *XHTML 1 Strict* (pour ne citer que les extrêmes). Dans la plupart des cas, vous aurez affaire aux mêmes éléments du lien : **href**, **title** et le texte affiché dans la page. Chacun de ces éléments porte une sémantique précise et différente des autres, il faut donc les utiliser avec dicernement.

Il faut par exemple éviter de négliger la propriété **title**, qui permet d'ajouter des mots clefs parfois précieux.

Un élément fondamental de la balise *lien* est bien évidemment la propriété **href**. Il faut donc y apporter le plus grand soin. Le texte ou le titre d'un lien peuvent changer, mais l'URL (comme son nom l'indique) est un identifiant universel. Il ne faut changer l'URL d'un document qu'après mûre réflexion, et après s'être assuré que l'ancienne URL redirige correctement vers la nouvelle.

L'évolution d'un site cause parfois le changement de technologie (par ex. de JSP vers PHP), ce qui a des conséquences parfois dramatiques sur le référencement (erreurs 404 *etc*.) ; il faut donc à tout prix éviter de montrer

l'extension du script dans son URL. Cela permet au serveur de changer de langage de script sans changer l'URL de chacun des documents.

Demandez-vous ce qu'un utilisateur attentif aimerait voir apparaître dans l'URL comme indicateur de ce que contient le document. Utilisez aussi vos notions de Webmarketing et votre expérience de l'évolution d'un site pour déterminer la meilleure URL pour chaque document.

L'en-tête HTTP "Content-Type" contient un paramètre "charset". La valeur de ce paramètre permet au navigateur de savoir dans quel jeu de caractères est encodé le document qu'il est en train d'afficher. Par exemple, un document encodé en **iso-8859-1** ne peut pas contenir certains caractères accentués dans son code source. Il existe néanmoins des équivalents appelés **entités HTML** : c'est ce que permet d'obtenir la fonction PHP **htmlentities()**.

Comprendre toute la problématique des jeux de caractères est un travail fastidieux qui ne présente pas un grand intérêt. Chaque jeu de caractères est une représentation informatique des caractères utilisés dans un groupe de langues, par exemple ISO-8859-1 contient suffisamment de caractères pour représenter tous les caractères utilisés dans 24 langues (*cf.* **Wikipedia**), ce qui n'est pas l'ensemble des langues au monde. Or nous nous intéressons ici à Internet, espace de rencontre et d'échange de toutes les cultures du monde : chaque site doit être en mesure d'afficher du texte issu de n'importe quelle langue, **UTF-8** est donc le choix le plus judicieux à cet effet.
Les autres choix sont UTF-7, envers lequel l'*Internet Mail Consortium* a émis un avis défavorable ; et UTF-16, dont la représentation interne n'est pas compatible avec les formats antérieurs, ce qui implique de nombreux problèmes de migration des applications.

La fonction **htmlspecialchars()** permet d'encoder uniquement les caractères ayant une signification en XML (et par conséquent en HTML et en XHTML). La fonction inverse est **html_entity_decode()** et l'ensemble du jeu de caractères peut être déterminé par la fonction **get_html_translation_table()** :

```
<pre><?php
print_r(get_html_translation_table(HTML_SPECIALCHARS, ENT_QUOTES));
```

```
Array
(
    ["] => &quot;
    ['] => &#39;
    [<] => &lt;
    [>] => &gt;
    [&] => &amp;
)
```

La fonction **htmlentities()** renvoie la représentation HTML de chacun des caractères, donc les mêmes que pour htmlspecialchars() ainsi que les caractères accentués, *etc*. La fonction inverse est **html_entity_decode()** et l'ensemble du jeu de caractères peut être déterminé par la fonction **get_html_translation_table()** :

```
<pre><?php
print_r(get_html_translation_table(HTML_ENTITIES, ENT_QUOTES));
```

```
Array
(
```

```
    [ ] =>  
    [¡] => &iexcl;
    [¢] => &cent;
    [£] => &pound;
    [¤] => &curren;
    [¥] => &yen;

    [¦] => &brvbar;
    [§] => &sect;
    [¨] => &uml;
    [©] => &copy;
    [ª] => &ordf;
    [«] => &laquo;

    [¬] => &not;
    [-] => &shy;
    [®] => &reg;
    [¯] => &macr;
    [°] => &deg;
    [±] => &plusmn;

    [²] => &sup2;
    [³] => &sup3;
    [´] => &acute;
    [µ] => &micro;
    [¶] => &para;
    [·] => &middot;

    [¸] => &cedil;
    [¹] => &sup1;
    [º] => &ordm;
    [»] => &raquo;
    [¼] => &frac14;
    [½] => &frac12;

    [¾] => &frac34;
    [¿] => &iquest;
    [À] => &Agrave;
    [Á] => &Aacute;
    [Â] => &Acirc;
    [Ã] => &Atilde;

    [Ä] => &Auml;
    [Å] => &Aring;
    [Æ] => &AElig;
    [Ç] => &Ccedil;
    [È] => &Egrave;
    [É] => &Eacute;

    [Ê] => &Ecirc;
    [Ë] => &Euml;
    [Ì] => &Igrave;
    [Í] => &Iacute;
    [Î] => &Icirc;
    [Ï] => &Iuml;

    [Ð] => &ETH;
    [Ñ] => &Ntilde;
    [Ò] => &Ograve;
    [Ó] => &Oacute;
    [Ô] => &Ocirc;
    [Õ] => &Otilde;

    [Ö] => &Ouml;
    [×] => &times;
    [Ø] => &Oslash;
    [Ù] => &Ugrave;
    [Ú] => &Uacute;
    [Û] => &Ucirc;

    [Ü] => &Uuml;
```

```
    [Ý] => &Yacute;
    [Þ] => &THORN;
    [ß] => &szlig;
    [à] => &agrave;
    [á] => &aacute;

    [â] => &acirc;
    [ã] => &atilde;
    [ä] => &auml;
    [å] => &aring;
    [æ] => &aelig;
    [ç] => &ccedil;

    [è] => &egrave;
    [é] => &eacute;
    [ê] => &ecirc;
    [ë] => &euml;
    [ì] => &igrave;
    [í] => &iacute;

    [î] => &icirc;
    [ï] => &iuml;
    [ð] => &eth;
    [ñ] => &ntilde;
    [ò] => &ograve;
    [ó] => &oacute;

    [ô] => &ocirc;
    [õ] => &otilde;
    [ö] => &ouml;
    [÷] => &divide;
    [ø] => &oslash;
    [ù] => &ugrave;

    [ú] => &uacute;
    [û] => &ucirc;
    [ü] => &uuml;
    [ý] => &yacute;
    [þ] => &thorn;
    [ÿ] => &yuml;

    ["] => &quot;
    ['] => &#39;
    [<] => &lt;
    [>] => &gt;
    [&] => &amp;


)
```

La fonction **utf8_encode()**, quant à elle, renvoie une représentation UTF-8 de la chaîne en paramètre. L'encodage en UTF-8 permet de représenter plus de jeux de caractères que les jeux latins. Un jeu de caractères ISO-8859-1 est représenté par un octet, soit 2^8 valeurs possibles. UTF-8 dispose de 4 octets, soit 2^32 caractères possibles (moins les caractères de contrôle).

Prenez soin de toujours encoder les caractères en fonction du format de sortie (HTML ou autre). Ne pas utiliser htmlspecialchars() rend vos visiteurs vulnérables à des attaques XSS, et ne pas utiliser un format d'encodage unique tout au long du document génère des caractères illisibles (cf. quelques chapitres plus loin pour des exemples).

La meilleure tactique est de tout encoder en UTF-8. De cette manière, vous pouvez même afficher dans vos pages des caractères issus de jeux de caractères qui n'ont pas d'équivalent HTML, comme les langages dont l'alphabet n'est pas latin (arabe, chinois, japonais...).

Vous pouvez aussi utiliser **htmlentities()** sur toutes vos chaînes encodées en UTF-8 car cela produit du HTML indépendant du jeu de caratères, et le navigateur le comprend parfaitement l'encodage HTML. C'est un bon moyen d'éviter les erreurs si vous ne maîtrisez pas totalement le sujet, mais ce n'est pas la solution absolue puisque les jeux de caractères autres que l'UTF-8 ont généralement des lacunes : votre site serait hermétique à certaines langues.

En fait, tout le texte de vos pages Web devrait être affiché de cette manière :

```
echo utf8_encode htmlspecialchars($text, ENT_QUOTES));
                (
```

Le script lui-même (en tant que fichier) porte un encodage dans le système de fichiers. C'est votre éditeur de code qui vous permet de modifier cet encodage, dont les valeurs les plus courantes sont "ANSI" ou bien "UTF-8".

L'encodage du script influe la manière dont PHP traite les chaînes de caractères présentes dans les scripts eux-mêmes.

Si vous essayez par exemple d'utiliser des caractères japonais dans vos scripts, vous aurez de mauvaises surprises si vos scripts sont encodés en ANSI. Certains EDI sont très confus à cause de cela. Modifiez l'encodage interne du script à "UTF-8" et vous pourrez utiliser les caractères du langage que vous voulez, à l'intérieur de vos scripts.

Ici par exemple, j'essaie de faire tenir le mot "japonais" (en japonais) dans un script encodé en UTF-8 puis en ANSI :

Il semble évident qu'UTF-8 doit être l'encodage de préférence pour vos scripts. Pensez à le préciser de manière bien visible dans la documentation de votre projet.

L'URL est affichée ainsi dans la barre d'adresse du navigateur :

```
http://localhost/tests/error.php?hl=en&q=php
```

Pourtant, il faut l'encoder en entités HTML dans le code source HTML :

```
<html>
<body>
http://localhost/tests/error.php?hl=en&amp;q=php
</body>
</html>
```

Pour y parvenir, PHP dispose des fonctions **htmlspecialchars()** et **htmlentities()**. La première convertit uniquement les caractères ayant une signification spéciale, tandis que l'autre convertit en entités HTML tous les caractères ayant une correspondance dans le jeu de caractères sélectionné en 3° paramètre.

On peut aussi utiliser la fonction ord() pour convertir du texte brut en texte HTML, mais cela produit des documents bien plus volumineux :

```
<?php
echo '&#'.ord('a'); //envoie "&#97" dans le source HTML, que le navigateur affiche "a"
```

La solution la plus efficace est soit d'utiliser **htmlentities()** avec le bon **charset** si on sait que le site n'est disponible que dans un certain nombre de langues (ce qui est extrêmement rare), soit d'utiliser **htmlspecialchars()** en convertissant le texte dans le **charset** de destination (par exemple UTF-8). Puisqu'htmlentities() ne sait pas gérer toutes les langues au monde, la solution avec **htmlspecialchars()** est largement meilleure :

```php
<?php
$string = "...Du texte ici...";

header('Content-Type: text/html; charset=utf-8');
echo utf8_encode(htmlspecialchars($string, ENT_QUOTES));
```

Un autre aspect à maîtriser est la barre d'adresse du navigateur, qui représente le format des URLs dans les requêtes HTTP. Ce format est défini dans la **RFC 1738**.

Si je charge l'URL suivante dans mon navigateur, il la traduit avant de la transmettre au serveur Web :

```
http://localhost/test.php?q=forum cinéma
```

```
GET /test.php?q=forum+cin%E9ma HTTP/1.1
Host: localhost
```

Il faut encoder séparément au format "RFC 1738" tous les éléments de la "query string" (donc ici la valeur "forum cinéma"), mais pas le reste de l'URL. D'excellents commentaires de la documentation officielle expliquent très clairement ce qui doit être encodé et avec quelle fonction (urlencode() ou bien rawurlencode()) :

- **http://fr2.php.net/manual/fr/function.rawurlencode.php#25182**
- **http://fr2.php.net/manual/fr/function.rawurlencode.php#26869**
- **http://fr2.php.net/manual/fr/function.urlencode.php#56426**
- **http://fr2.php.net/manual/fr/function.urlencode.php#71706**

Tous les caractères non alphanumériques et hors "-_." ont un code "*URL encoded*". Par exemple, le simple mot "cinéma" est encodé ainsi pour une URL (attention à l'encodage interne de votre script *cf.* les options de votre éditeur de code) :

```php
<?php

//affiche "forum+cin%E9ma"
echo urlencode('forum cinéma');

//affiche "forum%20cin%E9ma"
echo rawurlencode('forum cinéma');

//affiche "forum+cin%C3%A9ma"
echo urlencode(utf8_encode('forum cinéma'));

//affiche "forum%20cin%C3%A9ma"
echo rawurlencode(utf8_encode('forum cinéma'));
```

```php
<?php

//affiche "forum+cin%E9ma"
```

```
echo urlencode utf8_decode('forum cinéma')).'<br/>';
                (
//affiche "forum%20cin%E9ma"
echo raw urlencodeutf8_decode('forum cinéma')).'<br/>';
                  (
//affiche "forum+cin%C3%A9ma"
echo urlencode('forum cinéma').'<br/>';

//affiche "forum%20cin%C3%A9ma"
echo raw urlencode('forum cinéma').'<br/>';
```

On voit ici que la représentation interne des caractères du script (encodage du script, ou *file encoding*) a une incidence sur la représentation "URL encoded". Dans l'idéal, la représentation URL des caractères doit être identique au format d'encodage du document, mais c'est très complexe à mettre en place de manière exacte. Malheureusement, et bien que les navigateurs se débrouillent très bien à ce niveau, leur laisser un peu de liberté débouche trop souvent sur une faille de sécurité (même si la faute est bien souvent du site et non du navigateur). Ma recommandation est de tout encoder au format UTF-8, qui tend à devenir le standard sur Internet. L'arrivée d'Unicode dans PHP6 permettra de clarifier tout cela.


PHP décode automatiquement les caractères encodés selon la RFC 1738 avant le début du script, il est donc rarement nécessaire d'utiliser urldecode().

Les caractères à encoder sont les noms de fichiers, les noms des variables et leurs valeurs. Il ne faut pas encoder "?", "=" et "&" s'ils sont utilisés comme séparateurs, mais uniquement en tant que valeurs.

Par exemple, si je souhaite afficher un lien pour une recherche Google à propos de " **Boule & Bill**", les espaces et le "&" doivent être encodés puisqu'ils ne font pas partie de l'alphabet anglais :

```
http://www.google.com/search?hl=fr&q=Boule%20%26%20Bill
```

Voici un autre exemple avec '**E=mc²**", où "=" est converti uniquement lorsqu'il est utilisé comme partie de la question :

```
http://www.google.com/search?hl=en&q=E%3Dmc%C2%B2
```


Ne pas confondre l'encodage en vue d'afficher le lien dans une page HTML, et l'encodage en vue de transmettre une requête GET !

Reprenons l'exemple de "Boule & Bill". Je dois convertir ce texte d'abord selon la RFC 1738 puis en entités HTML :

```
GET /search?hl=fr&q=Boule%20%26%20Bill HTTP/1.1
Host: www.google.com
```

```
<a
    href="http://www.google.com/search?hl=fr&amp;q=Boule%20%26%20Bill"
    title="Recherche Google">Boule &amp; Bill</a>
```

```
<?php
//correspond à l'en-tête HTTP (brute) à envoyer
$url = 'http://www.google.com/search?hl='
```

```
    . rawurlencode('fr') //paramètre à encoder selon la RFC 1738
    . '&q='
    . rawurlencode('Boule & Bill'); //paramètre à encoder selon la RFC 1738

header('Content-Type: text/html; charset=utf-8');
?>
<a
    href="<?php echo html($url);?>"
    title="<?php echo html('Recherche Google'); ?>"
    ><?php echo html('Boule & Bill');?></a>
<?php

function html($string)

{   return utf8_encode(htmlspecialchars($string, ENT_QUOTES));
                    (
}
```

Il faut bien faire la distinction entre toutes les formes du caractère "&"

- **&** est la forme brute, elle porte une signification dans la "query string" finale puisqu'elle sépare les paramètres : c'est un **séparateur** dans l'URL ;
- **&** est la forme encodée en HTML, qui doit être placée dans la page Web afin que le navigateur la comprenne : elle a une signification HTML ;
- **%26** est la forme encodée selon la RFC 1738, elle ne porte aucune signification spéciale ni dans l'URL ni dans le HTML : elle est utilisée seulement en tant que **valeur**.

Cette méthode ne laisse pas de place aux erreurs, les failles de sécurité sont donc bien plus rares. Prenez garde aux séparateurs car ils peuvent avoir une signification différente (ou ne pas en avoir) selon le contexte, comme "&". Dans l'exemple ci-dessus, la chaîne "Boule & Bill" a été encodée de trois manières successives : une pour chaque contexte. L'ordre d'encodage est de type LIFO.

Certains navigateurs affichent ce lien en texte clair (décodé) dans la barre d'état au passage de la souris, mais transmettent malgré tout l'en-tête encodé.

```
http://localhost/test.php?q=forum+cin%C3%A9ma
```

```
<?php
header('text/html; charset=utf-8');
?>
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr-FR" lang="fr-FR">
<head>
    <title>Test</title>
    <meta http-equiv="content-Type" content="text/html; charset=utf-8" />
</head>
<body>
<?php echo htmlentities($_GET['q'], ENT_QUOTES, 'utf-8') ;?>
</body>
</html>
```

```
http://localhost/test.php?q=forum+cin%E9ma
```

```
<?php
header('text/html; charset=iso-8859-1');
?>
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr-FR" lang="fr-FR">
<head>
    <title>Test</title>
    <meta http-equiv="content-Type" content="text/html; charset=iso-8859-1" />
</head>
<body>
<?php echo htmlentities($_GET['q'], ENT_QUOTES, 'iso-8859-1') ;?>
</body>
</html>
```

```
http://localhost/test.php?q=forum+cin%C3%A9ma
```

```
<?php
header('text/html; charset=iso-8859-1');
?>
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr-FR" lang="fr-FR">
<head>
    <title>Test</title>
    <meta http-equiv="content-Type" content="text/html; charset=iso-8859-1" />
</head>
<body>
<?php echo htmlentities($_GET['q'], ENT_QUOTES, 'iso-8859-1') ;?>
</body>
</html>
```

```
http://localhost/test.php?q=forum+cin%E9ma
```

```
<?php
header('text/html; charset=utf-8');
?>
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr-FR" lang="fr-FR">
<head>
    <title>Test</title>
    <meta http-equiv="content-Type" content="text/html; charset=utf-8" />
</head>
```

```
<body>
<?php echo htmlentities ($_GET['q'], ENT_QUOTES, 'utf-8') ;?>
</body>
</html>
```

Si, dans cet exemple, Internet Explorer affiche des caractères lisibles, c'est en fait une erreur de sa part puisque le format du document est incorrect. Internet Explorer estime que le format d'encodage est incorrect, et prend la liberté de l'adapter à sa manière. Lucky guess, comme disent les anglophones. En revanche, Firefox et Opera ont raison de ne pas afficher une chaîne lisible. Il me semble d'ailleurs qu'Internet Explorer 6 affichait un simple "?".

L'un des aspects pris en compte par certains projets pour la sélection du charset, ou plutôt l'un des facteurs qui porte injustement du tort à UTF-8, est la taille de l'encodage des caractères. De nombreuses personnent pensent que, puisqu'UTF-8 est un jeu de caractères multibytes, des documents encodés en UTF-8 prendront nécessairement plus de place qu'en ANSI ou ISO. Cette information n'est pas totalement vraie, et il y a un moyen très simple pour le démontrer.

Le document produit par ce code pèse 36 octets :

```
<?php
header('Content-Type: text/html; charset=ISO-8859-1');
echo htmlspecialchars('Developpez, le Club des développeurs', ENT_QUOTES);
```

Le document produit par ce code pèse 37 octets :

```
<?php
header('Content-Type: text/html; charset=UTF-8');
echo utf8_encode(htmlspecialchars(
    'Developpez, le Club des développeurs',
    ENT_QUOTES));
```

L'exemple ci-dessus montre que tout caractère est encodé le plus bas possible : tous les caractères du jeu ASCII standard sont encodés sur 7 bits ; seul l'accent prend plus d'un octet puisqu'il ne fait pas partie d'ASCII (encodé sur 7 bits) et puisqu'ASCII étendu (encodé sur un octet complet) n'est pas pleinement standardisé ou suivi.

Ainsi, on peut parfaitement encoder intégralement nos documents en UTF-8 sans avoir une taille de fichier excessive. Tous les caractères UTF-8 ne sont pas encodés sur 4 octets, la taille du texte reste donc acceptable. Naturellement, il en va de même pour les bases de données et toutes les autres représentations d'UTF-8.

Un document mal encodé est souvent vulnérable à des failles de sécurité comme XSS. Même dans les situations qui ne présentent pas de danger, un document mal encodé n'est pas valide et cela peut amener le navigateur à réagir de manière inattendue. Les captures d'écran fournies ci-dessus en sont un bon exemple.

Comme nous l'avons vu, le meilleur choix pour présenter des pages Web est d'utiliser un jeu de caractères UTF-8. C'est celui qui offre le panel le plus vaste de caractères, c'est le plus complet des jeux de caractères "multibytes" sans pour autant être très compliqué ou très lourd.

Par ailleurs, nous avons vu qu'il y a plusieurs méthodes pour renseigner le navigateur sur le *charset* d'une page Web :

```
default_mimetype = "text/html"
default_charset = "utf-8"
```

```
<?php
header('Content-Type: text/html; charset=utf-8');
```

```
<meta http-equiv="content-Type" content="text/html; charset=utf-8" />
```

La configuration du *php.ini* semble pratique, mais elle n'est pas nécessairement appropriée si l'on utilise des scripts écrits par d'autres personnes, issus d'un autre projet, *etc*. Il ne faut pas se reposer sur cette configuration puisqu'elle ne correspond pas nécessairement à notre besoin.

Utiliser les deux autres méthodes dans un même document semble redondant, mais il faut prendre en compte certains aspects.

D'une part, transmettre un document HTML depuis le serveur Web vers un navigateur est une chose, et les en-têtes HTTP permettent au navigateur de savoir de quoi il s'agit, mais ils ne persistent pas dans le document une fois qu'il est copié sur une clef USB, dans un e-mail, *etc*. Il est donc intéressant d'avoir l'information du *charset* dans le document lui-même, afin que toutes les copies puissent être restituées fidèlement.

D'autre part, il faut prendre en compte la configuration par défaut du serveur Web : si Apache est configuré pour envoyer un charset ISO alors que notre document est encodé en UFT-8, cela produit un conflit et le navigateur Web peut avoir un comportement imprévisible. Utiliser la fonction **header()** en PHP permet de remplacer l'en-tête envoyé par défaut par Apache, et ainsi de solutionner le problème. Nous ne pouvons pas non plus n'envoyer aucun en-tête par défaut avec Apache, car alors les documents que nous enverrons sans en-tête (sans doute par mégarde ou parce que ce sont des scripts d'une personne tierce qui n'a pas prévu tous ces cas de figure) seront transmis sans aucun en-tête HTTP et sans information sur le type de contenu et l'encodage, ce qui peut avoir un effet désastreux au moment de l'affichage du document.

Il nous faut donc envoyer le type de document et son encodage à la fois au moyen d'en-têtes HTTP et dans une balise META, sans pour autant désactiver la configuration par défaut du serveur Web. Cela fait triple emploi mais c'est la seule solution à toute épreuve.


Utilisez toujours les outils de validation pour vérifier que vos documents sont corrects.


Voici une citation de Rasmus Lerdorf pour terminer : " Ultimately we need to get to Unicode everywhere ".

Le module **iconv** est une extension PHP incluse dans le noyau. Son activation a donc lieu lors de la compilation de PHP.

L'un des problèmes d'**iconv** est son manque de fonctionnalités. L'extension **mbstring** est bien plus intéressante à ce niveau.

```
http://localhost/test.php?string=Le%20club%20des%20d%E9veloppeurs
```

```php
<?php
header('Content-Type: text/html; charset=utf-8');
echo empty($_GET['string']) ?
    '' :
    iconv('ISO-8859-1', 'UTF-8', $_GET['string']);
```

```
Le club des développeurs
```

Ici par exemple, il nous faut savoir avec exactitude le *charset* des données en entrée. La moindre erreur est ennuyeuse :

```
Notice: iconv() [function.iconv]: Detected an illegal character
    in input string in C:\Web\online\http\test.php on line 2
```

Voici une solution plus élégante mais qui nécessite **mbstring** :

```
mbstring.detect_order = ISO-8859-1, UTF-8
```

```
http://localhost/test.php?string=Le%20club%20des%20d%E9veloppeurs
```

```php
<?php
header('Content-Type: text/html; charset=utf-8');
echo empty($_GET['string']) ?
    '' :
    iconv(mb_detect_encoding($_GET['string']), 'UTF-8', $_GET['string']);
```

```
Le club des développeurs
```

Bien entendu, il est toujours possible d'utiliser une fonction pour simplifier la procédure :

```php
<?php
header('Content-Type: text/html; charset=utf-8');
echo empty($_GET['string']) ? '' : to_utf8($_GET['string']);

function to_utf8($string)

{   return iconv(mb_detect_encoding($string), 'UTF-8', $string);
}
```

Pour conclure sur **iconv**, le module me semble peu intéressant, comparé à par exemple l'extension **mbstring**. En effet, il contraint le développeur soit à utiliser une autre extension pour déterminer le charset des variables (ce qui est absurde, dans la mesure où cette autre extension peut fonctionner seule), soit à savoir exactement de quel encodage est chaque variable (ce qui est totalement utopique dans un environnement Web). Il est possible d'utiliser les headers HTTP envoyés par le navigateur client, mais que se passerait-il si un navigateur malicieux cherchait à faire passer une chaîne UTF-8 pour de l'ISO-8859-1 ? Je crains que cela ouvre l'application à des failles de sécurité...

Cela dit, certaines fonctions d'**iconv** ont leur utilité, par exemple **iconv_mime_decode_headers()**

.

Jusqu'à l'arrivée de PHP 6, le langage PHP n'est pas prévu pour fonctionner nativement avec Unicode. Toutes les opérations classiques sur les chaînes, par exemple **strpos()** et **substr()**, ne fonctionnent pas avec les jeux de caractères "multibyte" tel que le japonais. Aucun alphabet non latin ne fonctionne correctement avec ces fonctions.

La bibliothèque **mbstring** permet de remédier à ce problème. Elle fournit des fonctions alternatives pour les tâches les plus courantes, par exemple mb_strlen() à la place de strlen() ou encore mb_strpos() au lieu de strpos(). Toutes les opérations sur les chaînes ne sont cependant pas disponibles, par exemple l'extension PCRE n'a pas été adaptée.

Voici un exemple de script qui ne fonctionne pas comme on pourrait s'y attendre :

Imaginez que je veuille utiliser **substr($string, 0, 2)**, est-ce que j'obtiendrais les 2 premiers caractères ou bien les 2/3 du premier caractère ? <insert smiley here>

En revanche, la bibliothèque mbstring permet de préciser l'encodage interne des caractères et d'en tenir compte :

Cette fois, si j'utilise **mb_substr($string, 0, 2)**, je suis sûr d'obtenir les 2 premiers caractères du mot.

La plupart des applications Web développées pour nos pays n'ont pas besoin de fonctions de texte aussi complexes. Vous ne devriez considérer l'utilisation de cette extension que si vous en avez réellement besoin, ou peut-être attendre PHP 6 pour uniformiser le comportement des chaînes Unicode.

NB : Les caractères utilisés ci-dessus sont la traduction japonaise du mot "japonais" selon Wikipédia.

Voici un exemple de conversion de chaînes d'entrée :

```php
<?php
mb_internal_encoding('UTF-8');

$variable_1 = to_utf8($variable_1);
$variable_2 = to_utf8($variable_2);
$variable_3 = to_utf8($variable_3);

function to_utf8($string)

{   return mb_convert_variables('UTF-8', mb_detect_encoding($string), $string);

}
```

Vous pouvez aussi utiliser cette autre méthode, moins fastidieuse, qui convertit automatiquement toutes les variables d'entrée :

```
[mbstring]
mbstring.internal_encoding = UTF-8
mbstring.http_output = UTF-8
mbstring.encoding_translation = On
mbstring.detect_order = auto
mbstring.substitute_character = none
mbstring.strict_encoding = On
```

Je ne compte pas entrer ici dans le détail des regex, puisque c'est un sujet trop vaste pour être abordé convenablement en quelques lignes. Cependant, un aspect de PCRE est souvent omis : les fonctions **preg_*()** peuvent gérer les caractères multibytes à condition de les écrire sous leur forme hexadécimale et d'utiliser le modificateur "u".

Par exemple, pour vérifier qu'un texte fait partie de l'un des trois systèmes d'écriture japonaise, on peut utiliser la regex suivante :

```php
<?php
if (preg_match('/[\x{4e00}-\x{9fbf}]/u', $variable))
{
    echo 'Kanji';
}
else if (preg_match('/[\x{3040}-\x{309f}]/u', $variable))
{
    echo 'Hiragana';
}
else if (preg_match('/[\x{30a0}-\x{30ff}]/u', $variable))
{
    echo 'Katagana';
}
else
{
    echo 'Pas un mot japonais';
}
```

Une chaîne UTF-8 invalide arrête la regex sans rien trouver. Avant d'utiliser ces masques, il est donc nécessaire de suivre quelques conseils, cf. ce commentaire de la documentation .

Dans un environnement Web, il y a trois sources principales de données : une base de données relationnelle ou objet, des fichiers XML et des services Web.

Les bases de données permettent d'emmagasiner et de retrouver de manière simple et efficace de très grandes quantités d'informations de tous types : numérique, texte, multimédia...

Pour accéder à une base de données, il faut généralement disposer d'un code d'accès, de l'adresse IP du serveur et du nom de la base de données sur le serveur. Une dernière information nous indique le type de la BDD qui nous servira pour lancer nos requêtes (MySQL, PostgreSQL, Oracle...).

La plupart des APIs fonctionnent sur un principe procédural, orienté objet ou bien les deux. Nous utiliserons de préférence la POO pour la flexibilité qu'elle nous offre.

Je vais partir du principe que vous connaissez les bases de données relationnelles et le langage SQL.

L'accès à une BDD peut se faire soit au moyen d'une API spécialisée pour votre BDD (MySQL, PostgreSQL, Oracle...), soit à travers une API générique (aussi appelée **couche d'abstraction**). Puisque les couches d'abstraction laissent davantage de liberté et de flexibilité que les APIs, il n'y a aucun avantage à utiliser une API spécifique. En revanche, le plus gros avantage d'utiliser une telle couche d'abstraction est de donner le choix du SGBD avec un minimum de modifications sur le code de l'application PHP.

Je vous propose donc d'utiliser **PHP Data Objects** (alias **PDO**), qui est la couche d'abstraction fournie en standard dans PHP.

Extensions PHP nécessaires : **php_pdo** + **php_pdo_mysql** (cette deuxième extension correspond à votre SGBD, dans mon cas MySQL).

L'avantage principal de PDO est l'extrême simplicité de changement de SGBD : il suffit de changer un paramètre lors de l'instanciation de PDO, le reste du code est inchangé. Cela permet par exemple de coder une application et de la vendre à n'importe quel client, quel que soit son SGBD de prédilection, sans surcharge de travail. Par conséquent, il n'est pas nécessaire d'apprendre le fonctionnement de chaque API spécifique, PDO est suffisant. Vos développements prennent donc moins de temps.

Des exemples de chaînes de connexion se trouvent dans la FAQ PHP , et d'autres tutoriels sont disponibles ici : Cours SGBD en PHP .

L'accès à une base de données est soumis à diverses étapes :

- Ouverture de la connexion au serveur ;
- Choix de la BDD sur le serveur (cela se fait parfois au moment de l'ouverture de la connexion) ;
- Préparation de requêtes (facultatif, dépend de l'API utilisée) ;
- Exécution de requêtes et récupération des résultats.
-

L'envoi de requêtes implique souvent le parcours des résultats, et dans certains cas la préparation préalable des requêtes (c'est le cas de PDO).

La connexion au serveur et la sélection de la BDD se font lors de l'instanciation de l'objet PDO :

```
$db = new PDO('mysql:host=localhost;dbname=developpez', 'utilisateur', 'motdepasse');
```

Pour envoyer une requête, il y a plusieurs solutions. La plus sécurisée est de passer par des requêtes préparées (*prepared statements*) :

```
$db = new PDO('mysql:host=localhost;dbname=developpez', 'utilisateur', 'motdepasse');

$select_users = $db->prepare('SELECT id, name FROM user');
$select_user = $db->prepare('SELECT id, name FROM user WHERE user_id = ?');
$insert_user = $db->prepare('INSERT INTO user (name, password) VALUES (?, ?)');

//récupère tous les utilisateurs (aucun puisque la table est vide)
$select_users->execute();
$users = $select_users->fetchAll();

$insert_user->execute(array('BrYs', '4321'));
$insert_user->execute(array('mathieu', '4321'));
$insert_user->execute(array('Yogui', '4321'));

$select_users->execute(); //récupère tous les utilisateurs (les 3)
$users = $select_users->fetchAll();

$select_user->execute(array(1)); //récupère l'utilisateur numéro 1
$user_1 = $select_user->fetchAll();

$select_user->execute(array(2)); //récupère l'utilisateur numéro 2
$user_2 = $select_user->fetchAll();

$select_user->execute(array(3)); //récupère l'utilisateur numéro 3
$user_3 = $select_user->fetchAll();
```

Comme nous pouvons le voir, une requête se prépare une seule fois pour l'ensemble du script. Chaque exécution de la requête se fait alors par la méthode **execute()**, et la récupération des résultats par la méthode **fetchAll()**.

Bien sûr, il est possible d'envoyer des requêtes directement depuis l'objet **$db**, mais alors il n'y a aucune forme de sécurisation des données puisque la méthode **quote()** n'est pas disponible pour tous les pilotes de BDD. Un code qui envoie des requêtes non préparées n'est donc pas **portable**.

Il y a deux avantages majeurs à préparer les requêtes :

- Chaque exécution est plus rapide en requête préparée qu'en envoyant la totalité de la requête à chaque fois ;
- Les paramètres sont transmis à la requête sous forme binaire, ce qui évite tout risque d'injection SQL.

Les paramètres peuvent être transmis à la requête préparée soit dans l'ordre, soit sous forme nommée. Les paramètres nommés peuvent être envoyés dans n'importe quel ordre :

```
<?php
$db = new PDO(
    'mysql:host=localhost;dbname=developpez',
    'utilisateur',
    'motdepasse');

$insert_1 = $db->prepare(
    'INSERT INTO user (name, password) VALUES (?, ?)');

$insert_2 = $db->prepare(
```

```
    'INSERT INTO user (name, password) VALUES (:name, :password)');

$insert_1->execute(array('BrYs', '4321'));
$insert_2->execute(array
    ':password' => '4321',
    ':name' => 'BrYs'
));
```

Dans une architecture classique, le serveur HTTP et le SGBD sont deux machines différentes sur un même réseau local. Exactement combien de machines entrent en jeu n'est pas notre propos, mais cela nous indique le protocole de communication habituellement utilisé pour le dialogue entre les deux applications (par exemple entre Apache et MySQL) : TCP/IP. Toutes les requêtes et leurs résultats transitent par la couche réseau. Il est donc plutôt facile de congestionner un réseau avec des requêtes renvoyant de nombreux résultats, surtout si elles sont fréquemment exécutées. C'est l'une des raisons en défaveur de la pratique "SELECT *", puisqu'elle retourne plus de champs qu'il n'est nécessaire, et fait donc transiter un gand nombre d'informations inutiles par le réseau.

C'est aussi à cause de ces échanges qu'il est important de réduire le nombre de requêtes exécutées à chaque page, sans pour autant que cela devienne une guerre sainte... Chaque exécution de requête implique d'attendre la réponse du SGBD avant de pouvoir poursuivre l'exécution du script, ce qui correspond à un aller-retour TCP/IP entre le serveur HTTP et le SGBD. La communication entre deux machines par un réseau local est certes rapide, mais c'est sans commune mesure avec des optimisations de type " *for plutôt que foreach* ". Soyez conscients que ce qui ralentit le plus votre application est sans doute davantage le temps nécessaire pour faire un aller-retour de requête et de son résultat, que des optimisations in-line de votre code PHP. J'ai entendu **Tim Bray** rappeler aux développeurs qu'un système de fichiers est souvent plus rapide qu'un SGBD : évaluez votre besoin, comparez ce que vous apporte chaque SGBD. SQLite est parfois un bon compromis.

Le principe des requêtes préparées permet d'enregistrer temporairement chaque requête du côté du SGBD. Ainsi, en plus des optimisations que cela permet d'obtenir côté serveur, pour exécuter chaque requête il suffit de renvoyer les paramètres (au lieu de l'ensemble de la requête). Avec l'avantage sécurité que nous avons déjà mentionné, cela nous donne une longue liste d'avantages en faveur des requêtes préparées...

À des fins d'illustration, voici un exemple (à ne pas suivre) utilisant une base de données de forum avec une clef étrangère entre **message.user_id** et **user.id** :

```
<?php
mysql_connect('localhost', 'utilisateur', 'motdepasse');
mysql_select_db('developpez');

$sql = 'SELECT id, title, user_id
        FROM message
        ORDER BY post_date';
$db_messages = mysql_query($sql);

header('Content-Type: text/html; charset=utf-8');
while($message = mysql_fetch_array($db_messages))

{   $sql = 'SELECT name
            FROM user
            WHERE id = '.(int)$message['user_id'];

    $db_user = mysql_query($sql);
    $user = mysql_fetch_array($db_user);

    echo utf8_encode
        htmlspecialchars($message['title'].' par '.$user['name'], ENT_QUOTES))
        .'<br/>';

}
```

Voici les erreurs du script ci-dessus :

- Utilisation de l'API MySQL plutôt que **PDO** ou même **MySQLi**, ce qui est un mauvais choix par rapport aux performances et à la portabilité du code ;
- Utilisation de ***fetch_array**, qui retourne un tableau de résultats à la fois sous forme d'index numérique et d'index associatif, ce qui consomme inutilement de la mémoire ;
- Exécution de nombreuses requêtes alors qu'**INNER JOIN** aurait été plus efficace.

Le script suivant donne le même résultat :

```php
<?php
$db = new PDO(
    'mysql:host=localhost;dbname=developpez',
    'utilisateur',
    'motdepasse');

$sql = 'SELECT m.id, m.title, u.name AS author
        FROM message AS m
        INNER JOIN user AS u ON m.user_id = u.id
        ORDER BY post_date';

$select_messages = $db->prepare($sql);
$select_messages->setFetchMode(PDO::FETCH_ASSOC);
$select_messages->execute();

header('Content-Type: text/html; charset=utf-8');
foreach ($select_messages->fetchAll() as $message)
{   echo utf8_encode(htmlspecialchars(
        $message['title'].' par '.$message['author'], ENT_QUOTES))
        .'<br/>';

}
```

La POO autorisant le principe d'héritage, autant en profiter. Dans l'exemple ci-dessus, on peut voir que les sélections utilisent deux méthodes à la suite (**execute()** puis **fetchAll()**) : cela peut être simplifié par l'héritage. Il faut également prendre en compte la configuration de la classe : chaque connexion dispose d'un certain nombre d'attributs, et PDO peut être configuré pour envoyer des exceptions plutôt que des erreurs PHP. Tout cela peut être fait dans une classe dérivée de PDO, afin de nous assurer que tous nos objets **$db** en bénéficient dans tous nos projets.

Nous aimerions avoir du code applicatif simple et concis, par exemple :

```php
<?php
$db = new MyPDO(
    'mysql:host=localhost;dbname=developpez', 'utilisateur', 'motdepasse');

$userTable = new UserTable($db);

$userTable->truncate();
$userTable->insert('Yogui', '4321');
$userTable->insert('mathieu', '4321');
$userTable->insert('BrYs', '4321');

?><pre><?php
print_r ($userTable->selectById(1));
print_r ($userTable->selectByName('u'));
print_r ($userTable->selectAll());
```

Pour y parvenir, nous avons besoin d'une classe "UserTable" qui représente la table du même nom :

```php
<?php
class UserTable
```

```
{   /*
     * Requêtes préparées
     */
    private $selectById;
    private $selectByName;
    private $selectAll;
    private $insert;
    private $truncate;

    public function __construct($db)
    {
        $this->selectAll = $db->prepare(
            "SELECT id, name FROM user ORDER BY name, id");

        $this->selectById = $db->prepare(
            "SELECT id, name FROM user WHERE id = ?");

        $this->selectByName = $db->prepare(
            "SELECT id, name FROM user WHERE name LIKE ? ORDER BY name, id");

        $this->insert = $db->prepare(
            "INSERT INTO user (name, password) VALUES (:name, :password)");

        $this->truncate = $db->prepare("TRUNCATE user");
    }

    public function insert($name, $password)
    {
        $this->insert->execute(
            array(':name' => $name, ':password' => $password));

        return $this->insert->rowCount();
    }

    public function selectAll()
    {
        $this->selectAll->execute();
        return $this->selectAll->fetchAll();
    }

    public function selectById($id)
    {
        $this->selectById->execute(array($id));
        return $this->selectById->fetch();
    }

    public function selectByName($name)
    {
        $this->selectByName->execute(array('%'.$name.'%'));
        return $this->selectByName->fetchAll();
    }

    public function truncate()
    {
        return $this->truncate->execute();
    }

}
```

Tout cela repose sur notre classe dérivée de PDO, qui configure la connexion ainsi que la manière de préparer les requêtes :

```
<?php
class MyPDO extends PDO

{   public function __construct($dsn, $user=NULL, $password=NULL
    {                                                          )
        parent::__construct($dsn, $user, $password);
```

```
        $this->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
    }

    public function prepare($sql, $options=NULL
    {                                       )
        $statement = parent::prepare($sql);
        if strpos strtoupper($sql), 'SELECT') === 0) //requête "SELECT"
        { (     (
            $statement->setFetchMode(PDO::FETCH_ASSOC);
        }

        return $statement;
    }

}
```

Résultat :

```
Array

(   [id] => 1
    [name] => Yogui

Array

(   [0] => Array
        (
            [id] => 2
            [name] => mathieu
        )

    [1] => Array
        (
            [id] => 1
            [name] => Yogui
        )


Array

(   [0] => Array
        (
            [id] => 3
            [name] => BrYs
        )

    [1] => Array
        (
            [id] => 2
            [name] => mathieu
        )

    [2] => Array
        (
            [id] => 1
            [name] => Yogui
        )


)
```


Maintenant est un bon moment pour vous remémorer mon exemple de code lors de l'explication des Late Static Bindings, et pour faire des essais de votre côté.

Tous les frameworks PHP proposent une solution bien mieux adaptée que celle que je présente ici. Par exemple, symfony génère deux classes pour chaque table de la BDD : l'une contient uniquement des méthodes statiques d'accès aux données au niveau de la table, et l'autre est la représentation objet d'un seul tuple

de la table. Zend Framework, quant à lui, opte pour deux classes à dériver : Zend_Db_Table_Abstract et Zend_Db_Table_Row_Abstract.

La base de données étant une source de données externe au script, les données qui en proviennent doivent être traitées de la même manière que les données utilisateur : avec prudence. Il faut valider et filtrer les données issues de la BDD.

Pour information, voici le code applicatif ci-dessus écrit avec l'API de base de PHP pour MySQL. Le majeur problème de cette approche est qu'elle est spécifique à MySQL et que nous obligeons ainsi tous nos clients à utiliser MySQL, même s'ils avaient déjà un SGBD en lequel ils avaient confiance (par exemple PostgreSQL ou Oracle). Changer de SGBD dans nos scripts nous oblige à effectuer de nombreuses modifications : PDO permet d'éviter ce genre de problèmes.

```php
<?php
mysql_connect('localhost', 'utilisateur', 'motdepasse');
mysql_select_db('developpez');

db_user_truncate();
db_user_insert('Yogui', '4321');
db_user_insert('mathieu', '4321');
db_user_insert('BrYs', '4321');

?><pre><?php
print_r(db_user_select_by_id(1));
print_r(db_user_select_by_name('u'));
print_r(db_user_select_all());
```

```php
<?php
function db_user_insert($name, $password)

{   $sql = "INSERT INTO user(name, password) VALUES('%s', '%s')";
    mysql_querysprintf
        $sql,  (       (
        mysql_real_escape_string($name),
        mysql_real_escape_string($password)));

    return mysql_affected_rows();

}
function db_user_select_by_id($id)

{   $sql = "SELECT id, name FROM user WHERE id = %d";
    $db_result = mysql_querysprintf($sql, $id));
    if ($user = mysql_fetch_assoc($db_result))
    {
        return $user;
    }
    else
    {
        return NULL
    }             ;

}
function db_user_select_by_name($name)

{   $sql = "SELECT id, name FROM user WHERE name LIKE '%s' ORDER BY name";
    $db_result = mysql_querysprintf
        $sql,                (       (
        mysql_real_escape_string('%'.$name.'%')));

    $users = array();
    while($user = mysql_fetch_assoc($db_result))
    {
        $users[] = $user;
    }
    return $users;

}
```

```php
function db_user_select_all()

{   $sql = "SELECT id, name FROM user ORDER BY name";
    $db_result = mysql_query($sql);
    $users = array();
    while($user = mysql_fetch_assoc($db_result))
    {
        $users[] = $user;
    }
    return $users;

}
function db_user_truncate()

{   mysql_query("TRUNCATE user");

}
```

D'autres inconvénients de l'approche par API spécifique est que ces API ne disposent pas toujours de méthodes pour protéger les requêtes, comme ici **mysql_real_escape_string()**. Par ailleurs, il est très facile d'oublier d'utiliser ces fonctions ou de transtyper la valeur. Cela peut très facilement conduire à des failles de sécurité dans les applications.

Il est donc indéniablement préférable d'utiliser des requêtes préparées.

Extensions nécessaires : aucune, tout est inclus en standard dans PHP.

Les fichiers XML sont très utiles pour faire transiter les informations entre deux bases de données, par e-mail, *etc* Naturellement, PHP fournit des classes permettant de faciliter leur manipulation. .

Il existe deux méthodes d'accès à un fichier XML :

- **SAX** : Lecture ou écriture séquentielle, comme un flux de données, au fur et à mesure de l'avancement du pointeur de fichier ; cette approche est avantageuse pour la faible quantité de mémoire utilisée, puisque chaque noeud est libéré de la mémoire après utilisation, mais elle est très complexe à utiliser ;
- **DOM** : Lecture de l'arbre complet du XML lors de l'ouverture du document (ou manipulation par hiérarchie dans le cas d'écriture) ; cette approche est plus simple mais plus gourmande en mémoire, et n'est pas adaptée pour les grands fichiers.

Nous allons nous concentrer sur une approche DOM, suffisante pour débuter et dans de très nombreuses situations.

SimpleXML est l'API la plus simple pour la lecture de documents XML. La lecture du fichier se fait selon une approche DOM, et on peut facilement filtrer les données par des requêtes XPath.

Prenons l'exemple d'une BDD de messages. Le fichier XML suivant peut être le dump d'une table *message* par exemple, et la table *user* peut être dans un autre fichier XML.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<messages>
    <message>
```

```
        <id>1</id>
        <user_id>1</user_id>
        <title>Bonjour</title>
        <body>Un bonjour de Paris ;)</body>
    </message>
    <message>
        <id>2</id>
        <user_id>2</user_id>
        <title>Super site</title>
        <body>J'apprécie ton site, continue ainsi !</body>
    </message>
    <message>
        <id>3</id>
        <user_id>2</user_id>
        <title>Merci pour tout</title>
        <body>J'oubliais de dire que ton site m'a beaucoup servi !</body>
    </message>
</messages>
```

```
<?php
//lecture du XML dans un objet PHP
$messages = simplexml_load_file('messages.xml');

foreach($messages as $message) //parcours du XML comme d'un tableau

{   $id = (int)$message->id;

    //pas de validation, le message doit être affiché tel quel
    $body = $message->body;
    echo html($id.' - '.$body).'<br/>';

}
function html($string)

{   return utf8_encode(htmlspecialchars($string, ENT_QUOTES));
                    (
}
```

```
1 - Un bonjour de Paris ;)
2 - J'apprécie ton site, continue ainsi !
3 - J'oubliais de dire que ton site m'a beaucoup servi !
```

Notez que j'ai pris soin de filtrer le texte (décoder l'UTF-8) lorsque je le récupère : cela m'évite d'avoir des problèmes d'encodage. Notez également que l'on peut utiliser **foreach** sur un objet **SimpleXMLElement** : c'est parce qu'il implémente l'interface **ArrayAccess** issu de la **SPL**

.


Un fichier XML est à considérer comme toute autre source de données externes au script : il faut filtrer les valeurs qui en proviennent.

Admettons maintenant que nous ayons besoin d'écrire le fichier XML ci-dessus à partir de notre BDD. Voici le SQL pour la mise en place de la table et de son contenu :

```
CREATE TABLE message (
  id int(12) NOT NULL auto_increment,
  user_id int(12) default NULL,
  title varchar(50) default NULL,
  body varchar(1000) default NULL,
  PRIMARY KEY (id)
);

INSERT INTO message (user_id, title, body)
```

```
VALUES (
    1,
    "Bonjour",
    "Un bonjour de Paris ;)");

INSERT INTO message (user_id, title, body)
VALUES (
    2,
    "Super site",
    "J'apprécie ton site, continue ainsi !");

INSERT INTO message (user_id, title, body)
VALUES (
    2,
    "Merci pour tout",
    "J'oubliais de dire que ton site m'a beaucoup servi !");
```

Nous aurons besoin de notre classe MyPDO et d'une classe MessageTable :

```
<?php
class MessageTable

{   private $select;

    public function __construct($db)
    {
        $this->select = $db->prepare(
            "SELECT id, user_id, title, body FROM message");
    }

    public function selectAll()
    {
        $this->select->execute();
        return $this->select->fetchAll();
    }

}
```

Ajoutons à cela une classe permettant de simplifier la création du XML :

```
<?php
class MyDOMDocument extends DOMDocument

{   public function __construct($version=NULL, $charset=NULL
    {                                                      )
        parent::__construct($version, $charset);
        $this->strictErrorChecking = TRUE
        $this->appendChild($this->createElement('messages'));
        $this->formatOutput = TRUE
    }                           ;

    public function appendMessage($id, $user_id, $title, $body)
    {
        $message = $this->createElement('message');
        $this->documentElement->appendChild($message);

        $message->appendChild(
            $this->createElement('id', (int)$id));

        $message->appendChild(
            $this->createElement('user_id', (int)$user_id));

        $message->appendChild(
            $this->createElement('title', $this->escapeText($title)));

        $message->appendChild(
            $this->createElement('body', $this->escapeText($body)));
```

```
    }

    protected function escapeText($string)
    {
        return utf8_encode(htmlspecialchars($string));
    }
}
```

Et enfin notre script principal :

```
<?php
$db = new MyPDO(
    'mysql:host=localhost;dbname=developpez', 'utilisateur', 'motdepasse');

$xml = new MyDOMDocument('1.0', 'utf-8');
$messageTable = new MessageTable($db);

foreach ($messageTable->selectAll() as $message)
{
    $id = (int)$message['id'];
    $user_id = (int)$message['user_id'];
    $body = preg_replace('/[[:cntrl:]]/', '', $message['body']);

    if (ctype_print($message['title']))
    {
        $title = $message['title'];
    }
    else
    {
        $title = preg_replace('/[^[:graph:]]/', '', $message['title']);
    }

    $xml->appendMessage($id, $user_id, $title, $body);

}
header('Content-Type: application/xml; charset=utf-8');
echo $xml->saveXML();
```

Notez comme je filtre les données en provenance de la BDD dans l'appel à appendMessage() . De plus, je les convertis selon leur format de destination lors de l'appel à createElement() .

L'objet $messageTable n'ayant aucune raison d'apparaître plusieurs fois dans notre code, on peut mettre en place un système pour éviter la création de multiples instances de la classe MessageTable . Il s'agit du design pattern "Singleton", que nous aborderons plus loin.

Les services Web

Les **services Web** sont un moyen de mettre un service (c'est-à-dire des informations) à disposition par l'intermédiaire d'un réseau. Appeler un Web service est la même chose qu'appeler une fonction dans un programme, mais au travers d'un réseau informatique (souvent Internet).

De nombreux services existent sur le Web, et tous n'utilisent pas le même protocole de communication. Certains utilisent un protocole REST, d'autres RPC, d'autres encore **SOAP**. Chaque utilisation dépend des méthodes et paramètres définis par l'API du Web service auquel vous faites appel.

Les différents types de services :

- **SOAP** utilise XML pour formater les messages à transmettre, ce qui se fait généralement par HTTP : une URL peut correspondre à plusieurs fonctions ;
- **RPC** utilise XML ou JSON pour formater les messages à transmettre, ce qui se fait généralement par des sockets : une URL peut correspondre à plusieurs fonctions;
- **REST** utilise le protocole HTTP pour transmettre ses messages : l'URL correspond à la fonction appelée et la méthode HTTP à l'action à mener, le message lui-même étant formaté en fonction des en-têtes HTTP qui l'accompagnent.

On a **beaucoup parlé** de l'affrontement entre les standards du W3C (**WS-***) et **REST**. Tim Bray lui-même **ne semblait pas emballé** par WS-* lors de la Conférence Internationale 2006, tandis que REST lui semblait plus séduisant.

Outre le format du message à transmettre par le service, les services Web SOAP et RPC utilisent un format XML pour décrire leur fonctionnement : le Web Services Description Language (**WSDL**).

L'action d'utiliser un service Web s'appelle la "consommation".

Un service Web peut être consommé par n'importe quelle API compatible. Nous allons créér ici des services en PHP et nous allons les consommer en PHP, mais nous pouvons aussi les consommer à l'aide d'un autre langage ; ou bien consommer en PHP un service Web écrit dans un autre langage. C'est aussi à cela que sert le WSDL.

Une fois le WSDL généré, le service Web est exposé au monde entier. À partir de ce moment-là, il ne faut plus modifier la signature des fonctions ou le protocole de communication, le WSDL ne doit donc plus évoluer. Si un jour nous devions nous passer de PHP, la nouvelle application devrait faire en sorte de correspondre exactement à ce WSDL afin que les consommateurs actuels du service ne soient pas perturbés. Ne pas générer de WSDL pose donc des problèmes de pérennité du service Web.

SOAP est un protocole de communication entre applications. Le format XML est utilisé pour les messages, ce qui permet une grande interopérabilité. L'organisme en charge des spécifications de SOAP est le W3C.

Ce protocole utilise le format WSDL pour décrire et pour interpréter la description d'un service Web.

PHP étant doté de variables à type dynamique, il n'a pas été jugé nécessaire de requérir un fichier WSDL. Le fichier WSDL est donc facultatif mais souvenez-vous que d'autres technologies peuvent avoir besoin d'un tel fichier pour être en mesure de consommer votre service...

L'extension **php_soap** est apparue avec PHP 5. Il faut donc l'activer dans *php.ini* avant de poursuivre ce tutoriel. Attention, certaines directives de configuration ne sont apparues qu'avec PHP 5.1.5 :

Options de configuration :

- **soap.wsdl_cache_enabled** : Activation du cache local du WSDL ;
- **soap.wsdl_cache_dir** : Chemin local (répertoire) du cache ;
- **soap.wsdl_cache_ttl** : Temps de validité du cache ;
- **soap.wsdl_cache_limit** : Nombre maximal de WSDL en cache local ;
- **soap.wsdl_cache** : Type du cache.

Voici un exemple de service d'inversement de texte :

```php
<?php
$server = new SoapServer(NULL, array('uri' => 'http://Reversing'));
$server->addFunction('reverse');
$server->handle();

function reverse($string)

{   return utf8_encode strrev utf8_decode($string)));
                    (      (
}
```

```php
<?php
$service = new SoapClient(NULL, array
    'location' => 'http://tests.localhost/soap/services/reverse.php',
    'uri' => 'http://Reversing'
));

echo $service->reverse('Yogui').'<br/>';
echo $service->reverse('BrYs').'<br/>';
echo $service->reverse('mathieu');


function html($string)

{   return htmlspecialchars($string, ENT_QUOTES);

}
```

```
http://localhost/tests/soap/
```

```
iugoY
sYrB
ueihtam
```

Les premiers services Web étaient du style RPC, une approche très similaire à un appel de fonction. RPC fonctionne avec des sockets.

Les bibliothèques et les frameworks récents continuent de proposer des méthodes facilitant l'utilisation de services Web RPC.

Voici un exemple d'exposition de service avec Zend Framework (en supposant que ZendFramework soit ajouté à **include_path** dans votre configuration) :

```php
<?php
require_once 'Zend/XmlRpc/Server.php';

/**
 * Returns the reverse of a string
 *
 * @param string $string Value to reverse
 * @return string reverse of value
 */
function reverse($string)

{   return utf8_encode strrev utf8_decode($string)));
                    (      (
}
$server = new Zend_XmlRpc_Server();
$server->addFunction('reverse');
```

```
echo $server->handle();
```

Zend Framework n'a pas besoin de fichier WSDL car il utilise des annotations dans le code pour exposer le service Web. C'est d'ailleurs cette tendance des services RPC à implémenter le service trop proche du langage, qui leur vaut une mauvaise réputation aujourd'hui.

Voici maintenant un exemple de consommation de ce service :

```
<?php
require_once 'Zend/XmlRpc/Client.php';

$consumer = new Zend_XmlRpc_Client(
    'http://localhost/tests/xmlrpc/services/reverse.php');

$service = $consumer->getProxy();

header('Content-Type: text/html; charset=utf-8');
echo html($service->reverse('Yogui')).'<br/>';
echo html($service->reverse('BrYs')).'<br/>';
echo html($service->reverse('mathieu'));


function html($string)

{   return htmlspecialchars($string, ENT_QUOTES);

}
```

```
http://localhost/tests/xmlrpc/
```

```
iugoY
sYrB
ueihtam
```

Contrairement aux scripts de Zend Framework, index.php et services/reverse.php doivent être exposés sous DocumentRoot.

Pour qu'un service Web RPC soit disponible, il faut pouvoir ouvrir un socket sur le serveur. Si le WS est très utilisé, cela peut avoir des conséquences sur la charge de votre serveur.

En PHP, ce type de service est implémenté à l'aide de **Service Component Architecture** (SCA) et il nécessite un fichier WSDL. La **page du projet** (IBM.com) contient toute la documentation nécessaire. Le projet OpenSOA pour PHP est composé de SCA et SDO, regroupés dans une extension et plusieurs scripts. Pour mettre en place des services Web SOA en PHP, il faut maîtriser SCA ; on peut également utiliser SDO, pratique pour utiliser diverses sources de données de la même manière, mais ce n'est pas obligatoire.

SCA se charge de générer le fichier WSDL à partir des commentaires présents dans le code source des classes PHP du service. Une valeur de retour SCA est toujours l'un des quatre types scalaires de PHP : booléen, entier, réel ou chaîne.

Pour installer SCA_SDO :

- Placez l'extension SDO dans le répertoire des extensions PHP ;

- Activez les extensions SDO, CURL et SOAP dans *php.ini* ;
- Copiez le contenu du répertoire des sources du projet SCA_SDO dans un répertoire accessible depuis **include_path** mais situé à l'extérieur de *DocumentRoot* ;
- Vérifiez que PHP a les permissions nécessaires pour écrire dans le dossier que vous utiliserez pour les classes PHP de vos services (dans mon cas "*C:\Web\online\http\tests\sca\services*").

Voici un exemple de service d'inversement de texte (le même que l'exemple RPC ci-dessus) :

```php
<?php
include 'SCA/SCA.php';

/**
 * @service
 * @binding.soap
 */
class Reversing

{   /**
     * @param string $string
     * @return string
     */
    public function reverse($string)
    {
        return utf8_encode strrev utf8_decode($string)));
    }                    (      (

}
```

```php
<?php
$wsdl = 'C:\Web\online\http\tests\sca\services\Reversing.wsdl';
if (!file_exists ($wsdl))

{   file_put_contents
        $wsdl,       (
        file_get_contents
            'http://localhost/tests/sca/services/Reversing.php?wsdl')
    );

}
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
  xmlns:tns2="http://Reversing"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  targetNamespace="http://Reversing">
  <types>
    <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
      targetNamespace="http://Reversing"
      elementFormDefault="qualified">
      <xs:element name="reverse">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="name" type="xs:string"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="reverseResponse">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="reverseReturn" type="xs:string"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:schema>
  </types>
```

```
  <wsdl:message name="reverseRequest">
    <wsdl:part name="reverseRequest" element="tns2:reverse"/>
  </wsdl:message>
  <wsdl:message name="reverseResponse">
    <wsdl:part name="return" element="tns2:reverseResponse"/>
  </wsdl:message>
  <wsdl:portType name="ReversingPortType">
    <wsdl:operation name="reverse">
      <wsdl:input message="tns2:reverseRequest"/>
      <wsdl:output message="tns2:reverseResponse"/>
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="ReversingBinding" type="tns2:ReversingPortType">
    <soap:binding
        transport="http://schemas.xmlsoap.org/soap/http"
        style="document"/>
    <wsdl:operation name="reverse">
      <soap:operation soapAction=""/>
      <wsdl:input>
        <soap:body use="literal"/>
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="ReversingService">
    <wsdl:port name="ReversingPort" binding="tns2:ReversingBinding">
      <soap:address
        location="http://localhost/tests/sca/services/Reversing.php"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>

<!-- this line identifies this file as WSDL generated by SCA for PHP. Do not remove -->
```

Voici maintenant un exemple de consommation de ce service :

```
<?php
include 'SCA/SCA.php';

$service = SCA::getService(
    'http://localhost/tests/sca/services/Reversing.wsdl');

header('Content-Type: text/html; charset=utf-8');
echo $service->reverse('Yogui').'<br/>';
echo $service->reverse('BrYs').'<br/>';
echo $service->reverse('mathieu');


function html($string)

{   return htmlspecialchars($string, ENT_QUOTES);

}
```

```
http://localhost/tests/sca/
```

```
iugoY
sYrB
ueihtam
```

Contrairement aux scripts contenus sous le répertoire "SCA", les fichiers index.php services/Reversing.php et services/Reversing.wsdl doivent être exposés sous DocumentRoot. Idéalement, le script de génération des fichiers WSDL devrait être situé en-dehors de DocumentRoot et appelé en lignes de commandes.

Le fichier WSDL est systématiquement généré à côté du script PHP qui lui correspond. Il ne faut pas faire include_once ou require_once du script SCA/SCA.php , mais plutôt include ou require

.

Les services RESTful n'utilisent pas le protocole SOAP, ils n'ont pas besoin de fichier WSDL ; d'après le W3C, il est toutefois possible de décrire un service REST grâce à un fichier WSDL. Nous n'allons néanmoins pas nous en occuper ici, dans la mesure où les standards d'utilisation de services RESTful suffisent à assurer sa pérennité.

Les services RESTful sont consommés au moyen du protocole HTTP :

- **POST** : Mise à jour de données ;
- **GET** : Lecture de données ;
- **PUT** : Enregistrement de nouvelles données ;
- **DELETE** : Suppression de données.

```php
<?php
$url = sprintf
    'http://search.yahooapis.com/WebSearchService/V1/webSearch?appid=%s&query=%s&region=fr',
    rawurlencode('<mettre ici votre clef>'),
    rawurlencode('<mettre ici votre question>'));

$xml = simplexml_load_file ($url);

header('Content-Type: text/html; charset=utf-8');
foreach($xml as $result)

{   echo htmlspecialchars $result->Title, ENT_QUOTES).'<br/>';
                          (
}
```

L'utilisation de services REST en méthode GET est très simple, mais les autres méthodes (POST, PUT et DELETE) sont plus complexes à mettre en place. Il est préférable d'utiliser une bibliothèque afin de ne pas compliquer notre code, Zend Framework étant un exemple parmi d'autres.

```php
<?php
require_once 'Zend/Service/Yahoo.php';

$yahoo= new Zend_Service_Yahoo('<mettre ici votre clef>');
$results = $yahoo->webSearch('<mettre ici votre question>');

header('Content-Type: text/html; charset=utf-8');
foreach($results as $result)

{   echo htmlspecialchars $result->Title, ENT_QUOTES).'<br />';
                          (
}
```

```php
<?php
require_once 'Zend/Service/Amazon.php';

$amazon= new Zend_Service_Amazon('<mettre ici votre clef>');
```

```
$results = $amazon->itemSearch(array
    'SearchIndex' => 'Books',
    'Keywords' => '<mettre ici vos mots clefs>'));

header('Content-Type: text/html; charset=utf-8');
foreach($results as $result)

{   echo htmlspecialchars($result->Title, ENT_QUOTES).'<br />';
}
```

Voyons maintenant comment mettre en place nous-mêmes un service REST.

```
<?php
require_once 'Zend/Rest/Server.php';

/**
 * Returns the reverse of a string
 *
 * @param string $string Value to reverse
 * @return string reverse of value
 */
function reverse($string)

{   return utf8_encode(strrev(utf8_decode($string)));
}
$server = new Zend_Rest_Server();
$server->addFunction('reverse');

$server->handle();
```

```
<?php
require_once 'Zend/Rest/Client.php';

$client = new Zend_Rest_Client(
    'http://localhost/tests/rest/services/reverse.php');

echo html($client->reverse('Yogui', '')->get()).'<br/>';
echo html($client->reverse('BrYs', '')->get()).'<br/>';
echo html($client->reverse('mathieu', '')->get());


function html($string)

{   return htmlspecialchars($string);

}
```

```
http://localhost/tests/rest/
```

```
iugoY
sYrB
ueihtam
```

Avant de mettre en place un service Web pour votre organisation, renseignez-vous sur l'existant. De nombreuses applications utilisent déjà une architecture orientée vers les services (SOA), il peut être judicieux dans ces situations

d'utiliser SCA pour PHP. Dans les autres cas, peut-être que SOAP ou REST sont plus simples à mettre en place, à maintenir et à interfacer avec d'autres services (Google, Yahoo, Amazon...).

Ce cours ne peut pas couvrir la totalité des formats que PHP permet de gérer, surtout si l'on prend en compte la diversité des extensions et bibliothèques disponibles.

Par exemple pour générer des fichiers PDF, on peut utiliser :

- **PDFlib** ;
- **FPDF** ;
- **File_PDF (PEAR)** ;
- **Zend_Pdf** ;
- *etc*.

PHP dispose de bibliothèques ou d'extensions pour gérer tous types de format, même des formats multimédia ou des formats complexes comme Excel. Si vous ne trouvez pas votre bonheur sur php.net ou pecl.php.net, tentez votre chance avec Google ;)

En particulier, les formats d'images sont rendus très accessibles par de nombreuses bibliothèques permettant de générer notamment des graphiques. PHP peut ainsi construire facilement des rapports détaillés et agrémentés de visuels attractifs.

Les exemples qui suivent sont tous bâtis sur le même modèle, à savoir un site de tutoriels. Nous verrons plusieurs manières d'aborder cette thématique, les avantages et inconvénients de chaque méthode, et les conclusions qui s'imposent. Attention, l'ergonomie n'est pas au programme : il faudra utiliser vos propres dons ou vous inspirer d'une autre source.

Ce site est fait le plus simplement possible, chaque script étant indépendant des autres.

Arborescence sous "C:\Web\online\http\cours-php\1-simple" (DocumentRoot) :

- /index.php
- /cours.php
- /auteur.php
-

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr-FR" lang="fr-FR">
<head>
    <title>G. Rossolini - Tutoriels PHP</title>
    <meta http-equiv="content-type" content="text/html; charset=utf-8" />
</head>
<body>

<ul style="text-align: center">
    <li style="display: inline"><a
        href="./"
        title="G. Rossolini - Tutoriels PHP">Accueil</a> -</li>
    <li style="display: inline"><a
        href="./articles.php"
        title="G. Rossolini - Liste des publications">Publications</a> -</li>
    <li style="display: inline"><a
        href="./auteurs.php"
        title="G. Rossolini - Liste des auteurs">Auteurs</a></li>
</ul>
<h1 style="text-align: center">Guillaume Rossolini - Tutoriels PHP</h1>

<h2 style="text-decoration: underline">Présentation</h2>
<p>J'ai suivi une formation dans plusieurs domaines, notamment la communication,
    le développement de logiciels, la gestion de projet,
    <span style="font-style: italic">etc</span>.</p>
<p>Aujoud'hui, je suis un développeur PHP muni de la certification Zend :<br />
<a
    href="http://www.zend.com/store/education/certification/authenticate.php?ClientCandidateID=ZEND005053&Registr
    title="Zend Certified Engineer Details"><img
        src="http://www.zend.com/images/training/php5_zce_logo.gif"
        alt="Logo ZCE"
        style="border: none"/></a></p>

<p style="text-align: center; font-size: 0.7em;">Copyright
    Guillaume Rossolini 2007-<?php echo date('Y'); ?><br />
```

```
Contact : <a
    href="mailto:g-rossolini@developpez.com"
    title="E-mail">g-rossolini@developpez.com</a></p>
</body>
</html>
```

```
<?php
mysql_connect('localhost', 'utilisateur', 'motdepasse');
mysql_select_db('developpez');

function html($string)

{   return utf8_encode(htmlspecialchars($string, ENT_QUOTES));
                    (
}?>
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr-FR" lang="fr-FR">
<head>
    <title>G. Rossolini - Articles</title>
    <meta http-equiv="content-type" content="text/html; charset=utf-8" />
</head>
<body>

<ul style="text-align: center">
    <li style="display: inline"><a
        href="./"
        title="G. Rossolini - Tutoriels PHP">Accueil</a> -</li>
    <li style="display: inline"><a
        href="./articles.php"
        title="G. Rossolini - Liste des publications">Publications</a> -</li>
    <li style="display: inline"><a
        href="./auteurs.php"
        title="G. Rossolini - Liste des auteurs">Auteurs</a></li>
</ul>

<?php
if(empty($_GET['id']))
  (
{   ?>
    <h1 style="text-align: center">Publications</h1>
    <?php
    $sql = 'SELECT c.id, c.name, COUNT(*) AS nb_articles
            FROM category AS c
            INNER JOIN article AS a ON a.category_id = c.id
            GROUP BY c.id, c.name
            ORDER BY c.name';
    $db_categories = mysql_query($sql);
    while($category = mysql_fetch_assoc($db_categories))
    {
        ?><h2><?php echo html($category['name']); ?></h2><?php
        $sql = 'SELECT ar.id, ar.title,
                    au.id AS author_id, au.name AS author_name
                FROM article AS ar
                INNER JOIN author AS au ON ar.author_id = au.id
                WHERE ar.category_id = %d
                ORDER BY ar.title';
        $db_article = mysql_query(sprintf($sql, $category['id']));
        ?><ul><?php             (
        while($article = mysql_fetch_assoc($db_article))
        {
            ?>
                <li><a href="./articles.php?id=<?php
                    echo (int)$article['id'];
                ?>" title="<?php echo html($article['title']); ?>"><?php
                    echo html($article['title']);
                ?></a>, par <a href="./auteurs.php?id=<?php
                    echo (int)$article['author_id'];
```

```
                ?>" title=""><?php
                    echo html($article['author_name']);
                ?></a></li>
            <?php
        }
        ?></ul><?php
    }

else

{   $id = (int)$_GET['id'];
    $sql = 'SELECT ar.title, ar.text,
                au.id AS author_id, au.name AS author_name
            FROM article AS ar
            INNER JOIN author AS au ON ar.author_id = au.id
            WHERE ar.id = %d';
    $db_article = mysql_querysprintf ($sql, $id));
    if ($article = mysql_fetch_assoc($db_article))
    {
        $article['text'] = utf8_decode($article['text']);
        ?>
        <h1 style="text-align: center"><?php
            echo html($article['title']);
        ?></h1>
        <div style="text-align: center; font-style: italic">Par <a
            href="./auteurs.php?id=<?php
                echo html($article['author_id']);
            ?>"
            title="Publications de <?php
                echo html($article['author_name']);
            ?>"><?php
                echo html($article['author_name']);
            ?></a></div>
        <?php
        echo nl2br(html($article['text']));
    }
    else
    {
        echo "Cet article n'existe pas...";
    }

?>

<p style="text-align: center; font-size: 0.7em;">Copyright
    Guillaume Rossolini 2007-<?php echo date('Y'); ?><br />
Contact : <a
    href="mailto:g-rossolini@developpez.com"
    title="E-mail">g-rossolini@developpez.com</a></p>
</body>
</html>
```

```
<?php
mysql_connect('localhost', 'utilisateur', 'motdepasse');
mysql_select_db('developpez');

function html($string)

{   return utf8_encode htmlspecialchars($string, ENT_QUOTES));
                    (
}
?>
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr-FR" lang="fr-FR">
<head>
    <title>G. Rossolini - Publications</title>
    <meta http-equiv="content-type" content="text/html; charset=utf-8" />
</head>
<body>
```

```
<ul style="text-align: center">
    <li style="display: inline"><a
        href="./"
        title="G. Rossolini - Tutoriels PHP">Accueil</a> -</li>
    <li style="display: inline"><a
        href="./articles.php"
        title="G. Rossolini - Liste des publications">Publications</a> -</li>
    <li style="display: inline"><a
        href="./auteurs.php"
        title="G. Rossolini - Liste des auteurs">Auteurs</a></li>
</ul>

<?php
if (empty($_GET['id']))
{   ?>
    <h1 style="text-align: center">Auteurs</h1>
    <ul style="list-style-type: none">
    <?php
    $sql = 'SELECT au.id, au.name, COUNT(*) AS nb_articles
            FROM author AS au
            LEFT JOIN article AS ar ON ar.author_id = au.id
            GROUP BY au.id, au.name
            ORDER BY name';

    $db_authors = mysql_query($sql);
    while($author = mysql_fetch_assoc($db_authors))
    {
        ?><li><a
            href="./auteurs.php?id=<?php echo (int)$author['id']; ?>"
            title="<?php echo html($author['name']); ?>"><?php
                echo html($author['name']);
            ?></a> (<?php
                echo (int)$author['nb_articles'];
            ?> articles)</li><?php
    }
    ?>
    </ul>
    <?php

else

{   $id = (int)$_GET['id'];
    $sql = 'SELECT name
            FROM author
            WHERE id = %d';

    $db_author = mysql_query(sprintf($sql, $id));
    if ($author = mysql_fetch_assoc($db_author))
    {
        ?>
        <h1 style="text-align: center">Publications de <?php
            echo html($author['name']);
        ?></h1>
        <ul>
        <?php
        $sql = 'SELECT id, title
                FROM article
                WHERE author_id = %d
                ORDER BY title';

        $db_article = mysql_query(sprintf($sql, $id));
        while($article = mysql_fetch_assoc($db_article))
        {
            ?><li><a
                href="./articles.php?id=<?php echo html($article['id']); ?>"
                title="Article par <?php echo html($author['name']); ?>"><?php
                    echo html($article['title']);
                ?></a></li><?php
        }
```

```
        ?>
        </ul>
        <?php
    }
    else
    {
        echo "Cet auteur n'existe pas...";
    }

?>

<p style="text-align: center; font-size: 0.7em;">Copyright
    Guillaume Rossolini 2007-<?php echo date('Y'); ?><br />
Contact : <a
    href="mailto:g-rossolini@developpez.com"
    title="E-mail">g-rossolini@developpez.com</a></p>
</body>
</html>
```

Même si tout est parfaitement fonctionnel et que le code XHTML produit est parfaitement standard, voyons maintenant les inconvénients de cette approche, si souvent utilisée par les débutants avant qu'ils prennent de l'expérience.

Ce code est très rapide à écrire.

Puisque le code de chaque script n'est pas très long, nous avons une bonne vision détaillée de leur comportement (en un clin d'oeil).

L'en-tête et le pied de page (code XHTML) sont répétés dans tous les documents, alors qu'ils sont toujours identiques. Nous pourrions définir des fonctions, mais le plus adéquat est d'inclure des fichiers.

Les paramètres d'accès et la connexion à la base de données sont eux aussi répétés dans plusieurs scripts. Une inclusion serait également bienvenue.

Dans le même ordre d'idée, la fonction **html()** est déclarée à l'identique dans tous les scripts.

Le dialogue avec la base de données se fait au moyen de l'API standard de MySQL, ce qui est très restrictif. Il serait préférable d'utiliser PDO.

À chaque retour de requête, nous sommes obligés de faire manuellement une conversion d'encodage. Utiliser une classe étendue de PDO nous permettra de nous affranchir de ce problème.

Un problème moins évident est que le terme "php" apparaît dans les URL. Par conséquent, ces URLs sont éligibles pour toutes les recherches sur "php" dans les moteurs de recherche. Dans mon cas ce n'est pas grave puisqu'il s'agit justement de tutoriels PHP, mais imaginons un site sur l'alimentaire ou tout autre sujet en décalage complet avec PHP... En ce cas, l'extension de fichier dans l'URL n'est pas une bonne chose, il est préférable de s'en débarasser. Cela peut même améliorer la sécurité de votre site...

Enfin, il y a bien trop de mélanges de PHP et de XHTML, le code du script est donc très complexe à lire. Imaginez que j'introduise du code CSS, JavaScript *etc*., le code source de mon script deviendrait rapidement incompréhensible.

L'inconvénient majeur est donc la maintenabilité du code : nous n'avons adopté que très peu de règles d'écriture du code, et par conséquent il est difficile à relire.

Dans un premier temps, nous allons corriger la répétition du code.

Arborescence sous "C:\Web\online\http\cours-php\2-includes" (DocumentRoot) :

- /index.php
- /articles.php
- /auteurs.php
-

Arborescence sous "C:\Web\offline\sites\cours-php\2-includes" (hors de DocumentRoot) :

- /footer.html
- /header.html
- /common.php
- /functions.php
-

Le moyen le plus efficace de rendre les fichiers à inclure accessibles aux scripts du site, est de modifier la configuration d'Apache *via* le **httpd.conf** ou **.htaccess**. Cela nous évitera de mettre tout le chemin vers les fichiers dans tous nos scripts. Voici comment cela se présente dans mon cas :

```
<Directory "C:/Web/online/http/cours-php/2-includes">
    AllowOverride None
    php_value include_path ".;C:/Web/offline/sites/cours-php/2-includes"
</Directory>
```

```
<?php
include 'header.tpl';
?>
<h1 style="text-align: center">Guillaume Rossolini - Tutoriels PHP</h1>

<h2 style="text-decoration: underline">Présentation</h2>
<p>J'ai suivi une formation dans plusieurs domaines, notamment la communication,
    le développement de logiciels, la gestion de projet,
    <span style="font-style: italic">etc</span>.</p>
<p>Aujoud'hui, je suis un développeur PHP muni de la certification Zend :<br />
<a href="http://www.zend.com/store/education/certification/authenticate.php?ClientCandidateID=ZEND005053&Registra
    title="Zend Certified Engineer Details"><img
        src="http://www.zend.com/images/training/php5_zce_logo.gif"
        alt="Logo ZCE"
        style="border: none"/></a></p>
<?php
include 'footer.tpl';
```

```
<?php
```

```php
require_once 'common.php';
require_once 'functions.php';
include 'header.tpl';

if (empty($_GET['id']))
{
    ?>
    <h1 style="text-align: center">Publications</h1>
    <?php
    $sql = 'SELECT c.id, c.name, COUNT(*) AS nb_articles
            FROM category AS c
            INNER JOIN article AS a ON a.category_id = c.id
            GROUP BY c.id, c.name
            ORDER BY c.name';

    $db_categories = mysql_query($sql) or die (mysql_error());
    while ($category = mysql_fetch_assoc($db_categories))
    {
        ?><h2><?php echo html($category['name']); ?></h2><?php
        $sql = 'SELECT ar.id, ar.title,
                    au.id AS author_id, au.name AS author_name
                FROM article AS ar
                INNER JOIN author AS au ON ar.author_id = au.id
                WHERE ar.category_id = %d
                ORDER BY ar.title';
        $db_article = mysql_query(sprintf ($sql, $category['id']));
        ?><ul><?php
        while ($article = mysql_fetch_assoc($db_article))
        {
            ?><li><a href="./articles.php?id=<?php echo (int)$article['id']; ?>"
                title="<?php echo html($article['title']); ?>"><?php
                    echo html($article['title']);
                ?></a>, par <a href="./auteurs.php?id=<?php
                    echo (int)$article['author_id'];
                ?>" title=""><?php
                    echo html($article['author_name']);
                ?></a></li><?php
        }
        ?></ul><?php
    }

}
else

{   $id = (int)$_GET['id'];
    $sql = 'SELECT ar.title, ar.text,
                au.id AS author_id, au.name AS author_name
            FROM article AS ar
            INNER JOIN author AS au ON ar.author_id = au.id
            WHERE ar.id = %d';

    $db_article = mysql_query(sprintf ($sql, $id));
    if ($article = mysql_fetch_assoc($db_article))
    {
        $article['text'] = utf8_decode($article['text']);
        ?>
        <h1 style="text-align: center"><?php
            echo html($article['title']);
        ?></h1>
        <div style="text-align: center; font-style: italic">Par <a
            href="./auteurs.php?id=<?php echo (int)$article['author_id']; ?>"
            title="Publications de <?php
                echo html($article['author_name']);
            ?>"><?php echo html($article['author_name']); ?></a></div>
        <?php
        echo nl2br(html($article['text']));
    }
    else
    {
        echo "Cet article n'existe pas...";
    }
```

```
}
include 'footer.tpl';
```

```
<?php
require_once 'common.php';
require_once 'functions.php';
include 'header.tpl';

if (empty($_GET['id']))
{
    ?>
    <h1 style="text-align: center">Auteurs</h1>
    <ul style="list-style-type: none">
    <?php
    $sql = 'SELECT au.id, au.name, COUNT(*) AS nb_articles
            FROM author AS au
            LEFT JOIN article AS ar ON ar.author_id = au.id
            GROUP BY au.id, au.name
            ORDER BY name';

    $db_authors = mysql_query($sql);
    while($author = mysql_fetch_assoc($db_authors))
    {
        ?><li><a
            href="./auteurs.php?id=<?php echo (int)$author['id']; ?>"
            title="<?php echo html($author['name']); ?>"><?php
                echo html($author['name']);
            ?></a> (<?php
                echo (int)$author['nb_articles'];
            ?> articles)</li><?php
    }
    ?>
    </ul>
    <?php

}
else
{
    $id = (int)$_GET['id'];
    $sql = 'SELECT name
            FROM author
            WHERE id = %d';

    $db_author = mysql_query(sprintf($sql, $id));
    if ($author = mysql_fetch_assoc($db_author))
    {
        ?>
        <h1 style="text-align: center">Publications de <?php
            echo html($author['name']);
        ?></h1>
        <ul>
        <?php
        $sql = 'SELECT id, title
                FROM article
                WHERE author_id = %d
                ORDER BY title';

        $db_article = mysql_query(sprintf($sql, $id));
        while($article = mysql_fetch_assoc($db_article))
        {
            ?><li><a
                href="./articles.php?id=<?php echo (int)$article['id']; ?>"
                title="Article par <?php
                    echo html($author['name']);
                ?>"><?php echo html($article['title']); ?></a></li><?php
        }
        ?>
        </ul>
        <?php
```

```
    }
    else
    {
        echo "Cet auteur n'existe pas...";
    }

}
include 'footer.tpl';
```

```
<?php
mysql_connect('localhost', 'utilisateur', 'motdepasse');
mysql_select_db('developpez');
```

```
<?php
function html($string)

{   return utf8_encode htmlspecialchars($string, ENT_QUOTES));
                    (
}
```

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr-FR" lang="fr-FR">
<head>
    <title>G. Rossolini - Tutoriels PHP</title>
    <meta http-equiv="content-type" content="text/html; charset=utf-8" />
</head>
<body>

<ul style="text-align: center">
    <li style="display: inline"><a
        href="./"
        title="G. Rossolini - Tutoriels PHP">Accueil</a> -</li>
    <li style="display: inline"><a
        href="./articles.php"
        title="G. Rossolini - Liste des publications">Publications</a> -</li>
    <li style="display: inline"><a
        href="./auteurs.php"
        title="G. Rossolini - Liste des auteurs">Auteurs</a></li>
</ul>
```

```
<p style="text-align: center; font-size: 0.7em;">Copyright
    Guillaume Rossolini 2007-<?php echo date('Y'); ?><br />
Contact : <a
    href="mailto:g-rossolini@developpez.com"
    title="E-mail">g-rossolini@developpez.com</a></p>
</body>
</html>
```

Toute modification de l'en-tête ou du pied de page est immédiatement répercutée dans toutes les pages du site.

Puisqu'ils sont hors du *DocumentRoot*, les fichiers à inclure sont protégés de tout accès externe. Leur chemin d'accès étant dans l'*include_path* de PHP, ils sont également facilement accessibles.

Les fichiers à inclure ne sont plus intégralement copiés dans les scripts effectifs, mais l'on se rend compte que les instructions require* et include* se répètent d'un script à l'autre. Nous verrons plus loin que MVC nous permet d'éviter cette situation.

Par exemple, si je souhaite conserver des statistiques sur mes visiteurs, une solution serait de créer un script "stats.php" à inclure dans toutes mes pages. Je serais donc obligé de répéter ce code dans tous les scripts :

```php
<?php
require_once 'common.php';
require_once 'functions.php';
include 'stats.php';
include 'header.tpl';
```

Un problème évident avec le code ci-dessus est que nous pourrions oublier d'inclure "stats.php" dnas l'un des scripts... Un autre est que le bloc d'includes/requires est tout simplement du code répété, or il nous faut éviter ce genre de situations.

Nous avons déjà établi que l'extension ".php" est un problème dans l'URL. Maintenant que notre menu d'en-tête est géré par des inclusions, nous pouvons plus facilement modifier les liens du site.

Le principe est d'utiliser des URLs absolues à partir de la racine du site, donc commençant par "/" plutôt que "./", pour chacun des liens. Nous allons mettre chaque script dans un répertoire, ce qui nous permettra de l'appeler "index.php" et ainsi de nous passer totalement du nom du script dans l'URL. Un nouveau fichier de configuration permet de respecter **DRY** (*don't repeat yourself*).

Arborescence sous "C:\Web\online\http\cours-php\3-extensions" (DocumentRoot) :

- /index.php
- /articles/index.php
- /auteurs/index.php
-

Arborescence sous "C:\Web\offline\sites\cours-php\3-extensions" (hors de DocumentRoot) :

- /footer.html
- /header.html
- /common.php
- /functions.php
-

```
<Directory "C:/Web/online/http/cours-php/3-extensions">
    AllowOverride None
    php_value include_path ".;C:/Web/offline/sites/cours-php/3-extensions"
    SetEnv HTTP_ROOT /cours-php/3-extensions/
```

```
</Directory>
```

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr-FR" lang="fr-FR">
<head>
    <title>G. Rossolini - Tutoriels PHP</title>
    <meta http-equiv="content-type" content="text/html; charset=utf-8" />
</head>
<body>

<ul style="text-align: center">
    <li style="display: inline"><a
        href="<?php echo html($_SERVER['HTTP_ROOT']); ?>"
        title="G. Rossolini - Tutoriels PHP">Accueil</a> -</li>
    <li style="display: inline"><a
        href="<?php echo html($_SERVER['HTTP_ROOT']); ?>articles/"
        title="G. Rossolini - Liste des publications">Publications</a> -</li>
    <li style="display: inline"><a
        href="<?php echo html($_SERVER['HTTP_ROOT']); ?>auteurs/"
        title="G. Rossolini - Liste des auteurs">Auteurs</a></li>
</ul>
```

```
<?php
require_once 'common.php';
require_once 'functions.php';
include 'header.tpl';

if (empty($_GET['id']))
{
    ?>
    <h1 style="text-align: center">Publications</h1>
    <?php
    $sql = 'SELECT c.id, c.name, COUNT(*) AS nb_articles
            FROM category AS c
            INNER JOIN article AS a ON a.category_id = c.id
            GROUP BY c.id, c.name
            ORDER BY c.name';

    $db_categories = mysql_query($sql) or die mysql_error());
    while($category = mysql_fetch_assoc($db_categories))
    {
        ?><h2><?php echo html($category['name']); ?></h2><?php
        $sql = 'SELECT ar.id, ar.title,
                    au.id AS author_id, au.name AS author_name
                FROM article AS ar
                INNER JOIN author AS au ON ar.author_id = au.id
                WHERE ar.category_id = %d
                ORDER BY ar.title';

        $db_article = mysql_query(sprintf($sql, $category['id']));
        ?><ul><?php
        while($article = mysql_fetch_assoc($db_article))
        {
            ?><li><a href="<?php
                echo html($_SERVER['HTTP_ROOT']);
            ?>articles/?id=<?php
                echo (int)$article['id'];
            ?>" title="<?php
                echo html($article['title']);
            ?>"><?php
                echo html($article['title']);
            ?></a>, par <a href="<?php
                echo html($_SERVER['HTTP_ROOT']);
            ?>auteurs/?id=<?php
                echo (int)$article['author_id'];
```

```
            ?>" title=""><?php
                echo html($article['author_name']);
            ?></a></li><?php
        }
        ?></ul><?php
    }

else

{   $id = (int)$_GET['id'];
    $sql = 'SELECT ar.title, ar.text,
                au.id AS author_id, au.name AS author_name
            FROM article AS ar
            INNER JOIN author AS au ON ar.author_id = au.id
            WHERE ar.id = %d';

    $db_article = mysql_query(sprintf($sql, $id));
    if ($article = mysql_fetch_assoc($db_article))
    {
        $article['text'] = utf8_decode($article['text']);
        ?>
        <h1 style="text-align: center"><?php
            echo html($article['title']);
        ?></h1>
        <div style="text-align: center; font-style: italic">Par <a href="<?php
            echo html($_SERVER['HTTP_ROOT']);
        ?>auteurs/?id=<?php
            echo (int)$article['author_id'];
        ?>" title="Publications de <?php
            echo html($article['author_name']);
        ?>"><?php
            echo html($article['author_name']);
        ?></a></div>
        <?php
        echo nl2br(html($article['text']));
    }
    else
    {
        echo "Cet article n'existe pas...";
    }

}
include 'footer.tpl';
```

```
<?php
require_once 'common.php';
require_once 'functions.php';
include 'header.tpl';

if (empty($_GET['id']))
{   ?>
    <h1 style="text-align: center">Auteurs</h1>
    <ul style="list-style-type: none">
    <?php
    $sql = 'SELECT au.id, au.name, COUNT(*) AS nb_articles
            FROM author AS au
            LEFT JOIN article AS ar ON ar.author_id = au.id
            GROUP BY au.id, au.name
            ORDER BY name';

    $db_authors = mysql_query($sql);
    while($author = mysql_fetch_assoc($db_authors))
    {
        ?><li><a href="<?php
            echo html($_SERVER['HTTP_ROOT']);
        ?>auteurs/?id=<?php
            echo (int)$author['id'];
        ?>" title="<?php
```

```
            echo html($author['name']);
        ?>"><?php
            echo html($author['name']);
        ?></a> (<?php
            echo (int)$author['nb_articles'];
        ?> articles)</li><?php
    }
    ?>
    </ul>
    <?php

}
else

{   $id = (int)$_GET['id'];
    $sql = 'SELECT name
            FROM author
            WHERE id = %d';

    $db_author = mysql_query(sprintf($sql, $id));
    if ($author = mysql_fetch_assoc($db_author))
    {
        ?>
        <h1 style="text-align: center">Publications de <?php
            echo html($author['name']);
        ?></h1>
        <ul>
        <?php
        $sql = 'SELECT id, title
                FROM article
                WHERE author_id = %d
                ORDER BY title';

        $db_article = mysql_query(sprintf($sql, $id));
        while($article = mysql_fetch_assoc($db_article))
        {
            ?><li><a href="<?php
                echo html($_SERVER['HTTP_ROOT']);
            ?>articles/?id=<?php
                echo (int)$article['id'];
            ?>" title="Article par <?php
                echo html($author['name']);
            ?>"><?php
                echo html($article['title']);
            ?></a></li><?php
        }
        ?>
        </ul>
        <?php
    }
    else
    {
        echo "Cet auteur n'existe pas...";
    }

}
include 'footer.tpl';
```

L'extension ".php" n'apparaît plus dans l'URL, les moteurs de recherche peuvent donc référencer la totalité de nos URLs sans trouver de mot inutile.

Nous avons maintenant autant de répertoires publics que de scripts, ce qui n'est pas nécessairement très pratique ou utile pour notre organisation interne.

Arborescence sous "C:\Web\online\http\cours-php\4-modeles" (DocumentRoot) :

- /index.php
- /articles.php
- /auteurs.php
-

Arborescence sous "C:\Web\offline\sites\cours-php\4-modeles" (hors de DocumentRoot) :

- /AbstractTable.php
- /Article.php
- /Author.php
- /Category.php
- /common.php
- /footer.html
- /functions.php
- /header.html
- /MyPDO.php
-

```
<Directory "C:/Web/online/http/cours-php/4-modeles">
    AllowOverride None
    php_value include_path ".;C:/Web/offline/sites/cours-php/4-modeles"
    SetEnv HTTP_ROOT /cours-php/4-modeles/
</Directory>
```

```
<?php
abstract class AbstractTable

{   protected $db;
    protected $selectAll;
    protected $selectById;

    public function __construct($db)
    {
        $this->db = $db;
    }

    public function getAll()
    {
        $this->selectAll->execute();
        return $this->selectAll->fetchAll();
    }

    public function getById($id)
    {
        $this->selectById->execute(array($id));
        $row = $this->selectById->fetchAll();
        if (empty($row[0]))
        { (
            return array();
        }
```

```
        else
        {
            return $row[0];
        }
    }

}
```

```
<?php
spl_autoload_register('generic_autoload');
$db = new MyPDO(
    'mysql:host=localhost;dbname=developpez', 'utilisateur', 'motdepasse');
```

```
<?php
function html($string)

{   return utf8_encode htmlspecialchars($string, ENT_QUOTES));
                    (
}
function generic_autoload($class)

{   require_once $class.'.php';

}
```

```
<?php
class MyPDO extends PDO

{   public function __construct($dsn, $user=NULL, $password=NULL
    {                                                          )
        parent::__construct($dsn, $user, $password);
        $this->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
    }

    public function prepare($sql, $options=NULL
    {                                       )
        $statement = parent::prepare($sql);
        if strpos strtoupper($sql), 'SELECT') === 0) //requête "SELECT"
        { (      (
            $statement->setFetchMode(PDO::FETCH_ASSOC);
        }

        return $statement;
    }

}
```

```
<?php
class Article extends AbstractTable

{   protected $selectById;
    protected $selectByAuthor;
    protected $selectByCategory;

    public function __construct($db)
    {
        parent::__construct($db);

        $sql = 'SELECT ar.title, ar.text,
                    au.id AS author_id, au.name AS author_name
                FROM article AS ar
                INNER JOIN author AS au ON ar.author_id = au.id
                WHERE ar.id = ?';
        $this->selectById = $this->db->prepare($sql);

        $sql = 'SELECT id, title
```

```
                FROM article
                WHERE author_id = ?
                ORDER BY title';
        $this->selectByAuthor = $this->db->prepare($sql);

        $sql = 'SELECT ar.id, ar.title,
                    au.id AS author_id, au.name AS author_name
                FROM article AS ar
                INNER JOIN author AS au ON ar.author_id = au.id
                WHERE ar.category_id = ?
                ORDER BY ar.title';
        $this->selectByCategory = $this->db->prepare($sql);
    }

    public function getById($id)
    {
        $this->selectById->execute(array($id));
        $articles = $this->selectById->fetchAll();
        if (empty($articles[0]))
        { (
            return array();
        }
        else
        {
            $articles[0]['text'] = utf8_decode($articles[0]['text']);
            return $articles[0];
        }
    }

    public function getByAuthor($id)
    {
        $this->selectByAuthor->execute(array($id));
        return $this->selectByAuthor->fetchAll();
    }

    public function getByCategory($id)
    {
        $this->selectByCategory->execute(array($id));
        return $this->selectByCategory->fetchAll();
    }

}
```

```
<?php
class Author extends AbstractTable

{   public function __construct($db)
    {
        parent::__construct($db);

        $sql = 'SELECT au.id, au.name, COUNT(*) AS nb_articles
                FROM author AS au
                LEFT JOIN article AS ar ON ar.author_id = au.id
                GROUP BY au.id, au.name
                ORDER BY name';
        $this->selectAll = $this->db->prepare($sql);

        $sql = 'SELECT name
                FROM author
                WHERE id = ?';
        $this->selectById = $this->db->prepare($sql);
    }

}
```

```
<?php
class Category extends AbstractTable

{   public function __construct($db)
```

```
    {
        parent::__construct($db);

        $sql = 'SELECT c.id, c.name, COUNT(*) AS nb_articles
                FROM category AS c
                INNER JOIN article AS a ON a.category_id = c.id
                GROUP BY c.id, c.name
                ORDER BY c.name';
        $this->selectAll = $this->db->prepare($sql);
    }

}
```

```
<?php
require_once 'functions.php';
include 'header.tpl';
?>
<h1 style="text-align: center">Guillaume Rossolini - Tutoriels PHP</h1>

<h2 style="text-decoration: underline">Présentation</h2>
<p>J'ai suivi une formation dans plusieurs domaines, notamment la communication,
    le développement de logiciels, la gestion de projet,
    <span style="font-style: italic">etc</span>.</p>
<p>Aujoud'hui, je suis un développeur PHP muni de la certification Zend :<br />
<a href="http://www.zend.com/store/education/certification/authenticate.php?ClientCandidateID=ZEND005053&Registra
    title="Zend Certified Engineer Details"><img
        src="http://www.zend.com/images/training/php5_zce_logo.gif"
        alt="Logo ZCE"
        style="border: none"/></a></p>
<?php
include 'footer.tpl';
```

```
<?php
require_once 'functions.php';
require_once 'common.php';
include 'header.tpl';

$articles   = new Article($db);
$categories = new Category($db);

if (empty($_GET['id']))
{
    ?>
    <h1 style="text-align: center">Publications</h1>
    <?php
    foreach ($categories->getAll() as $category)
    {
        ?><h2><?php echo html($category['name']); ?></h2><?php
        ?><ul><?php
        foreach ($articles->getByCategory($category['id']) as $article)
        {
            ?><li><a href="<?php
                echo html($_SERVER['HTTP_ROOT']);
            ?>articles/?id=<?php
                echo (int)$article['id'];
            ?>" title="<?php
                echo html($article['title']);
            ?>"><?php
                echo html($article['title']);
            ?></a>, par <a href="<?php
                echo html($_SERVER['HTTP_ROOT']);
            ?>auteurs/?id=<?php
                echo (int)$article['author_id'];
            ?>" title=""><?php
                echo html($article['author_name']);
            ?></a></li><?php
        }
        ?></ul><?php
```

```
    }
else

{   $id = (int)$_GET['id'];
    if ($article = $articles ->getById($id))
    {
        ?>
        <h1 style="text-align: center"><?php
            echo html($article['title']);
        ?></h1>
        <div style="text-align: center; font-style: italic">Par <a
            href="./auteurs.php?id=<?php echo (int)$article['author_id']; ?>"
            title="Publications de <?php
                echo html($article['author_name']);
            ?>"><?php echo html($article['author_name']); ?></a></div>
        <?php
        echo nl2br(html($article['text']));
    }
    else
    {
        echo "Cet article n'existe pas...";
    }

}
include 'footer.tpl';
```

```
<?php
require_once 'functions.php';
require_once 'common.php';
include 'header.tpl';

$authors = new Author($db);
$articles  = new Article($db);

if  empty($_GET['id']))
  (
{   ?>
    <h1 style="text-align: center">Auteurs</h1>
    <ul style="list-style-type: none">
    <?php
    foreach $authors->getAll() as $author)
    {       (
        ?><li><a href="<?php
            echo html($_SERVER['HTTP_ROOT']);
        ?>auteurs/?id=<?php
            echo (int)$author['id'];
        ?>" title="<?php echo html($author['name']); ?>"><?php
            echo html($author['name']);
        ?></a> (<?php echo (int)$author['nb_articles']; ?> articles)</li><?php
    }
    ?>
    </ul>
    <?php

else

{   $id = (int)$_GET['id'];
    if ($author = $authors->getById($id))
    {
        ?>
        <h1 style="text-align: center">Publications de <?php
            echo html($author['name']);
        ?></h1>
        <ul>
        <?php
        foreach $articles ->getByAuthor($id) as $article)
        {       (
```

```
            ?><li><a href="<?php
                echo html($_SERVER['HTTP_ROOT']);
            ?>articles/?id=<?php
                echo (int)$article['id'];
            ?>" title="Article par <?php echo html($author['name']); ?>"><?php
                echo html($article['title']);
            ?></a></li><?php
        }
        ?>
        </ul>
        <?php
    }
    else
    {
        echo "Cet auteur n'existe pas...";
    }

}
include 'footer.tpl';
```

Le code peut maintenant être porté sans aucun problème d'un SGBD à un autre. De plus, les requêtes sont facilement identifiables puisqu'elles sont regroupées dans des classes, et nous sommes totalement protégés contre les injections SQL.

Les scripts à inclure sont tous regroupés et mélangés, il n'y a aucune organisation. Cela peut convenir pour un site de quelques pages mais, à mesure que le site grossit, il devient apparent qu'il faut structurer ces fichiers. Il est temps d'adopter une structure **MVC** .

Il reste encore beaucoup de mélange PHP/HTML. Heureusement, nous avons effectué de nombreuses mises en facteur du reste du code, et par conséquent ces mélanges PHP/HTML sont à présent tout à fait lisibles.

Nous pouvons maintenant résoudre le problème des URLs d'accès à nos pages. En effet, nous avons vu plus tôt dans ce cours que les URLs à base de paramètres GET visibles sont peu efficaces, tant pour l'oeil humain que pour les robots qui s'occupent de référencer nos sites.

Une solution à ces problèmes est la **réécriture de liens**, aussi appelée "URL Rewriting" ou encore "routage".

Nous allons utiliser certains aspects avancés d'Apache, et en particulier le module **mod_rewrite** (qu'il faut donc activer dans **httpd.conf**). Il nous faudra aussi de bonnes connaissances en **expressions régulières** (*aka* "regex" ou "expressions rationnelles").

Le principe de la réécriture de liens est d'utiliser une URL publique ne correspondant pas à un fichier sur le serveur. Le **mod_rewrite** nous permet de router ces URLs vers les scripts de notre serveur, en interne, c'est-à-dire sans causer de redirection HTTP et sans que le client s'en rende compte. Nous pouvons ainsi choisir les paramètres GET les plus adaptés, sans pour autant encombrer le code de nos scripts PHP.

Arborescence sous "C:\Web\online\http\cours-php\5-routing" (DocumentRoot) :

- /

•/index.php
/articles.php
• /auteurs.php
•

Arborescence sous "C:\Web\offline\sites\cours-php\5-routing" (hors de DocumentRoot) :

- /AbstractTable.php
- /Article.php
- /Author.php
- /Category.php
- /common.php
- /footer.html
- /functions.php
- /header.html
- /MyPDO.php
•

Pour pouvoir utiliser la réécriture avec Apache dans un fichier **.htaccess**, vous aurez besoin d'activer l'option **+FollowSymLinks** :

```
Options +FollowSymLinks
RewriteEngine On
```

Sans cette option, utiliser le **RewriteEngine** dans un fichier **.htaccess** donnera lieu à une erreur HTTP 403 et à une entrée log de ce style :

```
Options FollowSymLinks or SymLinksIfOwnerMatch is off
    which implies that RewriteRule directive is forbidden
```

Nous aurons peu de modifications. Les liens doivent maintenant être absolus afin de ne pas porter à confusion ; pour le reste, le **httpd.conf** est à peu près le seul fichier à modifier. Je vais indiquer ici uniquement les fichiers modifiés :

```
<Directory "C:/Web/online/http/cours-php/5-routing">
    AllowOverride None
    php_value include_path ".;C:/Web/offline/sites/cours-php/5-routing"
    SetEnv HTTP_ROOT /cours-php/5-routing/
    RewriteEngine on
    RewriteBase /cours-php/5-routing/
    RewriteRule ^auteurs/$ auteurs.php [L]
    RewriteRule ^auteur/([0-9])-?.*$ auteurs.php?id=$1 [L]
    RewriteRule ^articles/$ articles.php [L]
    RewriteRule ^article/([0-9])-?.*$ articles.php?id=$1 [L]
</Directory>
```

```
<?php
function html($string)

{   return utf8_encode htmlspecialchars($string, ENT_QUOTES));
                    (
}
function url_format($string)

{   $string = str_replace (' ', '-', strtolower($string));
    return html(preg_replace('#[^a-z0-9-]#', '', $string));
```

```
}
function generic_autoload($class)

{   require_once $class.'.php';

}
```

```
<?php
require_once 'functions.php';
require_once 'common.php';
include 'header.tpl';

$authors = new Author($db);
$articles  = new Article($db);

if empty($_GET['id']))
  (
{   ?>
    <h1 style="text-align: center">Auteurs</h1>
    <ul style="list-style-type: none">
    <?php
    foreach $authors->getAll() as $author)
    {       (
        ?><li><a href="<?php
            echo html($_SERVER['HTTP_ROOT']);
        ?>auteur/<?php
            echo url_format($author['id'].'-'.$author['name']);
        ?>" title="<?php
            echo html($author['name']);
        ?>"><?php
            echo html($author['name']);
        ?></a> (<?php echo (int)$author['nb_articles']; ?> articles)</li><?php
    }
    ?>
    </ul>
    <?php

else

{   $id = (int)$_GET['id'];
    if ($author = $authors->getById($id))
    {
        ?>
        <h1 style="text-align: center">Publications de <?php
            echo html($author['name']);
        ?></h1>
        <ul>
        <?php
        foreach $articles ->getByAuthor($id) as $article)
        {       (
            ?><li><a href="<?php
                echo html($_SERVER['HTTP_ROOT']);
            ?>article/<?php
                echo url_format($article['id'].'-'.$article['title']);
            ?>" title="Article par <?php
                echo html($author['name']);
            ?>"><?php echo html($article['title']); ?></a></li><?php
        }
        ?>
        </ul>
        <?php
    }
    else
    {
        echo "Cet auteur n'existe pas...";
    }

}
```

```php
include 'footer.tpl';
```

```php
<?php
require_once 'functions.php';
require_once 'common.php';
include 'header.tpl';

$articles  = new Article($db);
$categories = new Category($db);

if (empty($_GET['id']))
{
    ?>
    <h1 style="text-align: center">Publications</h1>
    <?php
    foreach ($categories->getAll() as $category)
    {
        ?><h2><?php echo html($category['name']); ?></h2><?php
        ?><ul><?php
        foreach ($articles->getByCategory($category['id']) as $article)
        {
            ?><li><a href="<?php
                echo html($_SERVER['HTTP_ROOT']);
            ?>article/<?php
                echo url_format($article['id'].'-'.$article['title'])
            ?>" title="<?php
                echo html($article['title']);
            ?>"><?php
                echo html($article['title']);
            ?></a>, par <a href="<?php
                echo html($_SERVER['HTTP_ROOT']);
            ?>auteur/<?php
                echo url_format(
                    $article['author_id'].'-'.$article['author_name']);
            ?>" title=""><?php echo html($article['author_name']); ?></a></li><?php
        }
        ?></ul><?php
    }

}
else

{
    $id = (int)$_GET['id'];
    if ($article = $articles->getById($id))
    {
        ?>
        <h1 style="text-align: center"><?php
            echo html($article['title']);
        ?></h1>
        <div style="text-align: center; font-style: italic">Par <a
            href="./auteurs.php?id=<?php echo html($article['author_id']); ?>"
            title="Publications de <?php
                echo html($article['author_name']);
            ?>"><?php echo html($article['author_name']); ?></a></div>
        <?php
        echo nl2br(html($article['text']));
    }
    else
    {
        echo "Cet article n'existe pas...";
    }

}
include 'footer.tpl';
```

L'inconvénient majeur est que nous avons maintenant une infinité de points d'entrée pour chaque page. En effet, les anciennes URLs fonctionnent encore : c'est inévitable puisque les scripts **auteurs.php** et **articles.php** sont encore accessibles depuis l'extérieur.

Par ailleurs, on peut maintenant mettre n'importe quelle chaîne à la suite des identifiants numériques : la présence de l'identifiant est suffisante pour charger la page. Cela pose de gros problèmes de *duplicate content* (plusieurs URLs pour une seule ressource).

Ces deux URLs chargent exactement le même document :

- http://localhost/cours-php/4-routing/article/6-dcouverte-des-principaux-moteurs-de-template-en-php
- http://localhost/cours-php/4-routing/article/6-alterner-entre-plusieurs-versions-dapache-et-de-php

Dans cet exemple, la 2° URL est fausse puisque le titre de l'article n° 5 est donné à l'URL conduisant à l'article n° 6. Cependant, nous n'avons mis en place aucun mécanisme pour indiquer à l'utilisateur qu'il n'utilise pas la bonne URL. Si nous envoyons encore une redirection HTTP, les navigateurs risquent de déceler une "redirection infinie" et de ne jamais afficher le document souhaité.

Un **design pattern** permet de résoudre la plupart des inconvénients énoncés ci-dessus : **MVC**.

L'objectif est d'utiliser des *contrôleurs* pour traiter les données, des *modèles* pour les récupérer et des *vues* pour les afficher. Un contrôleur principal, appelé **front controller** ou encore *contrôleur frontal*, est utilisé pour centraliser l'ensemble des requêtes du client. Ainsi, un seul script PHP est exposé en HTTP, ce qui limite les problèmes de *duplicate content* et améliore la sécurité du site. C'est le *front controller* qui répartit (*dispatch*) le travail aux autres contrôleurs.

Puisque les URLs d'un site utilisant une architecture MVC sont fictives, cela nous impose d'utiliser une forme d'URL Rewriting. Ainsi, les URLs peuvent rester identiques à celles définies depuis nos essais d'URL Rewriting. Tout dépend de notre découpage modules/actions.

Notre site actuel est trop simple pour être un exemple suffisamment explicite, mais nous pouvons facilement dégager des actions au sein de chaque contrôleur.

Les contrôleurs :

- **Article** : Permet de lister (et d'administrer) les articles ;
- **Auteur** : Permet de lister (et d'administrer) les auteurs ;
- **Erreur** : Dicte la conduite en cas d'erreur.

Actions dans le contrôleur "Auteur" :

- **index** : Liste les acteurs ;
- **display** : Affiche les articles d'un auteur ;
- **filter** : Dans un site plus complet, cette action permet d'avoir une liste filtrée) ;
- (**update** : Dans un site plus complet, cette action met à jour un auteur) ;
- (**delete** : Dans un site plus complet, cette action supprime un auteur) ;
- *etc*.

Actions dans le contrôleur "Article" :

- **index** : Liste les articles ;
- **display** : Affiche un article ;
- **filter** : Dans un site plus complet, cette action permet d'avoir une liste filtrée) ;
- (**update** : Dans un site plus complet, cette action met à jour un article) ;
- (**delete** : Dans un site plus complet, cette action supprime un article) ;
- (*etc*
- .

Actions dans le contrôleur "Erreur" :

- **index** : Serveur indisponible ;
- **http** : Redirection, page inaccessible ;
- **sql** : Erreur interne) ;
- (*etc*
- .

Il est maintenant très clair que nous devrions avoir un contrôleur "Catégorie" et que, depuis le début de ce tutoriel, il manque un script pour afficher les catégories. Une bonne architecture peut aider à déceler les problèmes dans les applications.

Arborescence sous "C:\Web\online\http\cours-php\6-mvc" (DocumentRoot) :

- /index.php
-

Arborescence sous "C:\Web\offline\sites\cours-php\6-mvc" (hors de DocumentRoot) :

- /MyPDO.php
- /Controller
- /Controller/Article.php
- /Controller/Author.php
- /Controller/Error.php
- /Controller/Index.php
- /Controller/Template.php
- /Model
- /Model/Article.php
- /Model/Author.php
- /Model/Category.php
- /Model/Template.php
- /View
- /View/footer.tpl
- /View/header.tpl
- /View/article/
- /View/article/index.tpl
- /View/article/display.tpl
- /View/author/
- /View/author/index.tpl
- /View/author/display.tpl
- /View/error/
- /View/error/index.tpl
- /View/index/
- /View/index/index.tpl
-

```
<Directory "C:/Web/online/http/cours-php/6-mvc">
    AllowOverride None
```

```
    php_value include_path ".;C:/Web/offline/sites/cours-php/6-mvc"
    SetEnv HTTP_ROOT /cours-php/6-mvc/
    RewriteEngine on
    RewriteCond %{REQUEST_URI} !\.(js|css|jpg|png|gif)$
    RewriteRule .* index.php
</Directory>
```

La directive RewriteCond ci-dessus n'est pas utile dans notre exemple, mais elle vous montre comment on évite aux images ainsi qu'aux scripts CSS et Javascript etc. de passer par notre contrôleur frontal PHP. En effet, ce serait une très grande perte de ressources et cela n'aurait aucune valeur ajoutée.

```
<?php
require 'functions.php';

spl_autoload_register('generic_autoload');
Controller_Template::$db = new MyPDO(
    'mysql:host=localhost;dbname=developpez', 'utilisateur', 'motdepasse');

preg_match
    '#^'.$_SERVER['HTTP_ROOT'].'(?:([a-z]+)/)?#',
    $_SERVER['REQUEST_URI'],
    $match);

if empty($match[1]))
  (
{   $controller = Controller_Index::getInstance();
    $controller->index();

}else

{   switch($match[1])
    {
        case 'articles':
            $controller = Controller_Article::getInstance();
            $controller->index();
        break
             ;
        case 'auteurs':
            $controller = Controller_Author::getInstance();
            $controller->index();
        break
             ;
        case 'article':
            if preg_match
              ( '#^'.$_SERVER['HTTP_ROOT'].'article/([0-9]+)-.*$#',
                $_SERVER['REQUEST_URI'],
                $match))
            {
                $controller = Controller_Article::getInstance();
                $controller->display($match[1]);
            }
            else
            {
                Controller_Error::documentNotFound(
                    "Page introuvable : URL incorrecte");
            }
        break
             ;
        case 'auteur':
            if preg_match
              ( '#^'.$_SERVER['HTTP_ROOT'].'auteur/([0-9]+)-.*$#',
                $_SERVER['REQUEST_URI'],
                $match))
            {
                $controller = Controller_Author::getInstance();
                $controller->display($match[1]);
```

```
            }
            else
            {
                Controller_Error::documentNotFound(
                    "Page introuvable : URL incorrecte");
            }
        break
             ;
        default
            Controller_Error::documentNotFound(
                "Page introuvable : URL incorrecte");
    }

}
```

```
<?php
function html($string)

{   return utf8_encode htm lspecialchars($string, ENT_QUOTES));
                    (
}
function url_format($string)

{   $string = str_replace (' ', '-', strtolow er($string));
    return html(preg_replace('#[^a-z0-9-]#', '', $string));

}
function generic_autoload($class)

{   require_once str_replace ('_', '/', $class).'.php';

}
```

```
<?php
class MyPDO extends PDO

{   public function __construct($dsn, $user=NULL, $password=NULL
    {                                                          )
        parent::__construct($dsn, $user, $password);
        $this->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
    }

    public function prepare($sql, $options=NULL
    {                                      )
        $statement = parent::prepare($sql);
        if strpos strtoupper($sql), 'SELECT') === 0) //requête "SELECT"
        { (      (
            $statem ent->setFetchMode(PDO::FETCH_ASSOC);
        }

        return $statement;
    }

}
```

```
<?php
class Controller_Article extends Controller_Template

{   protected $categoriesModel;

    protected function __construct()
    {
        parent::__construct();
        $this->selfModel = new Model_Article();
        $this->categoriesModel = NULL
    }                                ;

    public function index()
```

```
    {
        $title = "Liste des publications";
        $this->categoriesModel = new Model_Category();
        $categories = $this->categoriesModel->getAll();

        foreach($categories as $i => $category)
        {
            $categories[$i]['articles'] = $this->selfModel->getByCategory(
                $category['id']);
        }

        header('Content-Type: text/html; charset=utf-8');
        require 'View/header.tpl';
        require 'View/article/index.tpl';
        require 'View/footer.tpl';
    }

    public function display($id)
    {
        $article = $this->selfModel->getById($id);
        if (!$article)
        {
            Controller_Error::documentNotFound();
        }
        else
        {
            $title = $article['title'];

            header('Content-Type: text/html; charset=utf-8');
            require 'View/header.tpl';
            require 'View/article/display.tpl';
            require 'View/footer.tpl';
        }
    }

}
```

```
<?php
class Controller_Author extends Controller_Template

{   protected $articlesModel;

    protected function __construct()
    {
        parent::__construct();
        $this->selfModel = new Model_Author();
        $this->articlesModel = NULL
    }                           ;

    public function index()
    {
        $title = "Liste des auteurs";
        $authors = $this->selfModel->getAll();

        header('Content-Type: text/html; charset=utf-8');
        require 'View/header.tpl';
        require 'View/author/index.tpl';
        require 'View/footer.tpl';
    }

    public function display($id)
    {
        $author = $this->selfModel->getById($id);
        if (!$author)
        {
            Controller_Error::documentNotFound("Cet auteur n'existe pas");
        }
        else
        {
```

```
            $title = "Publications de ".$author['name'];
            $this->articlesModel = new Model_Article();
            $articles = $this->articlesModel->getByAuthor($id);

            header('Content-Type: text/html; charset=utf-8');
            require 'View/header.tpl';
            require 'View/author/display.tpl';
            require 'View/footer.tpl';
        }
    }

}
```

```
<?php
abstract class Controller_Error extends Controller_Template

{   public static function documentNotFound($title)
    {
        header('HTTP/1.1 404 Not Found');
        header('Content-Type: text/html; charset=utf-8');
        include 'View/header.tpl';
        include 'View/error/404.tpl';
        include 'View/footer.tpl';
    }

}
```

```
<?php
class Controller_Index extends Controller_Template

{   public function index()
    {
        $title = "Guillaume Rossolini - Tutoriels PHP";

        header('Content-Type: text/html; charset=utf-8');
        require 'View/header.tpl';
        require 'View/index/index.tpl';
        require 'View/footer.tpl';
    }

}
```

```
<?php
abstract class Controller_Template

{   protected $selfModel;
    protected static $instance;
    public static $db;

    protected function __construct()
    {
    }

    public static function getInstance()
    {
        $class = get_called_class();
        if (!$class::$instance)
        {
            $class::$instance = new $class();
            return $class::$instance;
        }
    }

}
```

L'utilisation de la fonction get_called_class() nous oblige à utiliser au minimum la version 5.3 de PHP.

```php
<?php
class Model_Article extends Model_Template

{   protected $selectById;
    protected $selectByAuthor;
    protected $selectByCategory;

    public function __construct()
    {
        parent::__construct();

        $sql = 'SELECT ar.title, ar.text,
                    au.id AS author_id, au.name AS author_name
                FROM article AS ar
                INNER JOIN author AS au ON ar.author_id = au.id
                WHERE ar.id = ?';
        $this->selectById = Controller_Template::$db->prepare($sql);

        $sql = 'SELECT id, title
                FROM article
                WHERE author_id = ?
                ORDER BY title';
        $this->selectByAuthor = Controller_Template::$db->prepare($sql);

        $sql = 'SELECT ar.id, ar.title,
                    au.id AS author_id, au.name AS author_name
                FROM article AS ar
                INNER JOIN author AS au ON ar.author_id = au.id
                WHERE ar.category_id = ?
                ORDER BY ar.title';
        $this->selectByCategory = Controller_Template::$db->prepare($sql);
    }

    public function getById($id)
    {
        $this->selectById->execute(array($id));
        $articles = $this->selectById->fetchAll();
        if (empty($articles[0]))
        { (
            return array();
        }
        else
        {
            $articles[0]['text'] = utf8_decode($articles[0]['text']);
            return $articles[0];
        }
    }

    public function getByAuthor($id)
    {
        $this->selectByAuthor->execute(array($id));
        return $this->selectByAuthor->fetchAll();
    }

    public function getByCategory($id)
    {
        $this->selectByCategory->execute(array($id));
        return $this->selectByCategory->fetchAll();
    }

}
```

```php
<?php
class Model_Author extends Model_Template

{   public function __construct()
    {
        parent::__construct();
```

```
        $sql = 'SELECT au.id, au.name, COUNT(*) AS nb_articles
                FROM author AS au
                LEFT JOIN article AS ar ON ar.author_id = au.id
                GROUP BY au.id, au.name
                ORDER BY name';
        $this->selectAll = Controller_Template::$db->prepare($sql);

        $sql = 'SELECT name
                FROM author
                WHERE id = ?';
        $this->selectById = Controller_Template::$db->prepare($sql);
    }

}
```

```
<?php
class Model_Category extends Model_Template

{   public function __construct()
    {
        parent::__construct();

        $sql = 'SELECT c.id, c.name, COUNT(*) AS nb_articles
                FROM category AS c
                INNER JOIN article AS a ON a.category_id = c.id
                GROUP BY c.id, c.name
                ORDER BY c.name';
        $this->selectAll = Controller_Template::$db->prepare($sql);
    }

}
```

```
<?php
abstract class Model_Template

{   protected $selectAll;
    protected $selectById;

    public function __construct()
    {
    }

    public function getAll()
    {
        $this->selectAll->execute();
        return $this->selectAll->fetchAll();
    }

    public function getById($id)
    {
        $this->selectById->execute(array($id));
        $row = $this->selectById->fetchAll();
        if (empty($row[0]))
        { (
            return array();
        }
        else
        {
            return $row[0];
        }
    }

}
```

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr-FR" lang="fr-FR">
```

```
<head>
    <title><?php echo html($title); ?></title>
    <meta http-equiv="content-type" content="text/html; charset=utf-8" />
<style type="text/css">
h1 {
    text-align: center;

}
</style>
</head>
<body>
<ul style="text-align: center">
    <li style="display: inline"><a
        href="<?php echo html($_SERVER['HTTP_ROOT']); ?>"
        title="G. Rossolini - Tutoriels PHP">Accueil</a> -</li>
    <li style="display: inline"><a
        href="<?php echo html($_SERVER['HTTP_ROOT']); ?>articles/"
        title="G. Rossolini - Liste des publications">Publications</a> -</li>
    <li style="display: inline"><a
        href="<?php echo html($_SERVER['HTTP_ROOT']); ?>auteurs/"
        title="G. Rossolini - Liste des auteurs">Auteurs</a></li>
</ul>
<h1><?php echo html($title); ?></h1>
```

```
<p style="text-align: center; font-size: 0.7em;">Copyright
    Guillaume Rossolini 2007-<?php echo date('Y'); ?><br />
Contact : <a
    href="mailto:g-rossolini@developpez.com"
    title="E-mail">g-rossolini@developpez.com</a></p>
</body>
</html>
```

```
<div style="text-align: center; font-style: italic">Par <a
    href="<?php echo html($_SERVER['HTTP_ROOT']); ?>auteur/<?php
        echo url_format($article['author_id'].'-'.$article['author_name']);
    ?>" title="Publications de <?php
        echo html($article['author_name']);
    ?>"><?php echo html($article['author_name']); ?></a></div>
<?php
echo nl2br(html($article['text']));
```

```
<?php foreach($categories as $category) : ?>
<h2><?php echo html($category['name']); ?></h2>
<ul>
    <?php foreach($category['articles'] as $article) : ?>
    <li><a href="<?php echo html($_SERVER['HTTP_ROOT']); ?>article/<?php
        echo url_format($article['id'].'-'.$article['title'])
    ?>" title="<?php echo html($article['title']); ?>"><?php
        echo html($article['title']);
    ?></a>, par <a href="<?php
        echo html($_SERVER['HTTP_ROOT']);
    ?>auteur/<?php
        echo url_format($article['author_id'].'-'.$article['author_name']);
    ?>" title=""><?php echo html($article['author_name']); ?></a></li>
    <?php endforeach; ?>
</ul>
<?php endforeach; ?>
```

```
<ul>
    <?php foreach($articles as $article) : ?>
    <li><a href="<?php echo html($_SERVER['HTTP_ROOT']); ?>article/<?php
        echo url_format($article['id'].'-'.$article['title']);
    ?>" title="Article par <?php echo html($author['name']); ?>"><?php
```

```
        echo html($article['title']);
    ?></a></li>
    <?php endforeach; ?>
</ul>
```

```
<ul style="list-style-type: none">
    <?php foreach($authors as $author) : ?>
    <li><a href="<?php echo html($_SERVER['HTTP_ROOT']); ?>auteur/<?php
        echo url_format($author['id'].'-'.$author['name']);
    ?>" title="<?php echo html($author['name']); ?>"><?php
        echo html($author['name']);
    ?></a> (<?php echo (int)$author['nb_articles']; ?> articles)</li>
    <?php endforeach; ?>
</ul>
```

```
Le document <?php echo html($_SERVER['REQUEST_URI']); ?> n'existe pas.
```

```
<h2 style="text-decoration: underline">Présentation</h2>
<p>J'ai suivi une formation dans plusieurs domaines, notamment la communication,
    le développement de logiciels, la gestion de projet,
    <span style="font-style: italic">etc</span>.</p>
<p>Aujoud'hui, je suis un développeur PHP muni de la certification Zend :<br />
<a href="http://www.zend.com/store/education/certification/authenticate.php?ClientCandidateID=ZEND005053&Registra
    title="Zend Certified Engineer Details"><img
        src="http://www.zend.com/images/training/php5_zce_logo.gif"
        alt="Logo ZCE"
        style="border: none"/></a></p>
```

Cette architecture est largement plus robuste et flexible que celle du premier exemple, mais elle n'est pas encore tout à fait parfaite. MVC a tendance à pousser à l'utilisation de nombreux *Design patterns*, notamment *Singleton* et *Registry*. Je n'ai pas mis en place un système trop complexe car notre petit site d'exemple ne le justifie pas, mais par exemple la connexion à la BDD mériterait un pattern **Registry** plutôt que d'être mise dans un contrôleur ou dans un modèle, alors que ni l'un ni l'autre ne sont prévus pour cela.

Par ailleurs, la simplicité fonctionnelle du site actuel ne rend pas justice au *front controller*. Je vous recommande d'étudier le fonctionnement de frameworks MVC et de juger par vous-mêmes de l'approche la plus adaptée à vos projets.

Ne vous fiez pas à la taille des scripts présentés ici pour juger de l'utilité de ce que je viens de vous montrer. Ce site d'exemple est bien trop petit pour que MVC soit utilisé correctement. Pour mieux comprendre l'intérêt du pattern MVC, ajoutez des fonctionnalités à ce site dans chacune des approches vues ici. C'est un très bon exercice de style...

Je ne vais évoquer ici que ce qui touche directement à PHP. L'analyse (**UML** ou autre), les outils de contrôle de versions de code source (**SCM**) et les outils de rapport de bug (*bug tracking system*) ne rentrent donc pas dans le cadre de mon introdution à la démarche qualité.

Je vais vous proposer une méthode pour développer des applications de manière à la fois flexible et qualitative.

Nous allons utiliser trois serveurs par projet :

1. **Développement** ;
2. **Staging / Test** ;
3. **Production / Live**

.

Le premier niveau est le serveur de **Développement**. Il est destiné aux développeurs, eux seuls y ont donc accès.

Une technique flexible pour le développement en équipe, consiste à utiliser un seul serveur pour le développement des applications. Si le développeur est seul dans son équipe, il peut utiliser sa propre machine ; dans le cas contraire, il est préférable d'utiliser une seule machine commune à tous les développeurs. Le serveur Web Apache permet de configurer des hôtes virtuels (VirtualHost) dans son fichier de configuration **httpd.conf** : c'est une manière idéale d'attribuer un serveur Web complet à chaque développeur.

Un module très utile pour configurer simplement Apache dans un tel cas de figure est **mod_macro**, qui permet de définir une macro (un "gabarit") puis de la réutiliser pour chaque développeur. Si vous avez un serveur de noms **DNS**) sur votre réseau, vous pouvez l'utiliser pour lier chaque *VirtualHost* à un développeur.
(

Côté PHP, vous aurez besoin de toutes les extensions du serveur de Production ainsi que d'extensions spécifiques au développement, par exemple **Xdebug** (*cf*. le débogage ci après).

C'est le serveur **Staging** qui contient le dépôt (*repository*) de votre outil SCM (par exemple CVS ou Subversion). Vous pouvez utiliser un utilitaire de "post-commit" pour que le code envoyé par les développeurs soit déployé automatiquement sur le serveur Web de l'environnement Staging.

La configuration de ce serveur Web est strictement identique à celle du serveur de Production : cela permet de s'assurer que la transition Staging/Prod se fera sans problème. L'idée est de faire valider par l'équipe, puis par le client, l'intégralité de l'application avant de la déployer (mise en Production).

Un nom alternatif pour Staging est Test

.

Le serveur de **Production** est la dernière phase. L'application doit avoir été validée par l'équipe puis par le client avant d'être envoyée en Production.

Souvenez-vous de ne jamais rien modifier directement en Production : développez u**patch** en Dev, faites un**commit** dans le SCM en Test puis utilisez le serveur Web de Test pour faire valider le patch. Une fois que vous avez la certitude que le code déposé dans votre outil SCM fonctionne, déployez-le en Production.


Un nom alternatif pour Production est Live


Ne modifiez jamais le code en Production. Déployez à partir de votre dépôt en Test (après validation).

Les tests sont une étape fondamentale de la démarche qualité. Ils permettent de valider le fonctionnement de l'application avant de la soumettre au client.

Les tests unitaires sont les tests les plus élémentaires. Il s'agit de vérifier qu'une fonctionnalité réagit bien comme on s'y attend.

Les tests ne sont pas utiles s'ils ne sont pas automatisés, c'est pourquoi on parle se "suites de tests".

Il existe plusieurs outils PHP pour écrire les tests de vos applications : PHPUnit, SimpleTest, phpt...

N'oubliez pas de rédiger un test à chaque rapport de bug : cela permet de valider à la fois l'existence du bug et sa résolution effective. Intégrer ces tests de bugs à vos suites de tests permet d'éviter les régressions (réapparition d'anciens bugs) lors de la vie d'une application. Un test peut permettre de valider soit qu'une partie de l'application fonctionne correctement, soit qu'un bug n'est pas réintroduit.

Site de PHPUnit : **http://www.phpunit.de/**

Site de SimpleTest : **http://simpletest.sourceforge.net/**

Site de phpt : **http://qa.php.net/write-test.php**


PHPUnit est intégré en standard dans plusieurs EDI du marché. Renseignez-vous sur les EDI avant d'investir votre temps et/ou votre argent dans l'un d'eux.

Le développement orienté par les tests consiste à écrire les tests avant d'écrire le code qui répond à la fonctionnalité testée. C'est le test qui décrit le besoin fonctionnel. Cela permet d'une part de ne pas oublier de tests et, d'autre part, de s'assurer que la fonctionnalité a bien été codée en conformité avec le besoin.

Les tests sont une étape fondamentale de la vie d'une application. Toutefois, c'est à vous de choisir votre stratégie.

La complexité des applications développées aujourd'hui ne permet plus de déboguer un site avec des armées d'**echo** ou **die()**. Ces techniques fonctionnent bien pour de petites applications mais, dès que l'application gagne en complexité et en taille, ajouter des instructions **echo** augmente le risque d'apparition de nouveaux bugs.

Par conséquent, il est primordial de mettre en place une solution permettant de déboguer les scripts à la manière d'un compilateur C++ : avec des points d'arrêt, des états instantanés sur les variables, une exécution pas à pas, *etc* .

C'est possible en installant une extension PHP. Historiquement, la première extension à permettre ce type de débogage est DBG. Bien que son protocole de communication soit encore utilisé, elle est aujourd'hui remplacée par sa concurrente Xdebug, bien plus performante et mise à jour plus fréquemment.

Installer une telle extension permet non seulement de déboguer l'application pas à pas, mais aussi d'en établir un profil afin de déterminer quels scripts consomment le plus de mémoire ou de temps CPU. Le débogage est donc non seulement un outil de développement, mais aussi d'optimisation.

- **APD** (*Advanced PHP Debugger* - extension PECL dont le développement semble arrêté depuis 2004) ;
- **DBG** (extension incompatible avec PHP 5.3 et 6) ;
- **Xdebug** (extension PECL) ;
- **Zend Studio** (application commerciale).

Site : **http://xdebug.org/**

Xdebug est l'extension de débogage PHP la plus efficace en ce moment. Elle est développée par Derick Rethans, elle est disponible pour toutes les versions de PHP, elle est gratuite (licence compatible PHP) et elle s'interface avec n'importe quel EDI capable de débogage avec le protocole DBGp.

```
zend_extension_ts = "full/path/to/php_xdebug.ext"
xdebug.remote_enable = On
xdebug.remote_autostart = On
xdebug.extended_info = On
xdebug.remote_mode = req; "req" ou "jit"
xdebug.profiler_enable = On
xdebug.profiler_enable_trigger = On
xdebug.profiler_output_dir = "path/to/profiling"
xdebug.profiler_output_name = "cachegrind.out.%t"
xdebug.trace_output_dir = "path/to/temp"
xdebug.trace_output_name = "trace.%c"
```

Un **motif de conception** (*design pattern*) est une technique permettant de résoudre un problème standard. Plutôt que d'avoir recours à votre propre solution à un problème, il est préférable de réutiliser un motif déjà identifié et résolu par d'excellents concepteurs de logiciels.

Un exemple simple concerne les variables globales. Tous les enseignements sont formels (avec raison) : il faut éviter les variables globales. Pourtant, il arrive fréquemment d'avoir besoin d'une variable dans plusieurs contextes à la fois, et les problèmes de portée (*scope*) nous mettent des bâtons dans les roues. La solution à ce problème est le pattern **Registry**. Il s'agit d'une classe qui regroupe toutes les variables qui pourraient être globales et qui nous les donne au besoin. Ce pattern repose sur le pattern **Singleton**, qui nous assure d'avoir une seule instance du registre dans tout notre code.

Les motifs de conception reposent très largement sur la POO, il est donc fortement recommandé d'être à l'aise dans ce domaine.

Voici l'un des plus connus des motifs de conception pour sites Web.

Le principe est d'avoir un unique point d'entrée au site Web : le *front controller*. Cela permet de placer la presque totalité de vos scripts en-dehors de la racine du serveur Web, et ainsi de protéger votre application. De plus, cela vous incite à utiliser une technique de réécriture d'URL (URL rewriting), ce qui a de nombreux avantages. Nous avons déjà débattu de tout cela dans ce tutoriel, je ne vais donc pas y revenir.

Le pattern MVC oblige le développeur à séparer la structure de son application en trois types de composants :

- **M pour "modèles"** : Ce sont les classes donnant accès aux données ;
- **V pour "vues"** : C'est la partie de l'application qui se charge de l'affichage des données ;
- **C pour "contrôleurs"** : Ces classes s'occupent de charger les modèles adéquats, d'appliquer les traitements nécessaires et d'envoyer aux vues les données demandées.

Le pattern Singleton permet de s'assurer qu'il n'existe qu'une seule instance de la classe. C'est utile dans certaines situations précises, comme par exemple certains composants d'architecture **MVC** ou un pattern **Registry**.

L'idée du Singleton est de rendre le constructeur **protected** afin d'empêcher l'instanciation de la classe, et d'utiliser une méthode statique pour construire une instance unique. Par convention, cette méthode s'appelle **getInstance()**.

```php
<?php
class MyClass

{   protected $instance;

    protected function __construct(){}

    public static function getInstance()
    {
        if (empty(self::$instance))
        { (     (
            self::$instance = new self();
        }
```

```
        return self::$instance;
    }

}
```

Au lieu d'instancier cette classe (opérateur "new"), le programmeur appelle la méthode **getInstance()** .

```
<?php
$objet = new MyClass(); //utilisation standard, ne peut pas fonctionner ici
```

```
<?php
$objet = MyClass::getInstance(); //pattern Singleton
```

Si vous êtes certain de n'avoir toujours besoin que d'une seule connexion à la base de données dans vos scripts, vous pouvez par exemple utiliser ce pattern pour implémenter votre classe de connexion. Un autre exemple peut être votre classe de configuration.

Ne **sur** estimez jamais votre capacité à résoudre seul les problèmes auxquels vous faites face. D'autres développeurs y ont été confrontés avant vous et ils ont eu tout le temps de parvenir à des solutions à la fois élégantes, simples et robustes. Ayez l'humilité d'utiliser leurs solutions plutôt que d'essayer d'y parvenir vous-même car, ayant moins d'expérience, vos solutions ne seront probablement pas aussi efficaces. Vous aurez tout le temps de révolutionner le monde du développement logiciel lorsque vous aurez suffisamment de recul pour pouvoir vous permettre de critiquer l'existant.

De nombreux autres motifs sont disponibles. Nous ne pouvons pas tous les expliquer ici car c'est l'objectif d'ouvrages bien plus complets.

Lorsque vous commencez à développer, il est préférable de simplement apprendre à utiliser le langage. Cependant, avec le temps, vous vous rendez compte que vous résolvez sans arrêt les mêmes problèmes et que vous réutilisez (recopiez) des composants d'une application à l'autre. Cela signifie que vous avez développé une sorte de **framework** utilisé par 0% des autres développeurs.

De plus, se plonger dans le code écrit (sans framework) par un autre développeur est souvent un cauchemard parce qu'il n'utilise pas les mêmes conventions que vous. Maintenir les applications des autres développeurs est donc problématique à cause de la courbe d'apprentissage du **framework** spécifique à chaque application (également utilisé par environ 0% des développeurs).

Une solution à tous ces problèmes est d'adopter des composants standards, écrits et utilisés par une communauté de développeurs. Cela ne vous immunise pas totalement dans le cas où vous devez maintenir du code développé avec un autre framework, mais les probabilités de rencontrer un framework inconnu passent de 100% à un nombre bien plus raisonnable. Un framework regroupe des composants standards, choisis spécialement et adaptés pour fonctionner parfaitement ensemble, et propose des conventions de codage afin de réduire au maximum les différences dans les styles de programmation des développeurs de la communauté.

Utiliser un framework ne produira pas un site Web en deux clics (ce n'est pas un **CMS**). En revanche, il vous permettra de produire des sites uniques tout en simplifiant les problématiques standards.

Les frameworks sont généralement classés comme "glue" ou bien "full-stack". Les premiers sont simplement des ensembles flexibles de composants, tandis que les autres obligent le développeur à suivre une ligne de conduite bien précise. Les deux catégories ont leurs avantages, et le choix d'un framework par rapport à un autre dépend du cahier des charges de chaque projet.

Site : **http://cakephp.org/**

Tutoriel rapide : **http://manual.cakephp.org/appendix/blog_tutorial**

CakePHP est un framework de type "full-stack".

Il adhère au principe "*convention over configuration*" qui prône les conventions de codage plutôt que la configuration. Cela permet de réduire les temps de développement, mais enlève au développeur une marge de liberté sur la structure de sa BDD ou de son code, voire dans certains cas sur l'aspect de son site.

La documentation de CakePHP suit de bons principes et le champ sémantique homogène (cookies, bakery...) dénote une bonne ambiance au sein de la communauté.

Liste des conventions de CakePHP : **http://manual.cakephp.org/appendix/conventions**

Site : **http://ezcomponents.org/**

Vue d'ensemble : **http://ezcomponents.org/overview**

eZ Components est la bibliothèque développée par eZ Systems et utilisée dans eZ Publish. Ce n'est pas un framework à proprement parler puisqu'il ne se présente pas en tant que tel, mais on peut probablement en parler comme d'un framework de type "glue". eZ Components a le soutien de Derick Rethans, l'un des participants majeurs à PHP (internals, xdebug...).

Tutoriels eZ Publish : **http://php.developpez.com/cours/?page=scripts#ezp**

Site : **http://pear.php.net/**

PEAR est l'un des plus anciens frameworks "glue" pour PHP. Ils proposent des conventions de codage depuis de nombreuses années, et certaines ont été reprises par d'autres projets depuis lors. C'est aujourd'hui une bibliothèque de composants parfois rendus obsolètes par les frameworks concurrents.

Le groupe PEAR a su standardiser la diffusion d'extensions et de composants, ainsi que distribuer un outil qui n'est pas sans rappeler **apt-get** (Debian GNU/Linux).

PEAR est aujourd'hui connu principalement pour sa démarche qualité et pour ses outils de distribution (**go-pear** *etc*.).

Site : **http://www.symfony-project.org/**

Framework de type "full-stack".

Le projet **symfony**, d'origine française, s'appuie sur l'expérience de ses développeurs en matière de développement de sites Internet. Toutes les problématiques rencontrées de manière récurrente par les développeurs du framework ou par ses utilisateurs sont répertoriées, mises en facteur et éventuellement incluses dans une version ultérieure.

L'un des gros points forts de symfony est sa documentation : il existe un livre, disponible à la vente ou bien en téléchargement gratuit, également consultable en ligne et traduit en français.

Le projet symfony a le soutien de l'entreprise française SensioLabs.

Tutoriels symfony : **http://php.developpez.com/cours/?page=frameworks#symfony**

Site : **http://framework.zend.com/**

Framework de type "glue" à tendances "full-stack".

Zend, qui fournit notamment le Zend Engine depuis de nombreuses versions de PHP, a commencé en 2007 à diffuser son propre framework. L'idée est de diffuser un framework de très haute qualité en matière de développement de sites Web : best practices, design patterns, OOP, sécurité...

Le framework suit un procesus communautaire de développement, chaque proposition est étudiée avec soin puis soumise à validation par l'ensemble de la communauté, la documentation (traduite dans de nombreuses langues) est révisée fréquemment, des tests unitaires sont systématiquement exigés, *etc*

.

Le projet Zend Framework est soutenu par Zend Technologies.

Tutoriels Zend Framework : **http://zend-framework.developpez.com/cours/**

Au risque de me répéter...

Ne **sur** estimez jamais votre capacité à résoudre seul les problèmes auxquels vous faites face. D'autres développeurs y ont été confrontés avant vous et ils ont eu tout le temps de parvenir à des solutions élégantes. Ayez l'humilité d'utiliser leurs solutions plutôt que d'essayer d'y parvenir vous-même car, ayant moins d'expérience, vos solutions ne seront probablement pas aussi efficaces. Vous aurez tout le temps de révolutionner le monde du développement logiciel lorsque vous aurez suffisamment de recul pour pouvoir vous permettre de critiquer l'existant.

Une fois que votre application fonctionne parfaitement, vous pouvez essayer d'optimiser son temps d'exécution ou sa consommation de mémoire. Cependant, prenez garde à ne pas tomber dans l'eccès.

Optimiser le code PHP est généralement un faux problème pour plusieurs raisons :

- Ce qui prend le plus de temps n'est pas le code PHP mais les accès réseau (requêtes BDD, webscraping) ou disque (includes) ;
- Un outil de cache d'opcode peut se charger d'optimiser votre code PHP à votre place (dans une certaine mesure) ;
- Optimiser le code à outrance produit habituellement du code très complexe, donc difficile à maintenir.

La mise en cache vous permet de ne pas éxécuter de nouveau du code qui a déjà été exécuté récemment. Cela comporte une notion de temporalité.

Prenez l'exemple d'un site d'actualités dans lequel la page d'accueil présente les actualités les plus récentes. Chaque visiteur de la page d'accueil oblige PHP à envoyer une requête SQL par le réseau, attendre la réponse du SGBD, récupérer les données depuis le socket ouvert pour l'occasion, les traiter puis les afficher. Sachant que le taux de visites est très nettement supérieur à la fréquence de mise à jour des actualités, la très grande majorité des requêtes sont redondantes.

Admettons que notre site d'actualités génère 500 visites quotidiennes mais qu'il y a tout juste 10 nouvelles actualités par jour. Cela suppose 10 mises à jour de la page d'accueil contre 500 visites : entre 490 et 500 de ces visites quotidiennes obligent donc PHP à effectuer inutilement les traitements résumés ci-dessus...

Une solution de mise en cache serait par exemple de générer le code HTML complet de la page d'accueil à chaque mise à jour de la BDD. Ainsi, aucune requête inutile n'est envoyée au SGBD. Cependant, cette technique n'est pas raisonnable pour la totalité du site à cause de la trop grande quantité potentielle de contenu à compiler à l'avance. De plus, le contenu d'une URL peut changer en fonction de l'internaute. Il faut donc avoir recours à des techniques plus subtiles, et c'est ici que les outils de mise en cache deviennent intéressants.

Exemples d'outils :

- **APC** (extension PECL, incluse en standard à partir de PHP6) ;
- **eAccelerator** (extension) ;
- **memcache** (extension PECL)
- **Zend_Cache** (composant d'un framework) ;
- **Zend Platform** (application commerciale) ;
- *etc*.

Dans certaines situations, il peut être intéressant de compiler son code PHP en **bytecode**. Nous pourrions par exemple avoir besoin de distribuer notre application sous forme d'exécutable afin d'éviter aux utilisateurs de devoir installer PHP ; une autre raison est simplement de distribuer une application PHP en protégeant son code source (obfuscation).

Exemples d'outils :

- **bcompiler** (extension PECL) ;
- **ionCube PHP Encoder** (application commerciale) ;
- **Winbinder** (extension pour Windows) ;
- **Zend Guard** (application commerciale) ;
- *etc*.

N'oubliez pas que compiler PHP peut améliorer les temps de réponse, en particulier si vous êtes sous Windows puisque les exécutables distribués sont compilés contre d'anciennes DLL. Certaines personnes ont remarqué jusqu'à 30% d'amélioration.

Compiler PHP vous permet aussi d'activer uniquement les extensions dont vous avez besoin, ce qui réduit l'empreinte mémoire.

Je ne vais pas m'étendre sur le sujet, puisqu'il y a beaucoup de concepts qui dépassent le cadre des connaissances requises d'un développeur PHP.

Pour compiler PHP, vous devez également compiler ses dépendances. Certaines sont obligatoires (distribution standard : *bindlib*, *libiconv* et *libxml2*), d'autres sont optionnelles (extensions). Si vous souhaitez compiler PHP pour l'utiliser comme module Apache, vous aurez également besoin de compiler Apache.

Si vous êtes sous un système Unix ou dérivé, vous êtes probablement habitué à compiler les applications avant de les utiliser.

En revanche, si vous êtes sous Windows, sachez que les fichiers binaires distribués sur php.net sont compilés avec Microsoft Visual Studio 6. Ce compilateur n'étant plus distribué par Microsoft, il devient difficile de se le procurer mais PHP peut être compilé à l'aide d'une version plus récente de Visual Studio. Cependant, il n'est pour le moment pas possible de compiler Apache avec une version plus récente que Visual Studio 6. En résumé, pour compiler PHP comme module Apache, vous avez besoin de Visual Studio 6.

Cela pose certains problèmes outre la disponibilité du compilateur : les nouvelles versions de Visual Studio sont plus efficaces, les binaires compilés sont plus rapides, plus fiables, *etc*. Il n'y a pour le moment pas de solution à ce problème, seul VS 6 permet de compiler PHP comme module Apache.

Elizabeth Marie Smith est en train de chercher à compiler PHP et toutes ses dépendances avec Visual Studio, version 2005 au minimum. Si vous souhaitez l'aider, rendez-vous sur son blog : http://elizabethmariesmith.com/

Voici un sujet à la fois vaste et primordial. Il n'est pas demandé à un développeur PHP de savoir développer une extension pour PHP, mais il faut savoir que c'est possible.

PHP étant écrit en langage C, ses extensions suivent naturellement son exemple. Une extension PHP utilise l'API du Zend Engine. Pour rappel, PHP 5 utilise le Zend Engine version 2, tandis que PHP 6 en utilisera la version 3.

Une extension écrite en langage C donne de meilleures performances qu'un script écrit en langage PHP. Dans certaines situations, il peut être intéressant de compiler une extension plutôt que d'écrire un script.

Pour en savoir plus : **PHP at the Core: A Hacker's Guide to the Zend Engine**

Le site officiel de PHP utilise CVS pour la majorité des contributions. Dans le jargon, avoir un compte CVS sur php.net s'appelle "avoir du karma". Les mainteneurs accordent un accès CVS à ceux qui apportent des patches ou du nouveau code.

Bien que PHP lui-même soit écrit en langage C, vous n'avez pas nécessairement besoin de savoir programmer en langage C pour aider la communauté. En fait, vous n'avez pas besoin de savoir programmer du tout... Si vous connaissez le langage C, vous pouvez apporter votre aide au développement du core ou des extensions ; si vous connaissez seulement PHP, vous pouvez aider à l'écriture de tests ; dans tous les cas, vous pouvez aider à la documentation (anglaise ou traductions) et à l'éducation des autres développeurs.

Les personnes ayant du karma sont souvent considérées comme des mentors.

Lorsque vous pensez avoir détecté un bug dans le fonctionnement de PHP, vérifiez à deux fois. Le but est de ne pas faire un rapport de bug en double ou factice, car cela donne trop de travail de modération au PHP Group et car cela ralentit le développement du langage.

Vous devriez suivre ces étapes :

1. Vérifier dans le *Bug System* de php.net : **http://bugs.php.net/** ;
2. Chercher dans les listes de diffusion une ancienne mention de ce bug (c'est peut-être une *feature* que vous ne connaissiez pas) : **http://php.net/mailing-lists.php** ;
3. Envoyer un e-mail à la liste de diffusion concernée, en précisant que vous n'avez rien trouvé ni dans le système de rapport de bugs ni dans les listes ;
4. Lorsque quelqu'un ayant du "karma" vous répond que vous devriez soumettre un rapport de bug, allez-y.

Si vous souhaitez résoudre un bug, proposez simplement un patch à la liste de diffusion adéquate : **http://php.net/mailing-lists.php**

N'envoyez pas de message inutile aux listes de développement et à Internals car elles sont prévues essentiellement pour faire avancer PHP. Toute question doit se faire sur la liste php-general, réservez les autres listes pour les patches.

Le code source de PHP est testé au moyen de scripts PHP portant (par convention) l'extension ".phpt", un format proposé par la "PHP Quality Assurance Team" : **http://qa.php.net/**

.

Les statistiques sur la couverture du code par les tests est visible ici : **Test and code coverage analysis**

.

Les tests **phpt** sont apparus avec la version 5 de PHP. Le groupe QA s'est construit à l'initiative de Stefan Esser, qui l'a depuis quitté pour des raisons personnelles.

Tous les tests sont disponibles depuis CVS mais aussi dans les archives téléchargées du code source de PHP.

Voici la syntaxe complète d'un test :

```
--TEST--
--CREDITS--
--SKIPIF--
--POST--
--GZIP_POST--
--DEFLATE_POST--
--POST_RAW--
--GET--
--COOKIE--
--STDIN--
--INI--
--ARGS--
--ENV--
--FILE--
--FILEEOF--
--FILE_EXTERNAL--
--EXPECT--
--UEXPECT--
--EXPECTF--
--UEXPECTF--
--EXPECTREGEX--
--UEXPECTREGEX--
--REDIRECTTEST--
--HEADERS--
--EXPECTHEADERS--
--CLEAN--
===DONE===
```

Chaque section est indiquée par un marqueur**--SECTION--** sur une ligne vide. Le marqueur est suivi de son contenu, par exemple :

```
--TEST--
ZE2 An abstract method may not be called
--SKIPIF--
<?php if (version_com parezend_version(), '2.0.0-dev', '<')) die('skip ZendEngine 2 needed'); ?>
--FILE--                (
<?php

abstract class fail {
 abstract function show();

}
class pass extends fail {
 function show() {
  echo "Call to function show()\n";
 }
 function error() {
  parent::show();
 }

}
$t = new pass();
$t->show();
$t->error();

echo "Done\n"; // shouldn't be displayed
?>
--EXPECTF--
Call to function show()

Fatal error: Cannot call abstract method fail::show() in %s on line %d
```

L'effet d'un test est facile à deviner d'après le code du test. Il s'agit de tests unitaires extrêmement simples, ayant le minimum possible de code permettant de vérifier qu'une fonctionnalité précise donne le résultat attendu,*etc*. Dans le

cas ci-dessus, le test s'assure qu'une méthode abstraite ne peut pas être appelée par du code PHP. Si le test indiquait le mot "Done", alors il faudrait revoir (dans le code C) la manière dont les classes abstraites sont gérées par PHP.

```
cd path/to/php
php run-tests.php
```

La documentation anglaise est la première ressource. Ce qui n'est pas encore documenté doit l'être d'abord en anglais, puis éventuellement traduit dans d'autres langues si des volontaires se proposent. Toutes les traductions obsolètes (non mises à jour) ont été retirées du site officiel ainsi que des miroirs, afin d'éviter d'orienter les lecteurs dans de mauvaises directions.

La documentation de PHP est écrite au format **DocBook**, un schéma XML standard qui permet de générer la documentation finale dans le format voulu (gabarits) grâce à certaines transformations XSLT.

À l'heure de la rédaction de ce cours, un outil est en cours d'écriture et d'adoption par le *PHP Group* afin de faciliter les participations à la traduction de la documentation PHP : **PHPDOC Online XML Editing Tool** (attention, ce lien peut changer à tout moment puisque l'outil est encore en bêta). Les sources de cet outil sont disponiles sur le **CVS officiel** .

La documentation finale est compilée à l'aide de **PhD** .

Enseigner PHP est sans doute la manière la plus simple d'aider la communauté. C'est aussi très complexe, puisqu'il faut connaître tous les aspects du langage et en suivre les évolutions.

De très nombreuses communautés naissent, grandissent et disparaissent avec le temps. Autant la diversité a du bon, autant au jourd'hui il faut se faire une raison : créer une nouvelle communauté autour de PHP n'aide en rien la communauté au sens large. Il est nettement plus utile de rejoindre une communauté existante et d'y joindre vos efforts, que d'essayer d'en faire naître une nouvelle qui rentrera "en concurrence" avec les autres (au sens où elle partagera le temps des lecteurs entre les diverses communautés).

Outre la création d'une communauté, vous pouvez tout simplement rédiger des billets de blog, des tutoriels, des livres, *etc*. Toutes ces ressources, bien rédigées, organisées et référencées, sont extrêmement utiles aux développeurs de tous niveaux.

Historique

Historiquement, PHP est un langage de script.

Aujourd'hui, PHP conserve cet héritage mais il permet également de construire des applications très complexes, d'un niveau de qualité tel qu'une entreprise peut l'exiger. Il existe des frameworks, des standards de codage, des extensions, une communauté, *etc*.

PHP n'est pas adapté pour toutes les situations mais, dans un environnement Web, c'est aujourd'hui un excellent choix face à d'autres technologies comme Java, .NET ou ColdFusion.

Le PHP Group

Tous les sites du PHP Group sont recencés ici : **PHP.net: A Tourist's Guide**.

Guillaume Rossolini

Guillaume Rossolini est développeur et formateur pour sa société Alveod.