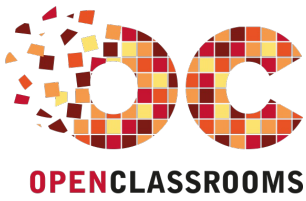
Votre site PHP (presque) complet : architecture MVC et bonnes pratiques

Par Savageman



www.openclassrooms.com

Sommaire

Votre site PHP (presque) complet : architecture MVC et bonnes pratiques

Par  Savageman

Mise à jour : 07/05/2010
Difficulté : Intermédiaire

Envie de progresser en PHP ?
Envie d'apprendre et de mettre en pratique une architecture MVC à travers la réalisation de différents modules* ?
De connaître des bonnes pratiques et avoir un code sécurisé ?

Tels sont les **objectifs** de ce tutoriel, alors bienvenue, et j'espère que le voyage vous plaira !

* Des exemples de modules sont : un espace membre, un livre d'or, une gestion de news, un album photo, un forum, etc. (la liste est longue !). À noter que ces exemples ne seront pas tous traités dans le tutoriel, ça serait beaucoup trop long !

Mes remerciements chaleureux à mes bêta-testeurs tarzoune, mrjay42, 11 et yøshï pour leurs commentaires et suggestions. Un remerciement particulier à `Haku, ex-membre impliqué et auteur de la majeure partie du chapitre "Bonnes pratiques à appliquer". Je remercie également ptipilou pour la correction de ce tutoriel, ainsi que Zopieux qui l'a validé.

Historique du tutoriel

13 janvier 2009 : deuxième validation avec corrections de nombreux bugs et ajout de la connexion automatique. Les fonctions combinaison_connexion_valide() et lire_infos_utilisateur() ont été mises à jour, pensez à les modifier !
7 janvier 2009 : première validation contenant toute l'introduction (avant-tutoriel), l'espace membre (inscription, (dé)connexion) et le profil (affichage, modification).
21 décembre 2008 : envoi à la zCorrection.
décembre 2008 : relectures et bêta-test.
novembre 2008 : rédaction du tutoriel.

Partie 1 : Introduction : l'avant-tutoriel

Cette partie :

- définit les termes importants à connaître ;
- présente la mise en place de l'architecture MVC et de l'arborescence pour le site ;
- explique les concepts importants de sécurité à maîtriser et les bonnes pratiques à appliquer ;
- et présente les librairies qui seront utilisées.

Avant-propos, comment fonctionne ce tutoriel

Introduction, objectifs et pré-requis

Dans la mesure du possible, les parties sont d'une difficulté croissante : la première partie est la plus facile, tandis que la dernière est la plus difficile. Les parties du tutoriel doivent êtres lues dans l'ordre, car chacune a comme pré-requis **l'ensemble des parties précédentes**. Par exemple, pour pouvoir faire le forum, il vous faut savoir faire le livre d'or.

Ce tutoriel est plus difficile que celui de M@teo21, mais les modules abordés - sous forme de TP quand c'est possible - sont plus approfondis ! Il est donc indispensable d'avoir déjà lu le tutoriel officiel avant de tenter celui-ci (mis à part la partie sur les expressions régulières qui nous sera inutile).

Il est donc adressé à des novices, mais des pré-requis de POO sont indispensables à sa compréhension. Rassurez-vous, il n'est pas nécessaire de savoir coder des classes en POO (ça sera déjà fait pour vous), seulement de comprendre leur fonctionnement et de savoir les utiliser. Le but n'est pas d'entrer dans le détail à ce niveau. Dans ce tutoriel, 2 classes principales seront utilisées :

- une pour l'accès à la base de données SQL (mySQL dans le cas présent) : nous utiliserons PDO, qui définit une excellente interface pour accéder à une BDD depuis PHP 5.1. Il est important de savoir que c'est le futur standard utilisé par PHP ; en effet, dans PHP 6, les fonctions mysql_* et mysqli_* (ou plus généralement les extensions mysql et mysqli) disparaîtront au profit de PDO. Le tutoriel de Draeli sur PDO résume bien la situation et sera largement suffisant dans le cadre de ce tutoriel !
- une pour la gestion des formulaires : c'est une classe que j'ai codée et qui simplifie largement la création et surtout le traitement des formulaires ! Je peux vous assurer que vous aurez beaucoup moins de lignes de code à écrire en l'utilisant ! Je la présente rapidement à la fin de la partie afin que vous sachiez vous en servir et que vous ayez un aperçu plus en détails des fonctionnalités de celle-ci.

Une troisième classe nommée Image fait également son apparition (code source disponible ici). Elle permet simplement de lire, redimensionner et sauvegarder des images aux formats JPG, PNG et GIF sans se soucier des fonctions à utiliser pour lire et enregistrer l'image.

Enfin, au niveau de la base de données, les requêtes préparées seront utilisées tout au long du tutoriel. Si vous ne savez pas ce que c'est, une lecture de la partie 5 du tutoriel de Draeli sur la question vous sera d'une grande utilité ! En résumé, la structure de la requête est stockée sur le serveur, puis le programmeur fait pointer les données qu'il souhaite sur la requête. Ainsi, la structure de la requête est figée avant l'exécution et les données fournies ne peuvent en aucun cas la modifier, ce qui rend les injections SQL tout simplement **impossibles**.

PHP 5.1 au minimum est nécessaire pour suivre ce tutoriel, étant donné que nous allons utiliser PDO (disponible depuis PHP 5.1) ainsi que ma librairie Form (fonctionnant avec PHP 5+).

N'essayez pas de lire la suite du tutoriel si vous n'avez pas lu le tutoriel officiel ou si vous ne connaissez rien à PDO ou aux requêtes préparées. Vous ne comprendriez tout simplement rien aux codes suggérés...

Organisation du code et des fichiers

Ce chapitre décrit l'arborescence que nous allons utiliser pour la réalisation de notre site et de nos modules.

Tous les mots en **bleu et gras** présentent du vocabulaire potentiellement inconnu pour vous. Ces définitions sont très importantes : si donc vous ne comprenez pas un de ces mots, relisez plusieurs fois le paragraphe en question !

Modules et actions

Chaque partie de ce tutoriel propose la création d'un **module** (exemple : un livre d'or, une gestion de news), qui seront chacun

constitués d'**actions** (exemples : lire une news, poster un message dans le livre d'or).

Dans cette optique, il nous faut une hiérarchie de répertoires capable de bien organiser notre code source à ce niveau. Voici ce que je vous propose et ce que je vais utiliser durant tout le tutoriel.

- Nous aurons un dossier ~/modules/ qui contiendra tous nos modules.
- Chaque module sera contenu dans un dossier portant le nom du module et situé dans ~/modules/.
- Chaque action sera placée dans un fichier spécifique dans le dossier du module, portant le nom de l'action.
- Toutes ces pages ne seront pas appelées directement, mais par la page principale index.php fournie. Toutes les requêtes passeront par cette page qui s'occupera de charger le bon module et la bonne action de manière sécurisée.

Le MVC, ou séparation des couches

Avant d'aller plus loin, une petite présentation du MVC est indispensable. **MVC signifie Modèle / Vue / Contrôleur.** C'est un découpage couramment utilisé pour développer des applications web.

Chaque action d'un module appartient en fait à un **contrôleur**. Ce contrôleur sera chargé de générer la page suivant la requête (HTTP) demandée par l'utilisateur. Cette requête inclut des informations comme l'URL, avec ses paramètres GET, des données POST, des COOKIES, etc. Un module peut être divisé en plusieurs contrôleurs, qui contiennent chacun plusieurs actions.

Pour générer une page, un contrôleur réalise presque systématiquement des opérations basiques telles que lire des données, et les afficher. Avec un peu de capacité d'abstraction, on peut voir deux autres couches qui apparaissent : une pour gérer les données (notre **modèle**) et une autre pour gérer l'affichage des pages (notre **vue**).

Le **modèle** : une couche pour gérer les données, ça signifie quoi ? Ça signifie qu'à chaque fois que nous voulons créer, modifier, supprimer ou lire une donnée (exemple, lire les informations d'un utilisateur depuis la base de données MySQL), nous ferons appel à une fonction spécifique qui nous retournera le résultat (sous forme d'un tableau généralement). Ainsi, nous n'aurons AUCUNE requête dans notre contrôleur, juste des appels de fonctions s'occupant de gérer ces requêtes.

La **vue** : une couche pour afficher des pages ? Cela signifie tout simplement que notre contrôleur n'affichera JAMAIS de données directement (via echo ou autre). Il fera appel à une page qui s'occupera d'afficher ce que l'on veut. Cela permet de séparer complètement l'affichage HTML dans le code. Certains utilisent un moteur de templates pour le faire, nous n'en aurons pas besoin : l'organisation des fichiers se suffit à elle-même.
Ceci permet de séparer le travail des *designers* et graphistes (s'occupant de créer et de modifier des vues) et celui des programmeurs (s'occupant du reste).

En externalisant les codes du modèle et de la vue de la partie contrôleur, vous verrez que le contenu d'un contrôleur devient vraiment simple et que sa compréhension est vraiment aisée. Il ne contient plus que la logique de l'application, sans savoir comment ça marche derrière : par exemple, on voit une fonction verifier_unicite_pseudo(), on sait qu'il vérifie que le pseudo est unique, mais on ne s'encombre pas des 10 lignes de codes SQL nécessaires en temps normal, celles-ci faisant partie du modèle. De même pour le code HTML qui est externalisé.

À propos des librairies

Un ensemble de fonctions servant dans un but (plus ou moins) précis peut être mis à disposition sous la forme d'une **librairie** (également appelée bibliothèque de fonctions). Une librairie contient du code utile que l'on ne désire pas avoir à réécrire à chaque fois (car comme tout le monde le sait, un programmeur est fainéant !).

Par exemple, la classe que nous utiliserons pour la gestion des formulaires (pour rappel : Form) est une librairie externe, et sera donc placée dans le dossier ~/libs/form.php (créé pour l'occasion). Elle permet de ne pas réécrire le code de validation du formulaire à chaque fois, ce qui fait économiser un sacré paquet de lignes de code !

J'avais dit que nous utiliserons PDO pour l'accès à la BDD MySQL, ce qui n'est pas tout à fait juste. Nous utiliserons en fait une version de PDO implémentant le design pattern *Singleton*, je l'ai appelée PDO2. Ceci permet de récupérer notre objet PDO à la demande (lorsque nous voulons faire une requête), celui-ci se connectant à la base de données lors de son premier appel. Ainsi, la connexion à la BDD est gérée de façon transparente et se fait uniquement lorsqu'on en a besoin ! Bien que ça ne soit pas vraiment une librairie (mais une interface d'accès à la BDD), nous la placerons quand même dans ~/libs/pdo2.php.

Architecture finale utilisée

Je vous ai parlé tout à l'heure de la page index.php par laquelle passeront toutes les requêtes de l'utilisateur. Cette page agit en fait comme un contrôleur frontal et redirige la requête de l'utilisateur vers le bon contrôleur du bon module. Le lecture du tutoriel de vincent1870, "Initiation au modèle MVC, une manière d'organiser son code" à propos du MVC est un très bon approfondissement à ce paragraphe, et il est vivement conseillé de le lire si vous avez des difficultés avec ces notions.

Il faut également définir un comportement lorsqu'aucun module ni action n'est spécifié. J'ai choisi de charger la page ~/global/accueil.php qui contiendra le code de la page d'accueil (bizarrement :p).

En partant du constat qu'un modèle peut servir dans plusieurs modules (exemple : les informations d'un utilisateur sont utiles à la fois pour les news et le forum), on placera les fichiers de modèles dans un dossier séparé, bien qu'ils seront toujours regroupés par "modules".
Ceci permet une meilleure modularité de l'ensemble du code, ce qui permet par exemple d'utiliser un modèle en rapport avec les news dans le module membre pour afficher les news proposées par un utilisateur dans son profil.
Cela permet également un meilleur découpage des modèles et peut éviter d'inclure des fonctions utiles seulement dans une action d'un module. Par exemple, la fonction pour vérifier l'unicité d'une adresse e-mail lors de l'inscription est utile uniquement lors de l'inscription. Nous la mettrons donc dans un fichier séparé pour éviter de surcharger le modèle pour la table "membres".

Les vues, quant à elles, seront placées dans un dossier /vues/ présent dans le dossier ~/modules/nom_du_module/. Je ne ferai pas plus de commentaire sur ce choix.

Squelette d'une page vierge

Une page web sémantique a besoin d'un code HTML minimal pour son bon fonctionnement, à savoir les balises **<html> <head> <title> <body>** et parfois quelques **<meta>** . Ce code HTML étant global car nécessaire à tous les modules, nous allons les créer dans un nouveau dossier ~/global/, afin de laisser index.php seul à la racine.
Nous avons besoin d'au moins 2 fichiers dans ce dossier ~/global/ : haut.php et bas.php. J'en utilise un troisième : menu.php que j'inclus depuis haut.php. Ceci permet d'avoir accès au menu rapidement, car on aura à modifier cette page au fur et à mesure de l'avancement dans le tutoriel !

Enfin, deux derniers fichiers sont nécessaires. Il s'agit de global/init.php et global/config.php. Le premier fichier (d'initialisation) contiendra dans un premier temps une suppression des guillemets magiques (on explique plus tard pourquoi c'est nécessaire) et l'inclusion du second fichier (de configuration). Le second fichier définit les constantes de notre site. Les informations de connexion à la base de données ainsi que des chemins vers les modèles et vues à inclure seront présents dans ce fichier de configuration.
Le fichier d'initialisation sera appelé sur toutes les pages depuis index.php, d'où son emplacement dans ~/global/.

Voici les codes sources des différents fichiers :

Code : PHP - ~/index.php

```php
<?php

// Initialisation
include 'global/init.php';

// Début de la tamporisation de sortie
ob_start();

// Si un module est specifié, on regarde s'il existe
if (!empty($_GET['module'])) {

 $module = dirname(__FILE__).'/modules/'.$_GET['module'].'/';

 // Si l'action est specifiée, on l'utilise, sinon, on tente une
action par défaut
 $action = (!empty($_GET['action'])) ? $_GET['action'].'.php' :
'index.php';

 // Si l'action existe, on l'exécute
 if (is_file($module.$action)) {

  include $module.$action;

 // Sinon, on affiche la page d'accueil !
```

```
 } else {

  include 'global/accueil.php';
 }

// Module non specifié ou invalide ? On affiche la page d'accueil !
} else {

 include 'global/accueil.php';
}

// Fin de la tamporisation de sortie
$contenu = ob_get_clean();

// Début du code HTML
include 'global/haut.php';

echo $contenu;

// Fin du code HTML
include 'global/bas.php';
```

La **tamporisation de sortie** permet de "capturer" toutes les sorties effectuées entre ob_start() et ob_get_clean(). Ceci permet de faire l'affichage de la page ~/global/haut.php après l'exécution des scripts de la page, qui sont ainsi capables de modifier des informations dont cette page aurait besoin (par exemple les sessions, ou même le titre de la page tout simplement).

Une **condition ternaire** est utilisée à cette ligne : $action = (!empty($_GET['action'])) ? $_GET['action.php'].'.php' : 'index.php';. Une condition ternaire s'utilise ainsi : (condition) ? (valeur si vrai) : (valeur si faux). Ainsi, si (!empty($_GET['action'])) est vrai, $action prendra pour valeur $_GET['action.php'].'.php', sinon $action prendra pour valeur 'index.php'.

Code : PHP - ~/global/init.php

```
<?php

// Inclusion du fichier de configuration (qui définit des
constantes)
include 'global/config.php';

// Désactivation des guillemets magiques
ini_set('magic_quotes_runtime', 0);
set_magic_quotes_runtime(0);

if (1 == get_magic_quotes_gpc())
{
 function remove_magic_quotes_gpc(&$value) {

  $value = stripslashes($value);
 }
 array_walk_recursive($_GET, 'remove_magic_quotes_gpc');
 array_walk_recursive($_POST, 'remove_magic_quotes_gpc');
 array_walk_recursive($_COOKIE, 'remove_magic_quotes_gpc');
}

// Inclusion de Pdo2, potentiellement utile partout
include CHEMIN_LIB.'pdo2.php';
```

Si vous ne savez pas ce que sont les guillemets magiques ni pourquoi les désactiver, c'est expliqué dans la partie suivante : *Bonnes pratiques à appliquer > L'enfer des guillemets magiques*

Code : PHP - ~/global/config.php

```
<?php
```

```php
// Identifiants pour la base de données. Nécessaires a PDO2.
define('SQL_DSN',      'mysql:dbname=tutoriel;host=localhost');
define('SQL_USERNAME', 'root');
define('SQL_PASSWORD', '');

// Chemins à utiliser pour accéder aux vues/modeles/librairies
$module = empty($module) ? !empty($_GET['module']) ? $_GET['module']
: 'index' : $module;
define('CHEMIN_VUE',    'modules/'.$module.'/vues/');
define('CHEMIN_MODELE', 'modeles/');
define('CHEMIN_LIB',    'libs/');
```

Code : PHP - ~/global/haut.php

```php
<!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">

<head>

 <meta http-equiv="content-type" content="text/html; charset=UTF-8"
/>

 <title>Tutoriel de Savageman - Créez votre site (presque) complet
PHP : architecture MVC et bonnes pratiques !</title>

 <meta http-equiv="Content-Language" content="fr" />

 <link rel="stylesheet" href="style/global.css" type="text/css"
media="screen" />

</head>

<body>

 <h1>Créez votre site (presque) complet PHP : architecture MVC et
bonnes pratiques !</h1>

<?php include 'global/menu.php'; ?>

 <div id="centre">
```

Code : HTML - ~/global/menu.php

```php
<div id="menu">

  <h2>Menu</h2>

  <ul>
   <li><a href="index.php">Accueil</a></li>
  </ul>

  (rien pour le moment)

 </div>
```

Code : HTML - ~/global/bas.php

```
    </div>

     <div id="bas">
      Tutoriel réalisé par Savageman.
     </div>

    </body>

    </html>
```

Code : HTML - ~/global/accueil.php

```
    <h2>Page d'accueil</h2>

    <p>Bienvenue dans mon super tutoriel !<br />
    (page que l'on trouvera très intéressante)</p>
```

Enfin, voici le fichier CSS que j'utilise, ne vous moquez pas, mes talents de graphiste sont très limités !

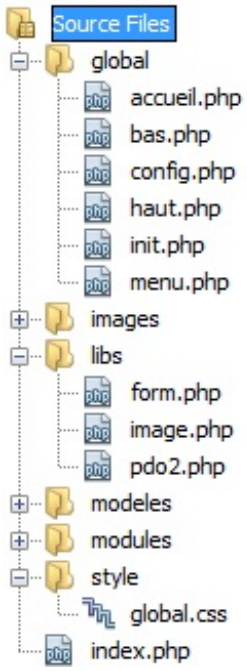
Code : CSS - ~/style/global.css

```
    *
    {
     font-family: Calibri;
    }
    body, #menu, #centre
    h1, h2, h3, h4, h5, h6
    {
     margin: 0;
     padding: 0;
     border: 0;
    }
    h1
    {
     background-color: #1166ff;
     padding: 0.5em;
    }
    h2, h3, h4, h5, h6
    {

     margin-bottom: 0.5em;
    }
    #menu
    {
     float: left;
     width: 18%;
     margin: 0em 1em 1em;
     padding: 1em;
     background-color: #22ff37;
    }
    #centre
    {
     margin: 1em 1em 1em 23%;
     padding: 1em;
     background-color: #eeeeee;
    }
    #bas
    {
     clear: both;
     padding: 1em;
     background-color: #3366ff;
    }
    label
    {
```

```
 float: left;
 width: 250px;
 text-align: left;
}
input
{
 margin-bottom: 2px;
}
table, tr, td, th
{
 border-collapse: collapse;
 border: 1px solid black;
}
```

Récapitulatif de l'arborescence



Bonnes pratiques à appliquer

Pas de bol, on ne commence pas encore à coder ! Cette partie a pour but de vous sensibiliser à certains problèmes connus et récurrents. Ces problèmes incluent notamment la sécurité de votre site et de vos applications avec la présentation de la faille XSS ainsi que les guillemets magiques.
Ensuite, nous verrons comment écrire un code lisible en pratiquant ce que j'appelle l'**auto-documentation** et en utilisant des constantes.

Voici le plan de cette partie :

1. **Never Trust User Input** : la faille XSS ;
2. **L'enfer des guillemets magiques** : pourquoi et comment les désactiver ?
3. **Produire un code lisible et compréhensible** :
   - un nommage correct de vos variables et fonctions dans un but d'auto-documentation,
   - une utilisation judicieuse des constantes.

Cette partie est très importante pour la suite. Ça n'est pas forcément le plus intéressant car elle aborde beaucoup de théorie, mais c'est une étape indispensable pour commencer à coder sereinement !

Never Trust User Input

Cette partie a entièrement été rédigée par `Haku, ancien membre impliqué du SdZ. Seules quelques modifications mineures y ont été apportées afin de la sortir du contexte de son tutoriel d'origine.

Les formulaires sont dangereux ! La plupart des failles de sécurité liées au PHP tirent leur origine de $_GET et de $_POST (ainsi que de $_COOKIE, mais c'est pour plus tard). Il y a des risques et on peut se faire avoir !

Mais dans tous les cas, soyez rassurés. La sécurité en PHP tient en quelques mots : Never Trust User Input.
Littéralement et en français, ça veut dire "Ne jamais croire / faire confiance aux entrées de l'utilisateur". Quand vous mettez un formulaire à disposition sur votre site, vous le faites généralement pour faire un sondage, ou pour laisser la possibilité aux gens de s'exprimer. Mais beaucoup de dérives sont possibles, à plusieurs niveaux. Ce petit texte n'a pas pour vocation de vous enseigner toutes les ficelles de la sécurité, car c'est avant tout une histoire de comportement. La faille que je vais vous montrer est la faille XSS.

Dans un dossier séparé, créez une page url.php et mettez-y ce code, puis validez le formulaire.

Code : PHP

```
<?php

if(isset($_POST['texte'])) {
        echo $_POST['texte'];
}

?>
<form action="./url.php" method="post">
        <p>
                <textarea name="texte" rows="10" cols="50">
                        <script type="text/javascript">
                        alert('Ohoh, vous avez été piraté par `Haku
!!');
                        </script>
                </textarea>
        <input type="submit" name="valid" value="go" />
        </p>
</form>
```

Testez ce code et... Oh mon dieu ! Je vous ai piratés. Non, rassurez-vous, je n'ai rien fait de méchant, votre PC ne va pas exploser, votre disque dur ne va pas danser la samba, les aliens ne vont pas débarquer, j'ai juste mis un petit code JavaScript qui permet d'afficher une boîte de dialogue. Ce n'est pas plus méchant qu'une plume.

Vous voyez donc que si on laisse l'utilisateur mettre ce qu'il veut, il peut faire des choses qui ne sont pas prévues à la base et qui

peuvent être ennuyeuses ou même dangereuses.

Par exemple, faire une boucle infinie qui afficherait tout le temps des boîtes de dialogue comme celle que j'ai affichée, ça serait ennuyeux. Vous devriez tuer le processus de votre navigateur Internet pour sortir de cette situation.
Mais en quoi est-ce dangereux ? C'est simple. En PHP, il existe une superglobale, $_COOKIE, qui permet de sauvegarder des informations sur l'ordinateur d'un visiteur. Ne criez pas au scandale, les cookies sont de simples fichiers texte, ils ne peuvent rien faire sur votre ordinateur, ce n'est pas Big Brother, ce sont juste des fichiers texte qui permettent au script de sauver des informations propres au visiteur (par exemple, si vous allez sur un site qui propose plusieurs designs, il stockera une information dans un cookie qui permettra de ne pas avoir à vous redemander quel design vous voulez à chaque visite).

Mais on peut mettre des informations beaucoup plus sensibles dans les cookies, comme par exemple des mots de passe pour vous connecter à des espaces membres (les mots de passe ne sont pas là tels quels, ils sont "cachés" grâce à une fonction de hashage).
Et c'est là que la faille XSS arrive. Vous avez vu que je peux injecter du code JavaScript et le faire s'exécuter. Le problème, c'est que le JavaScript **peut accéder aux cookies**. Et donc, si je remplaçais le code qui affiche une boîte de dialogue par un code plus vicieux, qui vous redirigerait vers une page piégée, je pourrais obtenir votre mot de passe et me faire passer pour vous sur tous les sites où vous employez ce mot de passe (c'est d'ailleurs pourquoi il est conseillé de ne pas utiliser tout le temps le même mot de passe, il faut en avoir plusieurs, idéalement, un par site / application).

Maintenant, comment s'en protéger ?
La méthode est simple et a fait ses preuves : utilisez la fonction PHP htmlspeciachars().
Cette fonction transforme 5 caractères en leur équivalent HTML. Bon, et alors ? Eh bien c'est simple.
Si vous tapez ceci dans une page HTML :

Code : HTML

```
<script type="text/javascript">alert('hoho');</script>
```

vous allez revoir une boîte de dialogue.
Maintenant on va remplacer les chevrons (< et >) par leur équivalent HTML respectif : < et >.
Remettez ce code dans une page HTML et regardez ce qu'il se passe :

Code : HTML

```
&lt;script type="text/javascript"&gt;alert('hoho');&lt;/script&gt;
```

Magie, plus de boîte de dialogue. La raison est simple : votre navigateur sait que s'il rencontre les balises <script></script>, il doit interpréter ce qu'il y a entre les balises comme du JavaScript. Mais comme on a remplacé les chevrons par leur équivalent HTML (c'est-à-dire une suite de caractères qui affiche la même chose mais qui n'a pas la même valeur), votre navigateur n'interprète pas ce qu'il y a entre les balises comme du JavaScript.

Et voilà comment se protéger de cette faille.
Reprenons le code du début en y adjoignant la fonction htmlspecialchars() et admirons le résultat :

Code : PHP

```
<?php

if(isset($_POST['texte'])) {

        echo htmlspecialchars($_POST['texte']);
}

?>
<form action="./url.php" method="post">
        <p>
                <textarea name="texte" rows="10" cols="50">
                        <script type="text/javascript">
                        alert('Ohoh, vous avez été piraté par `Haku
!!');
                        </script>
                </textarea>
        <input type="submit" name="valid" value="go" />
        </p>
</form>
```

Voici la liste des caractères remplacés par htmlspecialchars() :

- & => **&**
- < => **<**
- > => **>**
- " => **"**
- ' => **'**

Certains caractères ne sont remplacés que si vous donnez un second argument à la fonction ; pour savoir comment ça fonctionne, allez voir dans la documentation !

L'enfer des guillemets magiques

Cette partie a entièrement été rédigée par `Haku, ancien membre impliqué du SdZ.

Après un petit blabla fort intéressant sur la faille XSS, on va prendre un peu de temps pour discuter d'un autre problème plutôt ennuyeux (je voulais dire emmerdant, mais ce n'est pas très pédagogique !) : les guillemets magiques.

Ça me pose un problème d'en parler ici, puisque la cause de ces guillemets magiques, les magic_quotes, ce qui a justifié leur création vous est encore inconnu. Je vais donc en parler, mais très brièvement.

Testez ce code :

Code : PHP

```php
<?php

$text = 'lala des " et encore des " ' . " et des ' sans oublier les
'";
echo $text;
echo addslashes($text);

?>
```

Vous voyez que les deux echo (echo qui est une structure, pas une fonction, je le rappelle) n'affichent pas la même chose. On aurait pu s'en douter vu que l'un affiche directement $text alors que l'autre affiche la valeur de retour d'une fonction : addslashes().
Le but de cette fonction est très simple : elle rajoute un backslash (\) devant ces 4 chaînes de caractères : ", ', \ et NULL.

Bon, vous allez me demander quel est l'intérêt de cette fonction ; je pourrais vous répondre, mais ça servirait à rien parce qu'on n'a pas encore abordé le sujet. Tout ce que je peux dire, c'est que pendant longtemps, cette fonction a été la défense de bien des codeurs face à une possible faille.

Mais quel rapport avec les formulaires ? Eh bien c'est simple. Dans le fichier de configuration de PHP (un truc obscur que l'on a jamais vu), il y a des directives, c'est-à-dire des ordres que PHP doit suivre.
L'une de ces directives s'appelle **magic_quote_gpc**. Si cette directive est activée, PHP va appliquer la fonction addslashes() à toutes les valeurs présentent dans les arrays GPC ($_GET, $_POST et $_COOKIE). Et donc, on risque de se retrouver avec une panoplie de backslashes en trop !

Idéalement, il faudrait que cette directive disparaisse : elle n'a été implémentée que pour pallier au manque de mesures de sécurité des codeurs PHP. Seulement, comme cette directive est désactivable, la configuration peut changer. Si vous faites votre script sur un serveur qui a désactivé la directive et qu'ensuite vous décidez de changer de serveur et d'aller sur un autre qui a activé cette directive, vous aurez des backslashes dans tous les coins et vous devrez modifier votre script pour réparer les dégâts. Heureusement, la version 6 de PHP fera disparaître cette directive. Malheureusement, il y a des tas de serveurs qui resteront avec PHP version 5 pendant un bon moment, et même avec PHP version 4.
Je vais vous donner un code qui va vous permettre de ne plus avoir à vous soucier des guillemets magiques, et je vais vous l'expliquer.

Code : PHP

```php
<?php

if(get_magic_quotes_gpc()) {
```

```
            $_POST = array_map('stripslashes', $_POST);
            $_GET = array_map('stripslashes', $_GET);
            $_COOKIE = array_map('stripslashes', $_COOKIE);
    }

    ?>
```

Le fonctionnement est très simple : d'abord, on vérifie si les magic_quotes sont activés avec la fonction get_magic_quotes_gpc(). Si c'est le cas, on va utiliser la fonction inverse d'addslashes() : stripslashes().
Pour appliquer une fonction à l'ensemble des valeurs d'un array, on utilise la fonction array_map(). Le premier argument est le nom de la fonction qu'on veut appliquer (stripslashes() dans notre cas) et le second argument, l'array sur lequel on va travailler.

Cette fonction est rudimentaire et peut être améliorée, mais pour le moment, ça suffira amplement.
Mais comme je l'ai dit, l'idéal est de les désactiver. Donc si vous avez accès au fichier de configuration de PHP (php.ini pour les intimes), suivez cette procédure pour désactiver les magic_quotes :

- ouvrir le php.ini (chez moi, il est dans C:\Wamp\php\php.ini) ;
- rechercher la chaîne 'magic_quotes_gpc' ;
- mettre Off comme valeur pour les trois directives des magic_quotes pour obtenir ceci :

Code : Autre

```
; Magic quotes
;

; Magic quotes for incoming GET/POST/Cookie data.
magic_quotes_gpc = Off

; Magic quotes for runtime-
generated data, e.g. data from SQL, from exec(), etc.
magic_quotes_runtime = Off

; Use Sybase-style magic quotes (escape ' with '' instead of \').
magic_quotes_sybase = Off
```

Reste à redémarrer votre serveur pour que la modification soit effective.

Boum, plus de magic_quotes !
On reviendra là-dessus dans la partie II de ce tutoriel de toute façon, pour découvrir pourquoi cette directive existe .

Produire un code lisible et compréhensible

Un nommage correct de vos variables et fonctions dans un but d'auto-documentation

Ce paragraphe a été rédigé en majeure partie par `Haku, ancien membre impliqué du SdZ. Quelques modifications y ont été apportées afin de l'adapter à ce tutoriel.

Un point très important est le nom de vos variables. En effet, le nom d'une variable est un élément très important. S'il est trop long, ça serait très embêtant à taper (par exemple $NombreDeMessageParPage est un nom de variable certes très clair, mais d'une longueur affolante). Il faut donc vous limiter et ne pas exagérer.
Mais il ne faut pas non plus tomber dans l'excès inverse. Si vous mettez des noms de variables trop court, vous risquez de ne plus savoir ce qu'est telle ou telle variable (par exemple $inf est un nom très court, mais ça ne me dit rien du tout).
Il vous faut donc trouver un juste milieu entre clarté et longueur. L'idéal, c'est d'avoir un style de variable. Prenez les mots que vous utilisez le plus souvent dans vos noms de variables, et trouvez des diminutifs. Par exemple, au lieu de Nombre, vous pouvez mettre Nb, au lieu de Information, mettez Info. Vous pouvez également supprimer tous les mots blancs comme "de", "par", etc.

Si je reprends l'exemple du $NombreDeMessageParPage, ça donnerait $nb_mess_page. C'est bien plus court, mais c'est tout aussi lisible pour autant qu'on conserve toujours la même convention de nommage.

Un nom de variable bien choisi n'est donc pas trop long à taper tout en restant clair et facilement compréhensible, ce qui contribue à ce que j'appelle l'**auto-documentation** des scripts. Un nom de variable clair permet de ne pas écrire un commentaire qui aurait été utile avec un nom de variable inadapté. Prenez donc 10 secondes à utiliser de jolies noms de variables plutôt que d'écrire des commentaires inutiles. Votre code sera tout autant compréhensible, tout en étant plus court et plus rapide à lire.

Les mêmes règles de nommage peuvent s'appliquer pour les fonctions, les constantes et même les noms de fichiers.
Par exemple, dans ce tutoriel, je remplacerai "initialisation" par "init", "configuration" par "config" ou encore "librairie" par "lib".

Une utilisation judicieuse des constantes

Ce paragraphe a été rédigé en partie par `Haku, ancien membre impliqué du SdZ. Quelques modifications y ont été apportées afin de l'adapter à ce tutoriel.

Définition

Je vais maintenant vous parler très brièvement des constantes. Vous en avez peut-être déjà entendu parler, mais voici une explication plus poussée.

Une **constante** est une sorte de variable, dont on définit le nom et la valeur une fois, et dont la valeur sera impossible à modifier tout au long du script.

Voici comment on déclare une constante :

Code : PHP

```php
<?php

define('NOM_CONSTANTE', 'Valeur de la constante.');
echo NOM_CONSTANTE;

?>
```

Pour déclarer une constante, on utilise la fonction define(). Le premier argument est le nom qu'aura la constante et le second est la valeur qu'elle prendra (une expression quelconque).

On peut ensuite utiliser cette constante dans beaucoup de situations : affichage (comme dans l'exemple), concaténation, assignation, calcul, etc. Une constante s'utilise de la même manière qu'une variable, vous ne devriez donc pas avoir de souci à ce niveau.

Notez que la valeur d'une constante ne peut pas changer : si on essaye de définir une constante qui existe déjà, une belle erreur pointe le bout de son nez. Pour résumer, une constante ne se définit qu'une seule fois.

Mais quel intérêt ? Quand les utiliser ?

Je dirais qu'il y a trois cas majeurs où les constantes sont intéressantes.

Le premier cas, c'est si vous voulez vous assurer qu'une valeur ne change jamais. Une valeur qui resterait constante du début à la fin du script. Par exemple, pour configurer un script (nombre de news affichées par page, si on veut afficher les signatures sur un forum ou non, etc.), c'est très utile, parce qu'on ne risque pas de changer cette valeur.

Le second cas, c'est ce que j'appelle les arguments muets, c'est-à-dire sans signification claire. Quand vous créez une fonction, vous lui donnez souvent des arguments, mais il peut arriver que vous ayez besoin de donner des arguments que je qualifie de muets.
Voici la liste des arguments muets :

- un entier ;
- un nombre à virgule ;

- un booléen.

Pourquoi sont-ils muets ?
Tout simplement parce qu'on ne sait pas quel est leur rôle, à quoi ils servent. Un exemple vaut mieux qu'un discours et c'est cadeau :

Code : PHP

```php
<?php
echo fonction_affiche_news(NB_NEWS_PAGE);
?>
```

Code : PHP

```php
<?php
echo fonction_affiche_news(20);
?>
```

La constante NB_NEWS_PAGE est définie comme telle : define('NB_NEWS_PAGE', 20);.

Imaginez que ce ne soit pas votre script : vous tombez sur l'appel de cette fonction. Qui peut me dire à quoi sert le nombre 20 ? À afficher une news particulière ? Ou une catégorie particulière de news ? À limiter le nombre de news par page ?

On ne peut pas le deviner sans aller voir la déclaration de la fonction, et on perd du temps.

Tandis qu'avec une constante, on a un mot, un nom qui nous permet d'identifier l'utilité de ce paramètre. Quand les scripts sont petits et développés seuls, l'intérêt est limité, je l'avoue. Mais quand on a de gros scripts qu'on crée à plusieurs, ça devient vite le bordel et les constantes permettent d'améliorer un peu la lisibilité du code !

Enfin, **le troisième cas** dans lequel les constantes sont utiles est lorsque que l'on a des valeurs constantes présentes plusieurs fois directement dans son code.

Citons un cas concret.

Imaginez que vous placiez toutes vos fonctions dans des fichiers séparés dans un dossier précis, par exemple /include/fonctions/. Chaque fois que vous voudrez inclure une fonction pour l'utiliser, vous devez écrire le chemin vers ce dossier.
Vous utiliserez donc plusieurs fois ce chemin dans vos scripts et cette valeur ne changera pas d'une page à l'autre. C'est là qu'une constante est utile : plutôt que d'écrire à chaque fois ce chemin, vous le définissez une bonne fois pour toute dans une constante. En plus d'éviter de réécrire le chemin complet à chaque fois, vous pourrez facilement changer sa valeur, et donc rendre votre code plus flexible.

Généralement, c'est une bonne chose de remplacer toutes ses valeurs constantes présentes directement dans son code par une constante.

De plus, notez que si vous regroupez vos constantes en haut d'un fichier particulier, vous y aurez facilement accès et pourrez les modifier à votre guise. Ainsi organisé, votre fichier vous permettra de **configurer** votre site simplement, juste en modifiant la valeur de ces constantes.

Certaines constantes, une fois que vous avez commencé à travailler avec, ne peuvent plus être modifiées sans faire bugger votre script (exemple : le dossier pour stocker les avatars). Il convient donc de rester vigilant lorsqu'on le fait. Par exemple, si vous changez le dossier où sont stockés les avatars, il vous faudra bien penser à transférer les avatars

déjà existants vers le nouveau dossier.

Vous êtes désormais armés contre la faille XSS et les guillemets magiques. Vous devriez également être capables d'écrire du code lisible et compréhensible par tous, ce qui est une bonne chose.
Enfin, n'hésitez pas à utiliser des constantes pour bannir toute forme de paramètres sans signification claire (entiers, flottants et booléens).

Les classes utilisées dans ce tutoriel

Les classes distribuées dans ce tutoriel (Pdo2, Form et Image) sont **toutes** libres de droits (contrairement au tutoriel en lui-même). Vous pouvez les modifier et les ré-utiliser (même dans un cadre commercial) comme bon vous semble.

Je précise seulement qu'elle sont fournies dans l'état et qu'il n'y a aucune garantie ni service après-vente dessus.

Accès à la base de données avec PDO2

Nous allons donc utiliser PDO, qui - je le rappelle - définit une excellente interface pour accéder à une BDD depuis PHP 5.1.

Le seul problème étant que dans ce que nous avons développé, nous avons besoin d'une seule et unique connexion à la base de données. Je lui ai donc ajouté cette fonctionnalité.

Comment ça s'utilise ?
Pour faire simple, lorsque l'on veut faire une requête, nous demandons à récupérer un objet PDO (qui est déjà connecté à la base de données) grâce à la méthode PDO2::getInstance();

À titre d'information, voici comment ça fonctionne en interne : lorsque l'on demande à récupérer cet objet PDO, 2 solutions s'offrent à la classe :

- soit l'objet existe déjà et nous le fournissons à l'utilisateur ;
- soit l'objet n'existe pas et la classe le crée tout en le connectant à la base de données avant de le fournir également à l'utilisateur.

Au final, pour l'utilisateur de la classe (nous, vous), c'est facile : nous avons juste à demander l'objet PDO prêt à utiliser et la classe nous le retourne dans tous les cas.

L'avantage est que l'on ne demande jamais soi-même explicitement de se connecter à la base de données : la classe le fait pour nous. Et si l'on ne demande jamais l'objet, la connexion à la base de données ne se fera jamais non plus.

Sans plus attendre, voici le code source à copier dans un fichier ~/libs/pdo2.php :

Code : PHP - ~/libs/pdo2.php

```php
<?php

/**
* Classe implémentant le singleton pour PDO
* @author Savageman
*/

class PDO2 extends PDO {

 private static $_instance;

 /* Constructeur : héritage public obligatoire par héritage de PDO
*/
 public function __construct( ) {

 }
 // End of PDO2::__construct() */

 /* Singleton */
 public static function getInstance() {

  if (!isset(self::$_instance)) {

   try {

    self::$_instance = new PDO(SQL_DSN, SQL_USERNAME, SQL_PASSWORD);

   } catch (PDOException $e) {

    echo $e;
   }
```

```
        }
        return self::$_instance;
      }
      // End of PDO2::getInstance() */
    }

    // end of file */
```

On notera que la classe a besoin de 3 constantes pour fonctionner correctement : SQL_DSN, SQL_USERNAME, et SQL_PASSWORD

Une fois l'objet PDO récupéré avec la méthode PDO2::getInstance();, tout se passe comme si vous manipuliez un objet PDO. Mon but n'est pas de faire un cours là-dessus, étant donné que c'est un des pré-requis du tutoriel : savoir utiliser un objet. Pour les méthodes disponibles dans PDO, on se réfèrera au tutoriel de Draeli, et plus particulièrement à la partie 5 qui parle des requêtes préparées. On pourra également jeter un oeil du côté de la documentation officielle.

Gestion des formulaires avec la classe Form

Ce tutoriel étant le mien, j'en profite pour promouvoir un outil que j'ai créé et que j'utilise. C'est un ensemble de classes servant à manipuler des formulaires. Il automatise plusieurs processus en liant la création et la vérification du formulaire. Le principal avantage est que le code de vérification (validation) est beaucoup plus simple, car la classe fait tout pour vous ! De plus, vous pourrez placer le code de création du formulaire et celui de vérification sur la même page, ce qui simplifiera légèrement votre code.

Quoi, déjà une classe ? Je ne comprends pas, alors un ensemble de classes ? 

Pas de panique ! Vous avez juste besoin de comprendre comment cela fonctionne : je vais vous donner un exemple, vous verrez que ça n'est pas si compliqué que ça !

Commencez par récupérer le code source à cette adresse : source de la classe Form et placez-le dans le fichier ~/libs/form.php.

Création et affichage d'un formulaire

Premier exemple, cas le plus simple :

Code : PHP

```
<?php

// Ne pas oublier d'inclure la libraire Form
include 'form.php';

// Création d'un objet Form. // L'identifiant est obligatoire !
$mon_form = new Form('identifiant_unique');

// Affichage du formulaire
echo $mon_form;
```

Résultat :

Code : HTML

```
<form method="GET">
 <p>
  <input name="uniqid" type="hidden" value="identifiant_unique" />
 </p>
</form>
```

Deuxième exemple, modification des attributs HTML :

Code : PHP

```php
<?php

// Ne pas oublier d'inclure la libraire Form
include 'form.php';

// Création d'un objet Form. // L'identifiant est obligatoire !
$mon_form = new Form('identifiant_unique');

// Modification des attributs
$mon_form->method('POST');
$mon_form->action('index.php');

// Affichage du formulaire
echo $mon_form;
```

Résultat :

Code : HTML

```html
<form method="POST" action="index.php">
 <p>
  <input name="uniqid" type="hidden" value="identifiant_unique" />
 </p>
</form>
```

Il est également possible de spécifier l'attribut "method" directement dans le constructeur :

Code : PHP

```php
<?php

// Ne pas oublier d'inclure la libraire Form
include 'form.php';

// Création d'un objet Form. // L'identifiant est obligatoire !
$mon_form = new Form('identifiant_unique', 'POST');

// Modification de l'attribut "action"
$mon_form->action('test.php');

// Affichage du formulaire
echo $mon_form;
```

Résultat :

Code : HTML

```html
<form method="POST" action="test.php">
 <p>
  <input name="uniqid" type="hidden" value="identifiant_unique" />
 </p>
</form>
```

Ajout d'un champ

Premier exemple :

Code : PHP

```php
<?php

// Ne pas oublier d'inclure la libraire Form
include 'form.php';

// Création d'un objet Form. // L'identifiant est obligatoire !
$mon_form = new Form('identifiant_unique', 'POST'); //
'identifiant_unique'

// Ajout d'un champ de type "Text" nommé "prenom"
$mon_form->add('Text', 'prenom');

// Affichage du formulaire
echo $mon_form;
```

Résultat :

Code : HTML

```html
<form method="POST">
 <p>
  <input name="prenom" type="text" />
 </p>
 <p>
  <input name="uniqid" type="hidden" value="identifiant_unique" />
 </p>
</form>
```

Deuxième exemple, avec un label :

Code : PHP

```php
<?php

// Ne pas oublier d'inclure la libraire Form
include 'form.php';

// Création d'un objet Form. // L'identifiant est obligatoire !
$mon_form = new Form('POST'); // 'identifiant_unique'

// Ajout d'un champ texte nommé "prenom"
$mon_form->add('Text', 'prenom')
         ->label('Votre prénom SVP');

// Affichage du formulaire
echo $mon_form;
```

Résultat :

Code : HTML

```html
<form method="GET">
```

```
        <p>
         <label id="id_prenom">Votre prénom SVP :</label>
         <input for="id_prenom" name="prenom" type="text" />
        </p>
        <p>
         <input name="uniqid" type="hidden" value="POST" />
        </p>
       </form>
```

Les différents types de champs et leur utilisation

La classe permet de créer :

- des zones de texte ;
- des zones de texte multi-lignes ;
- des champs mot de passe ;
- des champs cachés ;
- des boutons de validation ;
- des listes d'options ;
- des listes d'options hiérarchisées ;
- des cases à cocher ;
- des champs pour uploader un fichier ;
- des champs "adresse email" ;
- des champs dates / heures.

La zone de texte

Elle correspond à la balise HTML `<input type="text" />` . Elle est représenté par le type **Text** et se crée donc ainsi :

Code : PHP - Ajout d'une zone de texte à $mon_form

```php
<?php
// Ajout d'un champ texte nommé "prenom"
$mon_form->add('Text', 'prenom');
```

La zone de texte multi-lignes

Elle correspond à la balise HTML `<textarea></textarea>` . Elle est représenté par le type **Textarea** et se crée donc ainsi :

Code : PHP - Ajout d'une zone de texte multi-lignes à $mon_form

```php
<?php
// Ajout d'un champ texte nommé "commentaires"
$mon_form->add('Textarea', 'commentaires');
```

Le champ mot de passe

Il correspond à la balise HTML `<input type="password" />` . Il est représenté par le type **Password** et se crée donc ainsi :

Code : PHP - Ajout d'un champ mot de passe à $mon_form

```php
<?php
```

```php
// Ajout d'un champ mot de passe nommé "mot_de_passe"
$mon_form->add('Password', 'mot_de_passe');
```

Le champ caché

Il correspond à la balise HTML `<input type="hidden" />` . Il est représenté par le type **Hidden** et se crée donc ainsi :

Code : PHP - Ajout d'un champ caché à $mon_form

```php
<?php
// Ajout d'un champ cache nommé "secret"
$mon_form->add('Hidden', 'secret');
```

Le bouton de validation

Il correspond à la balise HTML `<input type="submit" />` . Il est représenté par le type **Submit** et se crée donc ainsi :

Code : PHP - Ajout d'un bouton de validation à $mon_form

```php
<?php
// Ajout d'un bouton de validation nommé "submit"
$mon_form->add('Submit', 'submit');
```

La liste d'options

Elle est réalisée grâce à la balise HTML `<input type="radio" />` . Elle est représentée par le type **Radio** et se crée donc ainsi :

Code : PHP - Ajout d'une liste d'options à $mon_form

```php
<?php
// Ajout d'une liste d'options pour choisir une couleur
$mon_form->add('Radio', 'couleur')
         ->choices(array(
           'v' => 'Vert',
           'o' => 'Orange',
           'r' => 'Rouge',
           'b' => 'Bleu'
         ));
```

Une liste d'options a besoin d'un tableau de valeurs que l'on passe à la méthode choices(). La clé correspond à la valeur récupérée lorsque l'on fait $mon_form->get_cleaned_data('couleur'); et la valeur correspond à ce qui est affiché à l'utilisateur dans la page HTML.

La liste d'options hiérarchisée

Elle est réalisée grâce aux balises HTML `<select>` `<optgroup>` et `<option>` . Elle est représentée par le type **Select** et se crée donc ainsi :

Code : PHP - Ajout d'une liste d'options hiérarchisée à $mon_form

```php
<?php
// Ajout d'une liste d'options pour choisir une pays
$mon_form->add('Select', 'pays')
```

```
            ->choices(array(
              'Europe' => array(
                'fr' => 'France',
                'de' => 'Allemagne'
              ),
              'Asie' => array(
                'cn' => 'Chine',
                'jp' => 'Japon'
              )
            ));
```

Elle peut s'utiliser de la même façon que la liste d'options, mais comme le montre l'exemple, on peut également effectuer un regroupement dans les choix possibles, afin d'améliorer la lisibilité de la liste.

La case à cocher

Elle correspond à la balise HTML `<input type="checkbox" />` . Elle est représentée par le type **Checkbox** et se crée donc ainsi :

Code : PHP - Ajout d'une case à cocher à $mon_form

```
<?php
// Ajout d'une case a cocher nommeé "license"
$mon_form->add('Checkbox', 'license');
```

Le champ pour uploader un fichier

Il correspond à la balise HTML `<input type="file" />` . Il est représenté par le type **File** et se crée donc ainsi :

Code : PHP - Ajout d'un champ pour uploader un fichier à $mon_form

```
<?php
// Ajout d'un champ pour uploader un fichier nommé "fichier"
$mon_form->add('File', 'fichier')
         ->max_size(4096)
         ->extensions(array('jpg', 'gif', 'png'));
```

La classe fournit également une méthode max_size() pour limiter la taille du fichier uploadé (la taille est spécifiée en octets) ainsi qu'une autre méthode extensions() pour filtrer les extensions autorisées. L'exemple proposé n'autorise que les fichiers images ayant une taille inférieure à 4 ko.

Le champ "adresse email"

Il étend la balise HTML `<input type="text" />` . Il est représenté par le type **Email** et se crée donc ainsi :

Code : PHP - Ajout d'un champ adresse e-mail à $mon_form

```
<?php
// Ajout d'un champ adresse e-mail nommé "email"
$mon_form->add('Email', 'email');
```

Il possède une règle de validation qui n'autorise l'utilisateur qu'à rentrer une adresse e-mail valide syntaxiquement. Dans la cas contraire, une erreur sera retournée afin que l'utilisateur entre une adresse valide.

Le champ dates / heures

Il étend la balise HTML `<input type="text" />` . Il est représenté par le type **Date** et se crée donc ainsi :

Code : PHP - Ajout d'un champ date à $mon_form

```
<?php
// Ajout d'un champ date nommé "date"
$mon_form->add('Date', 'date')
         ->format('dd/mm/yyyy');
```

Un champ date a besoin qu'on lui fournisse un format de date et heure grâce à la méthode format().
Il possède une règle de validation qui n'autorise l'utilisateur qu'à entrer une date dont le format correspond à celui précisé grâce à la méthode format(). Dans la cas contraire, une erreur sera retournée afin que l'utilisateur entre une date ayant le bon format.

Liste des formats de date et heure

Les formats supportés sont les suivants :

- dd : jour sur deux chiffres (entre 01 et 31) ;
- mm : mois sur deux chiffres (entre 01 et 12) ;
- yy : année sur deux chiffres (entre 1970 et 2069) ;
- yyyy : année sur quatre chiffres (entre 1900 et 9999) ;
- HH : heure sur deux chiffres (entre 01 et 23) ;
- MM : minute sur deux chiffres (entre 00 et 59) ;
- SS : secondes sur deux chiffres (entre 00 et 59).

Liste des options communes à tous les champs

Tous les champs possèdent les méthodes suivantes.

- autocomplete($bool) : permet de définir la propriété HTML "autocomplete". Ceci permet par exemple de ne pas pouvoir pré-remplir un champ particulier. Ne vous en servez pas si vous ne savez pas ce que c'est.
- maxlength($int) : permet de définir la longueur maximale que doit avoir champ texte. Si la longueur du champ est plus grande, une erreur sera retournée à l'utilisateur afin qu'il y écrive une valeur moins longue.
- minlenght($int) : permet de définir la longueur minimale que doit avoir un champ texte. Si la longueur du champ est plus petite, une erreur sera retournée à l'utilisateur afin qu'il y écrive une valeur plus longue.

Résumé des types de champs disponibles et de leurs options

Type du champ	Résultat HTML	Remarques
Text	`<input type="text" />`	-
Password	`<input type="password" />`	La valeur initiale est toujours vide.
Hidden	`<input type="hidden" />`	-
Submit	`<input type="submit" />`	-
Textarea	`<textarea></textarea>`	Possibilité de spécifier le nombre de lignes et de colonnes affichées grâce aux méthodes rows($nombre) et cols($nombre).
Email	`<input type="text" />`	La valeur doit être une adresse e-mail valide.
		La date doit respecter le format donné.

Date	`<input type="text" />`	Un format doit être donné à l'aide de la méthode ->format($format).
File	`<input type="file" />`	Possibilité de filtrer les extensions autorisées avec la méthode extensions($liste). Possibilité de limiter la taille maximale du fichier (en octets) grâce à la méthode max_size($taille).
Radio	`<input type="radio" />`	La liste des choix disponibles se fait grâce à la méthode choices($tableau).
Select	`<select>` `<option>` et `<optgroup>`	La liste des options et des groupes d'options disponibles se fait grâce à la méthode choices($tableau).
Checkbox	`<input type="checkbox" />`	Possibilité de régler l'état coché ou non de la case avec la méthode checked($booleen).

Validation / vérification du formulaire et récupération des valeurs des champs

Prenons comme exemple ce formulaire :

Code : PHP - Formulaire d'exemple

```php
<?php

// Ne pas oublier d'inclure la libraire Form
include 'form.php';

// Création d'un objet Form. // L'identifiant est obligatoire !
$mon_form = new Form('exemple', 'POST');

// Ajout d'un champ texte nommé "pseudo"
$mon_form->add('Text', 'pseudo')
                ->label('Votre pseudo SVP');

// Ajout d'une liste d'options pour choisir un pays
$mon_form->add('Select', 'pays')
         ->label('Quel est votre pays préféré')
         ->choices(array(
           'Europe' => array(
             'fr' => 'France',
             'de' => 'Allemagne'
           ),
           'Asie' => array(
             'cn' => 'Chine',
             'jp' => 'Japon'
           )
         ));

// Affichage du formulaire
echo $mon_form;
```

Nous allons désormais afficher le résultat de ce formulaire après en avoir vérifié sa validité. C'est très simple :

Code : PHP - Formulaire d'exemple

```php
<?php

// Début identique
// [...]

// Si le formulaire est valide avec les données issues du tableau
```

```
        $_POST
        if ($mon_form->is_valid($_POST)) {

         // On récupère les valeurs
         list($pseudo, $pays) = $mon_form->get_cleaned_data('pseudo',
        'pays');

         // Et on les affiche
         echo 'Vous êtes '.$pseudo.' et votre pays préféré est
        "'.$pays.'".';

        } else {

         // Sinon on affichage le formulaire jusqu'à ce que ça soit valide
         echo $mon_form;
        }
```

Ça n'est pas plus compliqué ! 

Note : get_cleaned_data() *prend soit la liste des clés dont on veut récupérer la valeur, soit un tableau listant les clés que l'on veut récupérer. Elle peut également prendre une chaîne unique si l'on souhaite ne récupérer les données que d'un seul champ.*

La suite ?

Vous découvrirez d'autres fonctions utiles lors de la lecture de ce tutoriel, mais vous avez un aperçu global du fonctionnement de la classe, de son utilité et des manières de l'utiliser, ça suffira pour le moment.

Partie 2 : Des systèmes simples : espace membres et livre d'or

Voici le début des choses intéressantes : on va vraiment commencer à coder et à avoir un résultat visible ! Si vous avez bien suivi tous les conseils donnés lors de l'introduction, ça ne devrait pas vous poser trop de problèmes.

Ne vous découragez surtout pas !

Étant donné que nous allons manipuler beaucoup de fichiers différents, je conçois très bien que c'est un peu difficile au début. Mais rassurez-vous, je suis là pour vous guider et vous expliquer pas à pas afin que vous ne vous perdiez pas. Ça devrait bien se passer.

La base : l'espace membre

Cette partie aborde la mise en place de l'espace membre, qui est une brique indispensable à la mise en place d'une tonne d'autres modules (comme par exemple des news).

Cette partie est volontairement plus "assistée" que les suivantes. C'est pour vous permettre de vous habituer aux outils que vous n'avez sans doute pas l'habitude d'utiliser, comme par exemple la classe PDO2 ou bien la classe Form. De plus l'architecture est sans doute nouvelle pour vous, cette première partie vous permettra de bien comprendre comment tout fonctionne afin que je puisse vous lâcher dans la nature plus librement dans les parties suivantes.

Objectifs et préparation

Objectifs

L'objectif est de créer un espace membre. Parmi les fonctionnalités, on trouvera :

- inscription d'un membre ;
- connexion et déconnexion ;
- affichage du profil utilisateur ;
- modification du profil utilisateur.

Il vous sera nécessaire de connaître la partie sur les sessions en PHP, étant donné que nous allons nous en servir lors de la réalisation.

La table SQL

Code : SQL

```sql
CREATE TABLE membres (

  id               int(10) unsigned NOT NULL AUTO_INCREMENT,
  nom_utilisateur  varchar(32)  NOT NULL,
  mot_de_passe     char(40)     NOT NULL,
  adresse_email    varchar(128) NOT NULL,
  hash_validation  char(32)     NOT NULL,
  date_inscription date NOT     NULL,
  avatar           varchar(128) NOT NULL DEFAULT '',

  PRIMARY KEY (id),
  UNIQUE KEY nom_utilisateur (nom_utilisateur),
  UNIQUE KEY adresse_email (adresse_email),
  KEY mot_de_passe (mot_de_passe)

) ENGINE=InnoDB DEFAULT CHARSET=utf8 AUTO_INCREMENT=1 ;
```

Les champs sont explicites, je ne pense pas qu'il soit nécessaire de revenir dessus.
La taille du champ mot_de_passe est de 40 caractères pour pouvoir stocker le SHA1 du mot de passe.

Le champ hash_validation contient le hash de validation (un MD5) tant que le membre n'a pas validé son compte via l'e-mail qui lui a été envoyé. En gros, si ce champ est vide, le compte est validé ; sinon il n'est pas validé.

Créez donc la table SQL dans votre base de données (si ce n'est déjà fait).

Bonne pratique : utiliser des index et des contraintes unique là où c'est nécessaire dans votre base de données. Cela assurera un fonctionnement rapide de votre base de données ainsi qu'une intégrité des données qu'elles contient.

Inscription

Préparation du terrain

Nous travaillons sur l'inscription d'un membre. Elle correspond logiquement à l'action inscription du module membres. Créez donc le fichier ~/modules/membres/inscription.php. Afin d'accéder facilement à cette page, nous allons ajouter l'inscription dans le menu :

Code : HTML - ~/global/menu.php

```
<div id="menu">

  <h2>Menu</h2>

  <ul>
   <li><a href="index.php">Accueil</a></li>
  </ul>

<h3>Espace membre</h3>
<ul>
<li><a href="index.php?
module=membres&amp;action=inscription">Inscription</a></li>
</ul>

 </div>
```

Nous aurons besoin de vues, créez donc également le dossier ~/modules/membres/vues/.

Le formulaire

L'id, le hash de validation et la date d'inscription sont générés automatiquement, nous n'avons pas besoin de demander ces informations à l'utilisateur lors de son inscription. Dans notre formulaire, nous avons donc besoin :

- d'un nom d'utilisateur ;
- d'un mot de passe ;
- d'une adresse e-mail ;
- d'un avatar facultatif.

Ce qui nous mène à la création du formulaire suivant (en utilisant la classe Form) :

Code : PHP - ~/modules/membres/inscription.php

```
<?php

// Ne pas oublier d'inclure la librarie Form
include CHEMIN_LIB.'form.php';

// "formulaire_inscription" est l'ID unique du formulaire
$form_inscription = new Form('formulaire_inscription');

$form_inscription->method('POST');

$form_inscription->add('Text', 'nom utilisateur')
```

```
                    ->label("Votre nom d'utilisateur");

$form_inscription->add('Password', 'mdp')
                    ->label("Votre mot de passe");

$form_inscription->add('Password', 'mdp_verif')
                    ->label("Votre mot de passe (vérification)");

$form_inscription->add('Email', 'adresse_email')
                    ->label("Votre adresse email");

$form_inscription->add('File', 'avatar')
                    ->filter_extensions('jpg', 'png', 'gif')
                    ->max_size(8192) // 8 Kb
                    ->label("Votre avatar (facultatif)")
                    ->Required(false);

$form_inscription->add('Submit', 'submit')
                    ->value("Je veux m'inscrire !");

// Pré-remplissage avec les valeurs précédemment entrées (s'il y en
a)
$form_inscription->bound($_POST);
```

Bonne pratique : remarquez que chaque ligne de code est alignée avec la précédente, ce qui rend le code très lisible. L'indentation utilisée permet donc de lire rapidement et facilement le code du formulaire.

Voilà, le formulaire est créé : ça n'est pas très compliqué et tout de même bien pratique !
Par exemple, pour l'avatar, on peut demander directement à la classe de filtrer certaines extensions et de limiter la taille du fichier.


Vous ne voyez pas le joli formulaire que l'on vient de créer ? C'est normal, rien ne s'affiche pour le moment, et devinez pourquoi ? Allez, un petit effort...! Oui, c'est ça ! Nous n'avons encore aucune vue ! Réglons ça tout de suite.

Ajoutons ceci à la fin du fichier du fichier ~/modules/membres/inscription.php fraîchement créé :

Code : PHP

```
<?php

// Affichage du formulaire
include CHEMIN_VUE.'formulaire_inscription.php';
```

Voici la vue correspondante (si vous avez suivi, l'emplacement du chemin pour ce fichier ne devrait pas vous poser de problème) :

Code : PHP - ~/modules/membres/vues/formulaire_inscription.php

```
<h2>Inscription au site</h2>

<?php

echo $form_inscription;
```

La vérification

Le code du contrôleur

Il est temps de vérifier si les données du formulaire sont valides (mots de passe identiques, adresse e-mail valide) et si un membre n'a pas déjà le nom d'utilisateur et / ou l'adresse e-mail demandés.

Code : PHP - ~/modules/membres/inscription.php (suite)

```php
<?php

// Création d'un tableau des erreurs
$erreurs_inscription = array();

// Validation des champs suivant les règles en utilisant les données
du tableau $_POST
if ($form_inscription->is_valid($_POST)) {

 // On vérifie si les 2 mots de passe correspondent
 if ($form_inscription->get_cleaned_data('mdp') !=
$form_inscription->get_cleaned_data('mdp_verif')) {

  $erreurs_inscription[] = "Les deux mots de passes entrés sont
différents !";
 }

 // Si d'autres erreurs ne sont pas survenues
 if (empty($erreurs_inscription)) {

  // Traitement du formulaire à faire ici

 } else {

  // On affiche à nouveau le formulaire d'inscription
  include CHEMIN_VUE.'formulaire_inscription.php';
 }

} else {

 // On affiche à nouveau le formulaire d'inscription
 include CHEMIN_VUE.'formulaire_inscription.php';
}
```

La vue associée

Code : PHP - ~/modules/membres/vues/formulaire_inscription.php

```php
<h2>Inscription au site</h2>

<?php

if (!empty($erreurs_inscription)) {

 echo '<ul>'."\n";

 foreach($erreurs_inscription as $e) {

  echo ' <li>'.$e.'</li>'."\n";
 }

 echo '</ul>';
}

echo $form_inscription;
```

Commentaires et explications

$form_inscription->is_valid($_POST) s'occupe de vérifier l'intégralité du formulaire (champs obligatoires, format de l'adresse e-mail, extension de l'avatar, etc.). Si la condition passe, tous les champs sont valides syntaxiquement.

On réalise ensuite les tests spécifiques à notre application : un mot de passe correct. On stocke l'erreur potentielle dans un tableau qui permettra à la vue de l'afficher (d'autres erreurs peuvent survenir plus tard, d'où l'utilisation d'un tableau et non d'une simple chaîne de caractères).

On notera que la vérification de l'unicité du nom d'utilisateur et de l'adresse e-mail se font plus tard.

En effet, il ne faut pas utiliser de requête du type `SELECT COUNT(*) FROM membres WHERE nom_utilisateur = :nom_utiliateur_souhaite` (avec un COUNT(*)) car si deux utilisateurs s'inscrivent en même temps avec le même pseudo, une erreur pourrait se produire, et la base de données nous renverrait un joli message d'erreur car nous avons défini ce champ en `UNIQUE` dans le schéma de notre table.
Nous allons plutôt utiliser ce message d'erreur retourné par MySQL lors de la requête `INSERT INTO` pour savoir si un utilisateur avec le même nom et / ou la même adresse e-mail existe déjà. Après tout, la base de données elle-même est mieux placée que nous pour savoir si l'utilisateur existe déjà ou non ! Elle n'a pas besoin de notre COUNT(*) (qui en plus peut poser des problèmes si 2 utilisateurs s'inscrivent en même temps avec le même nom ou la même adresse e-mail).

Fin de la vérification et traitement quand tout s'est bien passé

Il s'agit d'ajouter le nouveau membre dans la base de données, puis de générer le hash de validation, d'envoyer le mail pour confirmer l'inscription, de redimensionner et de déplacer l'avatar éventuel dans le bon dossier (ne pas le laisser dans le dossier temporaire) - si tout s'est bien passé . Le traitement de l'avatar utilise la librairie Image qui permet d'ouvrir, de redimensionner et de sauvegarder une image dans les formats PNG, GIF et JPG. Vous pouvez télécharger le code de la classe à cette adresse. Mettez le dans un fichier nommé ~/libs/image.php

À nouveau, je vous donne le code directement pour que vous puissiez vous habituer progressivement à la méthode de travail.

Le code du contrôleur

Code : PHP - ~/modules/membres/inscription.php (traitement du formulaire)

```
<?php
// Tire de la documentation PHP sur <http://fr.php.net/uniqid>
$hash_validation = md5(uniqid(rand(), true));

// Tentative d'ajout du membre dans la base de donnees
list($nom_utilisateur, $mot_de_passe, $adresse_email, $avatar) =
 $form_inscription->get_cleaned_data('nom_utilisateur', 'mdp',
'adresse_email', 'avatar');

// On veut utiliser le modele de l'inscription
(~/modeles/inscription.php)
include CHEMIN_MODELE.'inscription.php';

// ajouter_membre_dans_bdd() est défini dans ~/modeles/inscription.php
$id_utilisateur = ajouter_membre_dans_bdd($nom_utilisateur,
sha1($mot_de_passe), $adresse_email, $hash_validation);

// Si la base de données a bien voulu ajouter l'utliisateur (pas de
doublons)
if (ctype_digit($id_utilisateur)) {

 // On transforme la chaine en entier
 $id_utilisateur = (int) $id_utilisateur;
```

```
 // Preparation du mail
 $message_mail = '<html><head></head><body>
<p>Merci de vous être inscrit sur "mon site" !</p>
<p>Veuillez cliquer sur <a href="'.$_SERVER['PHP_SELF'].'?
module=membres&amp;action=valider_compte&amp;hash='.$hash_validation.'">ce
lien</a> pour activer votre compte !</p>
</body></html>';

 $headers_mail  = 'MIME-Version: 1.0'                              ."\r\n";
 $headers_mail .= 'Content-type: text/html; charset=utf-8'         ."\r\n";
 $headers_mail .= 'From: "Mon site" <contact@monsite.com>'         ."\r\n";

 // Envoi du mail
 mail($form_inscription->get_cleaned_data('adresse_email'), 'Inscription
sur <monsite.com>', $message_mail, $headers_mail);

 // Redimensionnement et sauvegarde de l'avatar (eventuel) dans le bon
dossier
 if (!empty($avatar)) {

  // On souhaite utiliser la librairie Image
  include CHEMIN_LIB.'image.php';

  // Redimensionnement et sauvegarde de l'avatar
  $avatar = new Image($avatar);
  $avatar->resize_to(100, 100); // Image->resize_to($largeur_maxi,
$hauteur_maxi)
  $avatar_filename = 'images/avatar/'.$id_utilisateur
.'.'.strtolower(pathinfo($avatar->get_filename(), PATHINFO_EXTENSION));
  $avatar->save_as($avatar_filename);

  // On veut utiliser le modele des membres (~/modeles/membres.php)
  include CHEMIN_MODELE.'membres.php';

  // Mise à jour de l'avatar dans la table
  // maj_avatar_membre() est défini dans ~/modeles/membres.php
  maj_avatar_membre($id_utilisateur , $avatar_filename);

 }

 // Affichage de la confirmation de l'inscription
 include CHEMIN_VUE.'inscription_effectuee.php';

// Gestion des doublons
} else {

 // Changement de nom de variable (plus lisible)
 $erreur =& $id_utilisateur;

 // On vérifie que l'erreur concerne bien un doublon
 if (23000 == $erreur[0]) { // Le code d'erreur 23000 siginife "doublon"
dans le standard ANSI SQL

  preg_match("`Duplicate entry '(.+)' for key \d+`is", $erreur[2],
$valeur_probleme);
  $valeur_probleme = $valeur_probleme[1];

  if ($nom_utilisateur == $valeur_probleme) {

   $erreurs_inscription[] = "Ce nom d'utilisateur est déjà utilisé.";

  } else if ($adresse_email == $valeur_probleme) {

   $erreurs_inscription[] = "Cette adresse e-mail est déjà utilisée.";

  } else {

   $erreurs_inscription[] = "Erreur ajout SQL : doublon non identifié
présent dans la base de données.";
  }
```

```
        } else {

          $erreurs_inscription[] = sprintf("Erreur ajout SQL : cas non traité
(SQLSTATE = %d).", $erreur[0]);
        }

        // On reaffiche le formulaire d'inscription
        include CHEMIN_VUE.'formulaire_inscription.php';
    }
```

Ça vous paraît pas mal ? Pourtant, **2 mauvaises pratiques ont été utilisées** dans ce code... Saurez-vous les identifier ?

Aparté sur les bonnes pratiques

Voyez-vous la ligne resize_to(100, 100) dans le code ? Rappelez-vous la partie sur les bonnes pratiques à appliquer...
Non, toujours pas ? Eh bien si ! 100 est un argument muet et on ne sait pas trop à quoi il correspond (imaginez que je n'aie pas mis le commentaire à côté)... La largeur absolue ? La hauteur maximale ? On ne sait d'ailleurs même pas quel argument correspond à la largeur et quel argument correspond à la hauteur...

Il faut donc créer 2 constantes pour éliminer toute ambiguïté dans le code. Rendez-vous dans votre fichier de configuration ~/global/config.php et rajoutez ces 2 constantes :

Code : PHP - ~/global/config.php (début du fichier)

```
<?php

// Identifiants pour la base de données. Nécessaires à PDO2.
// [...]

// Chemins à utiliser pour accéder aux vues/modèles/librairies
// [...]

// Configurations relatives à l'avatar
define('AVATAR_LARGEUR_MAXI', 100);
define('AVATAR_HAUTEUR_MAXI', 100);
```

Modifiez également le fichier ~/modules/membres/inscription.php afin d'y intégrer ces constantes : `<?php $avatar->resize_to(AVATAR_LARGEUR_MAXI, AVATAR_HAUTEUR_MAXI);` . On peut même se payer le luxe d'enlever le commentaire à côté, ça n'est plus nécessaire car nous avons auto-documenté le code.

Bonne pratique : utiliser des constantes pour éliminer les paramètres muets et les valeurs constantes présentes dans le code. Vous vous souvenez ?

Si vous regardez bien, il reste encore une valeur constante présente dans le code. C'est images/avatars/. Nous allons mettre en application dès maintenant une bonne pratique en créant une constante DOSSIER_AVATAR !

Direction notre fichier ~/global/config.php pour rajouter la définition de la constante : `<?php define('DOSSIER_AVATAR', 'images/avatars/');` . N'oubliez pas non plus de remplacer images/avatars/ par la constante dans le fichier ~/modules/membres/inscription.php.

En bonus, on remarquera que si on veut modifier la largeur ou la hauteur maximale autorisée, il suffit d'aller faire un tour dans ~/global/config.php et de modifier la valeur de ces constantes. Elles agissent comme une configuration pour notre application.

L'objectif de cet aparté était de vous apprendre à repérer et à éliminer les paramètres muets et les valeurs constantes. J'espère qu'à partir de maintenant vous y penserez et que vous n'en utiliserez (presque) plus jamais !

Les fonctions des modèles inscription et membres

Code : PHP - ~/modeles/inscription.php

```php
<?php

function ajouter_membre_dans_bdd($nom_utilisateur, $mdp,
$adresse_email, $hash_validation) {

 $pdo = PDO2::getInstance();

 $requete = $pdo->prepare("INSERT INTO membres SET
nom_utilisateur = :nom_utilisateur,
mot_de_passe = :mot_de_passe,
adresse_email = :adresse_email,
hash_validation = :hash_validation,
date_inscription = NOW()");

 $requete->bindValue(':nom_utilisateur', $nom_utilisateur);
 $requete->bindValue(':mot_de_passe',    $mdp);
 $requete->bindValue(':adresse_email',   $adresse_email);
 $requete->bindValue(':hash_validation', $hash_validation);

 if ($requete->execute()) {

  return $pdo->lastInsertId();
 }
 return $requete->errorInfo();
}
```

Code : PHP - ~/modeles/membres.php

```php
<?php

function maj_avatar_membre($id_utilisateur , $avatar) {

 $pdo = PDO2::getInstance();

 $requete = $pdo->prepare("UPDATE membres SET
avatar = :avatar
WHERE
id = :id_utilisateur");

 $requete->bindValue(':id_utilisateur', $id_utilisateur);
 $requete->bindValue(':avatar',         $avatar);

 return $requete->execute();
}
```

La vue associée

Code : HTML - ~/modules/membres/vues/inscription_effectuee.php

```html
<h2>Inscription confirmée</h2>

<p>L'inscription s'est déroulée avec succès !</p>

<p>Vous allez bientôt recevoir un mail vous permettant d'activer
votre compte afin de pouvoir vous connecter.</p>
```

Commentaires et explications

La vérification sur les doublons (nom d'utilisateur et / ou adresse e-mail) est faite à cet endroit. Je ne vous demanderai pas de savoir la refaire, mais essayez au moins de la comprendre, c'est toujours utile. Quelques cas bonus (qui ne devraient théoriquement pas arriver) sont même gérés : bref, c'est cadeau !

La fonction ajouter_membre_dans_bdd() nous retourne :

- soit l'id du nouvel utilisateur enregistré ;
- soit le tableau d'erreur fourni par errorInfo() (qui nous permet de vérifier les doublons).

Pour vérifier que la requête SQL s'est bien passée, on vérifie donc que le résultat de la fonction ajouter_membre_dans_bdd() est numérique. On aurait également pu vérifier que ça n'est pas un tableau (au choix), mais il est préférable de s'assurer que le résultat est un entier, car l'id d'un utilisateur DOIT être numérique. C'est donc logiquement et sémantiquement plus correct.

La fonction ajouter_membre_dans_bdd() est spécifique à la phase d'inscription. Théoriquement, aucun autre module n'en a besoin, c'est pourquoi nous la placerons dans un fichier de modèle séparé (~/modeles/inscription.php), que l'on utilise seulement pour l'inscription, plutôt que dans le modèle global portant le nom de la table (~/modeles/membres.php). Ceci permet d'alléger le modèle global "membres" de cette fonction. La fonction maj_avatar_membre() pourra nous resservir, elle trouve donc sa place dans le modèle ~/modeles/membres.php.

Le reste du code contient un envoi de mail classique (si vous ne saviez pas faire, c'est désormais le cas, mais je ne pense pas qu'il soit utile de s'y attarder) ainsi qu'une utilisation de la classe Image pour redimensionner l'avatar et l'enregistrer dans le bon dossier. Notez que la classe ne modifie pas l'image originale s'il n'y a pas besoin de le faire et que la transparence est préservée entre les formats compatibles (GIF et PNG).

La validation par mail

L'utilisateur doit maintenant recevoir un mail et cliquer sur le lien présent pour activer son compte. Le lien pointe vers index.php?module=membres&action=valider_compte&hash=[_le_hash_]. L'action appelée est donc "valider_compte", du module "membres", et correspond donc au fichier ~/modules/membres/valider_compte.php, qu'il vous faut créer.

Si vous avez bien suivi, lors de l'inscription, un hash de validation est généré et inséré dans la table membres de la base de données. Tant que ce hash est présent, le compte n'est pas validé. Si le hash est absent, c'est que le compte a été validé. Valider le compte revient donc à supprimer ce hash de la base de données. Libre à vous de faire autrement si vous le souhaitez !

Le code du contrôleur

Code : PHP - ~/modules/membres/valider_compte.php

```php
<?php

// On vérifie qu'un hash est présent
if (!empty($_GET['hash'])) {

 // On veut utiliser le modèle des membres (~/modeles/membres.php)
 include CHEMIN_MODELE.'membres.php';

 // valider_compte_avec_hash() est définit dans
~/modeles/membres.php
 if (valider_compte_avec_hash($_GET['hash'])) {

  // Affichage de la confirmation de validation du compte
  include CHEMIN_VUE.'compte_valide.php';

 } else {

  // Affichage de l'erreur de validation du compte
  include CHEMIN_VUE.'erreur_activation_compte.php';
 }
```

```
} else {

 // Affichage de l'erreur de validation du compte
 include CHEMIN_VUE.'erreur_activation_compte.php';
}
```

La fonction du modèle membres

Code : PHP - ~/modeles/membres.php

```
<?php

function valider_compte_avec_hash($hash_validation) {

 $pdo = PDO2::getInstance();

 $requete = $pdo->prepare("UPDATE membres SET
hash_validation = ''
WHERE
hash_validation = :hash_validation");

 $requete->bindValue(':hash_validation', $hash_validation);

 $requete->execute();

 return ($requete->rowCount() == 1);
}
```

Les vues associées

Code : HTML - ~/modules/membres/vues/compte_valide.php

```
<h2>Confirmation de validation d'un compte</h2>

<p>Le compte demandé a été validé avec succès !</p>
```

Code : HTML - ~/modules/membres/vues/erreur_activation_compte.php

```
<h2>Erreur d'activation d'un compte</h2>

<p>Le compte demandé n'a pas pu être activé, plusieurs raisons sont
possibles :
- le hash de validation est manquant ou invalide ;
- le compte a déjà été validé.</p>
```

Connexion et déconnexion
Préparation du terrain

Nous travaillons dans cette partie sur la connexion et la déconnexion d'un membre. Ceci correspond logiquement aux actions connexion et deconnexion du module membres. Créez donc les fichiers ~/modules/membres/connexion.php et ~/modules/membres/deconnexion.php.

Nous commençons par coder la connexion. Afin d'accéder facilement à cette page, nous allons l'ajouter dans le menu :

Code : HTML - ~/global/menu.php

```
<div id="menu">

  <h2>Menu</h2>

  <ul>
   <li><a href="index.php">Accueil</a></li>
  </ul>

  <h3>Espace membre</h3>
  <ul>
   <li><a href="index.php?
module=membres&amp;action=inscription">Inscription</a></li>
<li><a href="index.php?
module=membres&amp;action=connexion">Connexion</a></li>
  </ul>

 </div>
```

Enfin, à partir de maintenant nous aurons besoin d'utiliser les sessions. Rendez-vous donc directement dans le fichier ~/global/init.php pour les activer.

Code : PHP - ~/global/init.php (début du fichier)

```
<?php

// Inclusion du fichier de configuration (qui définit des
constantes)
include 'global/config.php';

// Utilisation et démarrage des sessions
session_start();

// Reste du fichier inchangé
```

Le formulaire de connexion

Celui-ci est particulièrement facile : c'est le même que l'inscription, avec plein de champs en moins ! Je vous le donne, profitez-en bien, ça ne durera pas !

Le code du contrôleur

Code : PHP - ~/modules/membres/connexion.php

```
<?php

// Ne pas oublier d'inclure la librairie Form
include CHEMIN_LIB.'form.php';

// "formulaire_connexion" est l'ID unique du formulaire
$form_connexion = new Form('formulaire_connexion');

$form_connexion->method('POST');

$form_connexion->add('Text', 'nom_utilisateur')
               ->label("Votre nom d'utilisateur");

$form_connexion->add('Password', 'mot de passe')
```

```
                    ->label("Votre mot de passe");

$form_connexion->add('Submit', 'submit')
                    ->value("Connectez-moi !");

// Pré-remplissage avec les valeurs précédemment entrées (s'il y en
a)
$form_connexion->bound($_POST);
```

La vérification du formulaire de connexion

Le code est très similaire à celui de l'inscription, mais - encore une fois - en plus simple.

Le code du contrôleur

Code : PHP - ~/modules/membres/connexion.php (suite)

```
<?php

// Création d'un tableau des erreurs
$erreurs_connexion = array();

// Validation des champs suivant les règles
if ($form_connexion->is_valid($_POST)) {

 list($nom_utilisateur, $mot_de_passe) =
  $form_connexion->get_cleaned_data('nom_utilisateur',
'mot_de_passe');

 // On veut utiliser le modèle des membres (~/modeles/membres.php)
 include CHEMIN_MODELE.'membres.php';

 // combinaison_connexion_valide() est définit dans
~/modeles/membres.php
 $id_utilisateur = combinaison_connexion_valide($nom_utilisateur,
sha1($mot_de_passe));

 // Si les identifiants sont valides
 if (false !== $id_utilisateur) {

  $infos_utilisateur = lire_infos_utilisateur($id_utilisateur);

  // On enregistre les informations dans la session
  $_SESSION['id']     = $id_utilisateur;
  $_SESSION['pseudo'] = $nom_utilisateur;
  $_SESSION['avatar'] = $infos_utilisateur['avatar'];
  $_SESSION['email']  = $infos_utilisateur['adresse_email'];

  // Affichage de la confirmation de la connexion
  include CHEMIN_VUE.'connexion_ok.php';

 } else {

  $erreurs_connexion[] = "Couple nom d'utilisateur / mot de passe
inexistant.";

  // On réaffiche le formulaire de connexion
  include CHEMIN_VUE.'formulaire_connexion.php';
 }

} else {

    // On réaffiche le formulaire de connexion
    include CHEMIN_VUE.'formulaire_connexion.php';
```

```
}
```

La fonction du modèle membres

Code : PHP - ~/modeles/membres.php

```
<?php

function combinaison_connexion_valide($nom_utilisateur,
$mot_de_passe) {

 $pdo = PDO2::getInstance();

 $requete = $pdo->prepare("SELECT id FROM membres
WHERE
nom_utilisateur = :nom_utilisateur AND
mot_de_passe = :mot_de_passe AND
hash_validation = ''");

 $requete->bindValue(':nom_utilisateur', $nom_utilisateur);
 $requete->bindValue(':mot_de_passe', $mot_de_passe);
 $requete->execute();

 if ($result = $requete->fetch(PDO::FETCH_ASSOC)) {

  $requete->closeCursor();
  return $result['id'];
 }
 return false;
}

function lire_infos_utilisateur($id_utilisateur) {

 $pdo = PDO2::getInstance();

 $requete = $pdo->prepare("SELECT nom_utilisateur, mot_de_passe,
adresse_email, avatar, date_inscription, hash_validation
FROM membres
WHERE
id = :id_utilisateur");

 $requete->bindValue(':id_utilisateur', $id_utilisateur);
 $requete->execute();

 if ($result = $requete->fetch(PDO::FETCH_ASSOC)) {

  $requete->closeCursor();
  return $result;
 }
 return false;
}
```

Les vues associées

Code : PHP - ~/modules/membres/vues/formulaire_connexion.php

```
<h2>Connexion au site</h2>

<p>Si vous n'êtes pas encore inscrit, vous pouvez le faire en <a
href="index.php?module=membres&amp;action=inscription">cliquant sur
```

```
ce lien</a>.</p>

<?php

if (!empty($erreurs_connexion)) {

 echo '<ul>'."\n";

 foreach($erreurs_connexion as $e) {

  echo ' <li>'.$e.'</li>'."\n";
 }

 echo '</ul>';
}

echo $form_connexion;
```

Code : PHP - ~/modules/membres/vues/connexion_ok.php

```
<h2>Confirmation de connexion</h2>

<p>Bienvenue, <?php echo $_SESSION['pseudo']; ?>.<br />
Vous êtes maintenant connecté !</p>
```

Commentaires et explications

Ça va commencer à devenir une habitude, $form_connexion->is_valid($_POST) nous assure que les champs sont bien remplis. Il faut ensuite faire les tests spécifiques à notre application, dans notre cas : vérifier que le couple nom d'utilisateur / mot de passe est correct !

La fonction combinaison_connexion_valide() retourne false si la combinaison nom d'utilisateur / mot de passe est mauvaise, et retourne l'id de l'utilisateur de succès. Nous avons besoin d'une deuxième fonction lire_infos_utilisateur() pour remplir la session.

Les informations sont enregistrées en SESSION si l'utilisateur a fourni les bonnes informations pour se connecter. Une erreur est affichée à l'utilisateur dans le cas contraire.

Vérifier que l'utilisateur est connecté

Étant donné que lors d'une connexion réussie, nous enregistrons les informations en session, il suffit de vérifier si la variable de session existe pour voir si l'utilisateur est connecté.

Code : PHP - Nulle part, code de démonstration

```
<?php

if (!empty($_SESSION['id'])) {

 // L'utilisateur connecté, on peut récupérer :
 // $_SESSION['id'] - son id utilisateur
 // $_SESSION['pseudo'] - son nom d'utilisateur
 // $_SESSION['avatar'] - son avatar (s'il existe)

} else {

 // Utilisateur non connecté
}
```

Comme vous allez le voir, nous aurons besoin d'utiliser cette vérification très régulièrement, et comme nous n'aimons pas répéter le code, nous allons créer une fonction pour le faire à notre place. Nous avons besoin de cette fonction partout, je propose donc de la mettre dans le fichier ~/global/init.php :

Code : PHP - ~/global/init.php (fin du fichier)

```php
<?php

// Début du fichier identique

// Vérifie si l'utilisateur est connecté
function utilisateur_est_connecte() {

return !empty($_SESSION['id']);
}
```

Modification du menu et des droits d'accès

Maintenant que l'on sait vérifier si l'utilisateur est connecté, nous allons afficher les liens d'inscription et de connexion du menu uniquement si l'utilisateur est connecté. Sinon, on affiche son pseudo (attention à la sécurité) et un lien de déconnexion.

Code : PHP - ~/global/menu.php

```php
<h3>Espace membre</h3>

<?php if (!utilisateur_est_connecte()) { ?>
<ul>
<li><a href="index.php?
module=membres&amp;action=inscription">Inscription</a></li>
<li><a href="index.php?
module=membres&amp;action=connexion">Connexion</a></li>
</ul>
<?php } else { ?>
<p>Bienvenue, <?php echo htmlspecialchars($_SESSION['pseudo']); ?
>.</p>
<ul>
<li><a href="index.php?
module=membres&amp;action=deconnexion">Déconnexion</a></li>
</ul>
<?php } ?>
```

Cependant cette "protection" n'est pas du tout suffisante... Si l'utilisateur est déjà connecté et qu'il tape à la main l'adresse index.php?module=membres&action=inscription, il pourra tout de même s'inscrire...
Il faut donc vérifier que l'utilisateur n'est pas connecté pour accéder aux actions inscription.php, connexion.php et valider_compte.php et que l'utilisateur est connecté pour accéder à deconnexion.php.
Ces vérifications seront désormais à ajouter sur toutes les pages ! Plutôt que de copier les fichiers de vues dans chaque dossier de module, nous allons placer ces vues dans un dossier commun, et donc en profiter pour créer une nouvelle constante CHEMIN_VUE_GLOBALE pour y accéder. Nous pouvons ainsi placer nos 2 nouveaux fichiers de vues erreur_deja_connecte.php et erreur_non_connecte.php dans le dossier ~/vues_globales/.

Allez donc dans le fichier ~/global/config.php afin de rajouter la définition de la constante `<?php define('CHEMIN_VUE_GLOBALE', 'vues_globales/'); .`

Voici le code à rajouter pour protéger les 3 pages inscription.php, connexion.php et valider_compte.php présentes dans le dossier ~/modules/membres/.

Code : PHP - inscription.php, valider_compte.php et connexion.php (début des fichiers)

```php
<?php

// Vérification des droits d'accès de la page
if (utilisateur_est_connecte()) {

 // On affiche la page d'erreur comme quoi l'utilisateur est déjà
connecté
 include CHEMIN_VUE_GLOBALE.'erreur_deja_connecte.php';

} else {

 // Reste de la page comme avant
}
```

Pour la page ~/modules/membres/deconnexion.php, c'est l'inverse :

Code : PHP - ~/modules/membres/deconnexion.php (début du fichier)

```php
<?php

// Vérification des droits d'accès de la page
if (!utilisateur_est_connecte()) {

 // On affiche la page d'erreur comme quoi l'utilisateur doit être
connecté pour voir la page
 include CHEMIN_VUE_GLOBALE.'erreur_non_connecte.php';

} else {

 // Reste de la page comme avant
}
```

Enfin, voici les codes des vues erreur_deja_connecte.php et erreur_non_connecte.php :

Code : HTML

```html
<h2>Accès interdit !</h2>

<p>Inutile d'accéder à cette page si vous êtes connecté.</p>
```

Code : HTML

```html
<h2>Accès interdit !</h2>

<p>Vous devez être connecté pour accéder à cette page.</p>
```

La connexion automatique

Un peu de théorie

Voici quelque chose de très pratique... mais de très risqué aussi. Une connexion automatique est très intéressante à mettre en place, car les exigences en terme de sécurité y sont élevées. En effet, nous allons utiliser les COOKIES, qui sont l'une des premières cibles des *hackeurs* lorsqu'ils veulent s'en prendre à votre site : leur but va être de voler les cookies d'un membre, afin

de le fouiller et voir les informations qu'ils peuvent en tirer. Je vais lister quelques cas courants pouvant se produire :

- Parfois, le mot de passe de l'utilisateur est stocké en clair dans le cookie. Le hackeur peut alors simplement se connecter avec le nom d'utilisateur connu et le mot de passe récupéré. C'est très mal.
- Parfois, le mot de passe est stocké en version hashé (par exemple, avec l'algorithme SHA-1). On peut penser que dans ce cas le hackeur ne peut rien faire, mais **c'est faux** ! De nombreux dictionnaires en ligne existent : ils stockent les hashs de nombreuses valeurs et il n'est plus rare de pouvoir récupérer le mot de passe à partir de la valeur hashée. Cette technique s'appelle un *brute-force* (test de toutes les possibilités jusqu'à ce que le mot de passe soit trouvé), et ces dictionnaires la rende très efficace !
- Enfin si ces deux solutions échouent, il reste un dernier recours au hackeur : utiliser une attaque de type *replay*. Elle consiste à répéter la séquence de connexion automatique qui a permis au possesseur du cookie de s'identifier. Cette attaque est très difficile à contrer simplement, c'est pourquoi la protection que nous allons mettre en place ne sera pas efficace à 100%.

De cette leçon, nous pouvons retenir qu'il faut que le cookie ne stocke pas directement le mot de passe, même hashé. Pour se protéger du *brute-force*, on utilise un *salt* (grain de sel). Ça consiste à ajouter au mot de passe une chaîne quelconque assez longue avant de le hasher, afin que les dictionnaires ne puissent pas retrouver la valeur originale du hash. Par exemple, si les dictionnaires connaissent la valeur de sha1('mdp_facile'), ils ne connaissent pas celle de sha1('grain_de_sel_long_et_compliqué_mdp_facile'). Ici, le grain de sel est rajouté devant, mais on peut également le rajouter derrière ou bien - mieux encore - à la fois devant et derrière.

Arrêtons de jouer les astronautes et revenons-en à notre but principal : connecter l'utilisateur automatiquement. Nous aurons besoin d'un cookie contenant 2 valeurs : l'id de l'utilisateur (pour savoir qui on veut connecter), et un hash unique **associé à l'utilisateur** (chaque utilisateur doit avoir un hash différent) : pour cela, nous utiliserons son nom d'utilisateur et son mot de passe. Le hash doit bien entendu être salé pour être protégé contre une attaque de type *brute-force*. Enfin, pour se protéger des attaques de type *replay*, il nous faut quelque chose d'unique à l'utilisateur et que l'on peut connaître à chaque fois qu'il visite le site. On pourrait d'abord penser à son adresse IP, mais ça ne fonctionnerait pas chez les gens dont l'IP change entre 2 connexions, nous utiliserons donc **l'identifiant fournit par son navigateur**. C'est là qu'on ne se protège pas à 100% contre l'attaque de type *replay* : si le hackeur utilise le même navigateur que notre victime (qui s'est fait volé ses cookies), alors il pourra quand même se connecter sur son compte... C'est à vous de voir si vous vous autorisez un tel risque. Rassurez-vous : si votre site ne contient pas de faille XSS (que vous connaissez dorénavant sur le bout des doigts), aucun hackeur n'aura de moyen simple de voler des cookies sur votre site.

Passons au code !

Code : PHP - ~/modules/membres/connexion.php

```php
<?php

// Ajoutons d'abord une case à cocher au formulaire de connexion
$form connexion->add('Checkbox', 'connexion_auto')
->label("Connexion automatique");
```

Code : PHP - ~/modules/membres/connexion.php

```php
<?php

 // Si les identifiants sont valides
 if (false !== $id_utilisateur) {

  $infos_utilisateur = lire_infos_utilisateur($id_utilisateur);

  // On enregistre les informations dans la session
  $_SESSION['id']     = $id_utilisateur;
  $_SESSION['pseudo'] = $nom_utilisateur;
  $_SESSION['avatar'] = $infos_utilisateur['avatar'];
  $_SESSION['email']  = $infos_utilisateur['adresse_email'];

// Mise en place des cookies de connexion automatique
if (false != $form_connexion->get_cleaned_data('connexion_auto'))
{
```

```
$navigateur = (!empty($ SERVER['HTTP_USER_AGENT'])) ?
$ SERVER['HTTP_USER_AGENT'] : '';
$hash cookie =
sha1('aaa'.$nom_utilisateur.'bbb'.$mot_de_passe.'ccc'.$navigateur.'ddd');

setcookie( 'id', $ SESSION['id'], strtotime("+1 year"), '/');
setcookie('connexion_auto', $hash_cookie, strtotime("+1 year"), '/');
}

  // Affichage de la confirmation de la connexion
  include CHEMIN_VUE.'connexion_ok.php';

 } else {

  $erreurs_connexion[] = "Couple nom d'utilisateur / mot de passe
inexistant.";

// Suppression des cookies de connexion automatique
setcookie('id', '');
setcookie('connexion_auto', '');

  // On réaffiche le formulaire de connexion
  include CHEMIN_VUE.'formulaire_connexion.php';
 }
```

Code : PHP - ~/global/init.php (ajout à la fin du fichier)

```
<?php

// Vérifications pour la connexion automatique

// On a besoin du modèle des membres
include CHEMIN_MODELE.'membres.php';

// Le mec n'est pas connecté mais les cookies sont là, on y va !
if (!utilisateur_est_connecte() && !empty($_COOKIE['id']) && !empty($_COOKIE['cc
{
 $infos_utilisateur = lire_infos_utilisateur($_COOKIE['id']);

 if (false !== $infos_utilisateur)
 {
  $navigateur = (!empty($_SERVER['HTTP_USER_AGENT'])) ? $_SERVER['HTTP_USER_AGEN
  $hash =
sha1('aaa'.$infos_utilisateur['nom_utilisateur'].'bbb'.$infos_utilisateur['mot_c

  if ($_COOKIE['connexion_auto'] == $hash)
  {
   // On enregistre les informations dans la session
   $_SESSION['id']     = $_COOKIE['id'];
   $_SESSION['pseudo'] = $infos_utilisateur['nom_utilisateur'];
   $_SESSION['avatar'] = $infos_utilisateur['avatar'];
   $_SESSION['email']  = $infos_utilisateur['adresse_email'];
  }
 }
}
```

Commentaires et explications

Notez que désormais, le modèle des membres est chargé sur TOUTES les pages, vous pouvez donc le supprimer des autres pages qui en ont besoin.

Au niveau du sel, remplacez bien évidemment aaa bbb ccc et ddd par des valeurs assez **longues** et **compliquées** (exemple : °savageman°rox}trop[ ) et différentes pour chaque sel (celui présent au début, ceux du milieu et celui à la fin).

Vous voilà (normalement) parés pour un système sécurisé de connexion automatique.

Déconnexion

Nous terminons par le plus simple pour cette partie. Nous n'avons en effet pas besoin de formulaire pour déconnecter l'utilisateur ; il suffit de détruire sa session. Nous n'oublierons pas de supprimer les cookies concernant la connexion automatique, sinon l'utilisateur l'ayant activée se verrait à nouveau connecté immédiatement. Voici le code pour y parvenir :

Le code du contrôleur

Code : PHP - ~/modules/membre/deconnexion.php

```php
<?php

// Suppression de toutes les variables et destruction de la session
$_SESSION = array();
session_destroy();

// Suppression des cookies de connexion automatique
setcookie('id', '');
setcookie('connexion_auto', '');

include CHEMIN_VUE.'deconnexion_ok.php';
```

La vue associée

Code : PHP - ~/modules/membres/vues/deconnexion_ok.php

```php
<h2>Confirmation de déconnexion</h2>

<p>Vous êtes maintenant déconnecté.<br />
<a href="<?php echo $_SERVER['HTTP_REFERER']; ?>">Revenir à la page
précédente</a><br />
<a href="index.php">Revenir à la page d'accueil</a></p>
```

Vous avez désormais un espace membre certe assez minimaliste, mais fonctionnel. J'espère que ça vous a plu !

Bonus : le profil d'un membre

[Mini TP] Affichage d'un profil utilisateur
Affichage d'un profil [Premier TP]

Introduction et objectifs du TP

Afficher un profil, c'est bien gentil, mais nous avons plusieurs membres sur le site : il faut donc pouvoir choisir le profil de quel membre nous souhaitons afficher. Cette page aura donc besoin d'un argument GET dans l'URL, nous l'appellerons id et contiendra l'id de l'utilisateur pour lequel nous souhaitons afficher le membre.

Vous êtes (normalement) désormais assez préparés pour coder cette page tout seuls, comme des grands ! C'est pourquoi je vous propose un mini-TP, afin que **vous** puissiez travailler ! Voici les consignes :

- la page affichera les informations suivantes : nom d'utilisateur, avatar, adresse e-mail et date d'inscription ;
- si un utilisateur connecté regarde son profil, on lui proposera de l'éditer avec un lien vers l'action modifier_profil (qui sera codé dans la sous-partie suivante) ;
- vous gèrerez le cas d'erreur où le membre n'existe pas.

Plutôt trivial, non ? Assez parlé, au travail ! La correction est disponible un peu plus bas...

.

..

...

....

.....

......

....

....

....

....

Ne trichez pas, c'est pour votre bien ! Voici la correction.

Premièrement, l'affichage du profil est une nouvelle action pour le module membres ! Il faut donc créer une page afficher_profil.php dans le dossier ~/modules/membres/. Nous ne l'ajouterons pas dans le menu, l'affichage du profil se fera depuis des liens sur d'autres pages du site. Si vous le souhaitez, vous pouvez tout de même ajouter un lien pour que l'utilisateur affiche son propre profil (en utilisant $_SESSION['id'] comme attribut GET nommé id dans le lien) dans le menu.

Code : PHP - ~/modules/membres/afficher_profil.php

```php
<?php

// Pas de vérification de droits d'accès nécessaire : tout le monde
peut voir un profil utilisateur :)

// Si le paramètre id est manquant ou invalide
if (empty($_GET['id']) or !is_numeric($_GET['id'])) {

 include CHEMIN VUE.'erreur parametre profil.php';
```

```
} else {

 // On veut utiliser le modèle des membres (~/modules/membres.php)
 include CHEMIN_MODELE.'membres.php';

 // lire_infos_utilisateur() est défini dans ~/modules/membres.php
 $infos_utilisateur = lire_infos_utilisateur($_GET['id']);

 // Si le profil existe et que le compte est validé
 if (false !== $infos_utilisateur &&
$infos_utilisateur['hash_validation'] == '') {

  list($nom_utilisateur, ,$avatar, $adresse_email,
$date_inscription, ) = $infos_utilisateur;
  include CHEMIN_VUE.'profil_infos_utilisateur.php';

 } else {

  include CHEMIN_VUE.'erreur_profil_inexistant.php';
 }
}
```

Code : PHP - ~/modules/membres/vues/profil_infos_utilisateur.php

```
<h2>Profil de <?php echo htmlspecialchars($nom_utilisateur); ?></h2>

<p>
 <img class="flottant_droite" src="<?php echo
DOSSIER_AVATAR.urlencode($avatar); ?>" title="Avatar de <?php echo
htmlspecialchars($nom_utilisateur); ?>" />
 <span class="label_profil">Adresse email</span> : <?php echo
htmlspecialchars($adresse_email); ?><br />
 <span class="label_profil">Date d'inscription</span> : <?php echo
$date_inscription; ?>
</p>
```

Code : HTML - ~/modules/membres/vues/erreur_profil_inexistant.php

```
<h2>Erreur d'affichage du profil</h2>
<p>Cet utilisateur n'existe pas.</p>
```

Code : CSS - ~/style/global.css

```
.flottant_droite {

 float: right;
}
span.label_profil {

 font-weight: bold;
}
```

Commentaires et explications

C'est une page d'affichage, ça reste donc assez simple. En plus, la fonction lire_infos_utilisateur() existait déjà car on s'en sert

pour la connexion . On remarquera seulement que la structure du language list() peut prendre des arguments vides pour les valeurs que l'on ne veut pas récupérer (mot_de_passe et hash_validation dans notre cas).

Pensez juste à ne pas oublier d'utiliser la fonction htmlspecialchars() dans vos vues sur les données saisies par l'utilisateur, ou vous seriez vulnérables face à la faille XSS (vol de cookie, puis de compte) ! Celui-ci choisit son nom d'utilisateur et son adresse e-mail, il faut donc protéger ces variables à l'affichage.

Modification d'un profil utilisateur
Modification d'un profil

Voici une partie (un peu) plus difficile, mais également plus intéressante ! Cependant, je vais vous guider, car il y aura plusieurs subtilités.

L'objectif est de fournir à l'utilisateur une interface lui permettant de modifier les informations concernant son profil.

Un membre, c'est quoi ? Réponse : un id, un nom d'utilisateur, un mot de passe, une adresse e-mail, un avatar et une date d'inscription. Parmi ces 6 champs, nous en rendrons modifiables seulement 3 : mot de passe, adresse e-mail et avatar ! En effet, l'id et la date d'inscription sont fixés lors de l'inscription ; quant au nom d'utilisateur, c'est un choix que de ne pas permettre à l'utilisateur de le modifier. En tout cas, ça simplifie déjà le travail par deux, étant donné que l'on a seulement 3 champs à traiter sur les 6 ! Mais... ça n'est pas si simple que ça !

Le mot de passe étant une information très sensible (critique), il va falloir demander l'ancien afin de confirmer que c'est bien le membre en question qui est connecté.
Au niveau de l'adresse e-mail et de l'avatar, ça restera classique. Nous ferons juste attention aux doublons possibles pour l'adresse e-mail.

Dans cette optique, nous ferons 2 formulaires différents sur la même page : le premier pour modifier l'adresse e-mail et l'avatar. Le second pour modifier le mot de passe.

Je vous laisse créer l'action modifier_profil dans le module membres et y ajouter la vérification des droits d'accès (il faut que le membre soit connecté, vous l'aurez bien entendu deviné !). On ajoutera également la page dans le menu. Je ne vous passe pas les codes, vous avez j'en suis sûr réussi le TP précédent avec succès et je vous ai montré 2 fois comment faire avant !

Mise en place des formulaires

On va très largement s'inspirer du formulaire d'inscription et utiliser les champs qui nous conviennent.

Code : PHP - ~/modules/membres/modifier_profil.php

```php
<?php

// Vérification des droits d'accès de la page
if (!utilisateur_est_connecte()) {

 // On affiche la page d'erreur comme quoi l'utilisateur doit être
connecté pour voir la page
 include CHEMIN_VUE_GLOBALE.'erreur_non_connecte.php';

} else {

 // Ne pas oublier d'inclure la librairie Form
 include CHEMIN_LIB.'form.php';

 // "form_modif_infos" est l'ID unique du formulaire
 $form_modif_infos = new Form("form_modif_infos");

 $form_modif_infos->add('Email', 'adresse_email')
                  ->label("Votre adresse email")
                  ->Required(false)
                  ->value($_SESSION['email']);

 $form_modif_infos->add('Checkbox', 'suppr_avatar')
                  ->label("Je veux supprimer mon avatar")
```

```
                            ->Required(false);

        $form_modif_infos->add('File', 'avatar')
                            ->filter_extensions('jpg', 'png', 'gif')
                            ->max_size(8192) // 8 Kb
                            ->label("Votre avatar (facultatif)")
                            ->Required(false);

        $form_modif_infos->add('Submit', 'submit')
                            ->initial("Modifier ces informations !");

        // "form_modif_mdp" est l'ID unique du formulaire
        $form_modif_mdp = new Form("form_modif_mdp");

        $form_modif_mdp->add('Password', 'mdp_ancien')
                            ->label("Votre ancien mot de passe");

        $form_modif_mdp->add('Password', 'mdp')
                            ->label("Votre nouveau mot de passe");

        $form_modif_mdp->add('Password', 'mdp_verif')
                            ->label("Votre nouveau mot de passe
    (vérification)");

        $form_modif_mdp->add('Submit', 'submit')
                            ->initial("Modifier mon mot de passe !");

        // Création des tableaux des erreurs (un par formulaire)
        $erreurs_form_modif_infos = array();
        $erreurs_form_modif_mdp   = array();

        // et d'un tableau des messages de confirmation
        $msg_confirm = array();

        // Validation des champs suivant les règles en utilisant les
    données du tableau $_POST
        if ($form_modif_infos->is_valid($_POST)) {

          list($adresse_email, $suppr_avatar, $avatar) = $form_modif_infos-
    >get_cleaned_data('adresse_email', 'suppr_avatar', 'avatar');

          // On veut utiliser le modèle de
    l'inscription(~/modules/membres.php)
          include CHEMIN_MODELE.'membres.php';

          // Si l'utilisateur veut modifier son adresse e-mail
          if (!empty($adresse_email)) {

           $test = maj_adresse_email_membre($_SESSION['id'],
    $adresse_email);

           if (true === $test) {

            // Ça a marché, trop cool !
            $msg_confirm[] = "Adresse e-mail mise à jour avec succès !";

           // Gestion des doublons
           } else {

            // Changement de nom de variable (plus lisible)
            $erreur =& $test;

            // On vérifie que l'erreur concerne bien un doublon
            if (23000 == $erreur[0]) { // Le code d'erreur 23000 signifie
    "doublon" dans le standard ANSI SQL

             preg_match("`Duplicate entry '(.+)' for key \d+`is",
    $erreur[2], $valeur_probleme);
             $valeur_probleme = $valeur_probleme[1];
```

```
            if ($adresse_email == $valeur_probleme) {

             $erreurs_form_modif_infos[] = "Cette adresse e-mail est déjà
utilisée.";

            } else {

             $erreurs_form_modif_infos[] = "Erreur ajout SQL : doublon non
identifié présent dans la base de données.";
            }

           } else {

            $erreurs_form_modif_infos[] = sprintf("Erreur ajout SQL : cas
non traité (SQLSTATE = %d).", $erreur[0]);
           }

         }
        }

        // Si l'utilisateur veut supprimer son avatar...
        if (!empty($suppr_avatar)) {

         maj_avatar_membre($_SESSION['id'], '');
         $_SESSION['avatar'] = '';

         $msg_confirm[] = "Avatar supprimé avec succès !";

        // ... ou le modifier !
        } else if (!empty($avatar)) {

         // On souhaite utiliser la librairie Image
         include CHEMIN_LIB.'image.php';

         // Redimensionnement et sauvegarde de l'avatar
         $avatar = new Image($avatar);
         $avatar->resize_to(100, 100); // Image->resize_to($largeur_maxi,
$hauteur_maxi)
         $avatar_filename = DOSSIER_AVATAR.$id_utilisateur
.'.'.strtlower(pathinfo($avatar->get_filename(),
PATHINFO_EXTENSION));
         $avatar->save_as($avatar_filename);

         // On veut utiliser le modèle des membres (~/modules/membres.php)
         include CHEMIN_MODELE.'membres.php';

         // Mise à jour de l'avatar dans la table
         // maj_avatar_membre() est définit dans ~/modules/membres.php
         maj_avatar_membre($_SESSION['id'] , $avatar_filename);
         $_SESSION['avatar'] = $avatar_filename;

         $msg_confirm[] = "Avatar modifié avec succès !";
        }

       } else if ($form_modif_mdp->is_valid($_POST)) {

        // On vérifie si les 2 mots de passe correspondent
        if ($form_modif_mdp->get_cleaned_data('mdp') != $form_modif_mdp-
>get_cleaned_data('mdp_verif')) {

         $erreurs_form_modif_mdp[] = "Les deux mots de passes entrés sont
différents !";

        // C'est bon, on peut modifier la valeur dans la BDD
        } else {


         // On veut utiliser le modèle de l'inscription
(~/modules/membres.php)
         include CHEMIN_MODELE.'membres.php';
```

```
            maj_mot_de_passe_membre($_SESSION['id'], $mdp);

            $msg_confirm[] = "Votre mot de passe a été modifié avec succès
!";
           }

         }
        }

        // Affichage des formulaires de modification du profil
        include ('formulaires_modifier_profil.php');
```

Fonctions du modèle membres

maj_avatar_membre() existe déjà. Il faut donc seulement ajouter maj_email_membre() et maj_mot_de_passe_membre(). Allez, je vous laisse faire : il n'y a absolument aucun piège, c'est presque du recopiage de maj_avatar_membre(). Faites juste attention à ne pas oublier d'utiliser sha1() pour crypter le mot de passe ! (Comme dans ajouter_membre_dans_bdd() du fichier ~/modeles/inscription.php.)

La vue utilisée

Code : PHP - ~/modules/membres/vues/formulaires_modifier_profil.php

```
<h2>Modification de votre profil utilisateur</h2>

<?php

if (!empty($msg_confirm)) {

 echo '<ul>'."\n";

 foreach($msg_confirm as $m) {

  echo ' <li>'.$m.'</li>'."\n";
 }

 echo '</ul>';
}

if (!empty($erreurs_form_modif_infos)) {

 echo '<ul>'."\n";

 foreach($erreurs_form_modif_infos as $e) {

  echo ' <li>'.$e.'</li>'."\n";
 }

 echo '</ul>';
}

$form_modif_infos->fieldsets("Modification de l'e-mail et de
l'avatar", array('adresse_email', 'suppr_avatar', 'avatar'));

echo $form_modif_infos;

if (!empty($erreurs_form_modif_mdp)) {

 echo '<ul>'."\n";

 foreach($erreurs_form_modif_mdp as $e) {

  echo ' <li>'.$e.'</li>'."\n";
 }

 echo '</ul>';
```

```
        }

        $form_modif_mdp->fieldsets("Modification du mot de passe",
        array('mdp_ancien', 'mdp', 'mdp_verif'));

        echo $form_modif_mdp;
```

Commentaires et explications

Ouf, c'était plus long que d'habitude ! C'est normal, nous avons deux formulaires au lieu d'un seul auparavant. D'ailleurs, la création des formulaires étant aisée, je ne m'attarderai pas dessus.

N'oubliez pas la vérification des droits dans votre contrôleur ! Pour modifier son profil, il faut bien entendu être connecté !

Nous avons une nouveauté : $messages_confirmation. Pour le formulaire de modification des informations (adresse e-mail et avatar), la mise à jour de l'avatar peut réussir, alors que celle de l'adresse e-mail échoue. Il faut donc un nouveau tableau pour un nouveau type de messages : ceux qui indiquent à l'utilisateur qu'une action s'est bien déroulée. Avant, nous utilisions une vue spéciale pour afficher la confirmation d'un formulaire. Ici, le traitement du formulaire peut échouer (tentative de mise à jour de l'adresse e-mail pour une déjà existante, ce qui provoque **erreur de doublon**), alors que d'autres actions ont réussi (mise à jour de l'avatar). Nous affichons donc tout sur la même page : c'est presque plus simple, étant donné que nous avons une seule vue au lieu de plusieurs !
Étant donné qu'un seul des deux formulaires peut être validé à la fois, nous aurions pu utiliser un seul tableau pour stocker les messages d'erreur. Cependant, pour afficher les messages au-dessus du bon formulaire, il est plus simple d'utiliser deux tableaux distincts.

Je crois qu'il n'y a pas besoin de plus d'explications, tout le reste a déjà été vu avant !

Ça n'est bien entendu pas terminé. La prochaine partie devrait concerner la mise en place d'un petit livre d'or sous forme de TP. Un problème, une question ? J'ai mis en place un topic dédié au tutoriel dans le forum PHP, n'hésitez pas à y faire un tour. Vous pouvez également m'envoyer un MP pour les demandes particulières.