Sessions et MVC en php

Serge Rosmorduc

19 mai 2010

1 Sessions en php

Dans le protocole HTTP d'origine, chaque requête d'un utilisateur est *a priori* indépendante des précédentes. Le principe fondamental est que l'utilisateur envoie une requête, et que le serveur expédie le résultat, généralement sous forme de code HTML. La requête suivante du même utilisateur n'aura pour le serveur aucun lien avec la précédente.

C'est tout à fait adapté à la navigation hypertexte, mais beaucoup moins aux applications web. Par exemple, un système d'achat en ligne a besoin de se rapeller d'une requête à l'autre quels sont les achats de l'utilisateur.

On a donc développé plusieurs mécanismes pour résoudre ce problème. Le premier, ce sont les cookies. Ils sont cependant assez limités :

- leur taille est réduite ;
- ils circulent sur le web à chaque requête ;
- ils sont stockés sur la machine de l'utilisateur : celui-ci peut éventuellement les manipuler à sa guise (problème de sécurité).

Le mécanisme des sessions permet de conserver des données côté *serveur.* La session est identifiée par un numéro. À chaque requête, l'utilisateur transmet son numéro de session, en général à l'aide d'un cookie (on peut éventuellement le transmettre aussi en mode GET, dans l'URL).

Il existe un risque de *vol de session,* puisqu'il suffit a priori de connaître le numéro de la session pour se faire passer pour l'utilisateur de celle-ci. Les applications sécurisées peuvent ajouter généralement des vérifications supplémentaires (données d'identifications transmises à chaque fois, par exemple).

1.1 Création et destruction de session

1.1.1 Création

Pour qu'un script php fonctionne dans le cadre d'une session, il doit appeler `session_start()`. Cette commande démarre une nouvelle session s'il n'en existe pas, et charge les données de la session courante dans le cas contraire.

`session_start()` doit impérativement être appelée avant que le script php n'ait écrit quoique ce soit (elle manipule les en-têtes http). En conséquence, on l'écrit au début du script (rien n'interdit d'avoir du code php avant, mais celui-ci ne doit produire aucun affichage).

```
1 <?php
2
3 session_start();
```

1.1.2 Destruction

(cette partie est là plutôt à titre de référence qu'à titre de cours)
Pour supprimer totalement la session en cours, on doit :
- effacer `$_SESSION` ;
- se débarasser du cookie associé à la session ;
- appeler `session_destroy()`.

Le code suivant [1] réalise cette tâche :

```
1  $_SESSION = array();
2
3  // Suppression du cookie si nécessaire.
4  if (ini_get("session.use_cookies")) {
5      $params = session_get_cookie_params();
6      setcookie(session_name(), '', time() - 42000,
7          $params["path"], $params["domain"],
8          $params["secure"], $params["httponly"]
9      );
10 }
11
12 // Finalement, on détruit la session.
13 session_destroy();
```

Dans la plupart des cas, on peut attendre tranquillement que l'utilisateur se déconnecte ou se contenter de réinitialiser les parties importantes du tableau `$_SESSION`.

La durée de vie des sessions est limitée par plusieurs facteurs :
- la durée de vie du cookie de session sur le client. On appelle `session_set_cookie_params` en lui passant comme argument la dirée de vie du cookie, en secondes. Cet appel doit précéder l'appel à `session_start`.

```
1  session_set_cookie_params(60); // durée de vie : 1 minute (!)
2  session_start();
```

  Attention, comme toutes les techniques qui fonctionnent côté client, celle-ci n'est pas fiable. Si le client ne collabore pas, il peut conserver son cookie.
- la durée de vie des variables de session. Celle-ci est manipulable à travers divers paramètres de php, qui doivent, comme précédemment, être fixés avant l'appel à `session_start`.
  Il 'agit de `session.gc_maxlifetime`, `session.gc_probability` et `session.gc_divisor"`. En gros, les deux derniers contrôlent la probabilité qu'a php de lancer un ramasse-miette pour les variables de session*lors du chargement d'une page* (la probabilité est $\frac{\texttt{session.gc\_probability}}{\texttt{session.gc\_divisor}}$; le premier contrôle la durée de vie de la session. Ainsi donc, si une session est inactive depuis plus de `session.gc_maxlifetime` secondes, et qu'une nouvelle page est chargée *et que le ramasse-miettes se lance,* la session sera détruite. On voit que le procédé est assez aléatoire.
- on peut enfin stocker dans la session elle même la date de sa dernière utilisation :

```
1  $_SESSION['dernierAccess']= time();
```

  Il suffit alors, quand on charge une nouvelle page, de comparer la date de dernier accès et la date courante. Si l'écart est trop important, on ferme la session. cette

1. tiré de la documentation de php

technique ne permet pas de libérer l'espace occupé par les variables de sessions. Il s'agit plutôt de se protéger contre le cas de figure où un utilisateur quitte son ordinateur sans fermer sa sessio n, pour éviter que l'utilisateur suivant puisse en profiter.

1.2 Utilisation de la session

Dans une session, le programmeur a accès au tableau global `$_SESSION`. Celui-ci est accessible partout (sans avoir besoin de le déclarer comme `global`).

En principe, le tableau `$_SESSION` a comme durée de vie la connexion de l'utilisateur, c'est à dire que la session est détruite quand l'utilisateur ferme son navigateur. Les données placée dans le tableau session sont donc conservées durant tout ce temps.

On place très souvent dans la session des données sur l'utilisateur courant ; les applications commerciales utilisent aussi la session pour conserver la liste des produits commandés par l'utilisateur (son « panier »).

Exemple simple : un compteur.

```
1  <?
2  session_start();
3
4  if (isset($_SESSION['compte']))
5    $_SESSION['compte']++;
6  else
7    $_SESSION['compte']= 1;
8  ?>
9  <html>
10 <body>
11 <?php echo "bonjour, c'est la visite numéro ". $_SESSION['compte'];?>
12 </body>
13 </html>
```

1.3 Quelques précisions

1.3.1 Rappels sur les tableaux

`$_SESSION` est un tableau. En `PHP`, les tableaux sont manipulés par *valeur*. Ce qui signifie que le code suivant :

```
1  $t= $_SESSION;
```

crée une *copie* de `$_SESSION` et que les modifications de `t` n'auront aucune influence sur session.

De même :

```
1  ...
2  if (! isset($_SESSION['panier'])) {
3    $_SESSION['panier']=
4  array();
5  }
6
7  $panier= $_SESSION['panier'];
8  $panier[]= ....;
```

ne donnera pas le résultat escompté, car `$panier` est simplement une *copie* de `$_SESSION['panier']`.

On peut se tirer d'affaire avec une référence :

```
1  $panier= &$_SESSION['panier'];
2  $panier[]= ....;
```

Mais le plus simple est, soit d'utiliser directement `$_SESSION`, soit de passer par des *objets*, car, contrairement aux tableaux, ils sont manipulés par adresse et non pas par valeur.

1.3.2 Sessions et objets

Pour pouvoir ranger sans problème un objet dans une session, il faut que la classe de l'objet soit définie *avant* le début de la session.

```
1  <?php
2  // La classe doit être définie...
3  class Panier {
4    private $contenu=array();
5
6    function ajouter($v) {
7      $this->contenu[]= $v;
8    }
9
10   function getContenu() {
11     return $this->contenu;
12   }
13 }
14
15 // avant le "session_start".
16 session_start();
17
18 if (! isset($_SESSION['panier'])) {
19   $_SESSION['panier']= new Panier();
20  }
21
22 $p= $_SESSION['panier'];
23 $p->ajouter('b');
24 ?>
25 <html>
26 <body>
27 <?php
28         echo "<p>";
29         foreach ($p->getContenu() as $produit) {
30           echo $produit;
31         }
32 ?>
33 </body>
34 </html>
```

2 Architecture d'une application

2.1 Introduction

Quand un programme commence à dépasser une certaine taille, il devient capital qu'il soit bien *structuré* . Chaque partie du programme doit traiter une tâche bien définie, et être aussi simple que possible. La communication entre les diverses parties doit être claire.

Un programme mal structuré est difficile à comprendre, à corriger, à faire évoluer. Il se prête mal à la réutilisation du code, et on risque de réinventer continuellement la roue.

C'est vrai dans n'importe quel langage de programmation, et c'est particulièrement vrai en PHP, car la structure même du langage (pages web « embarquant » du code) encourage un style de programmation fort peu structuré.

On désire généralement séparer le code qui effectue des calculs (la « logique » du programme) de la partie qui intéragit directement avec l'utilisateur (récupération et affichage de données). L'intérêt de cette séparation est multiple :

- tout d'abord, chacune de ces fonctionnalités est suffisament complexe. En les mélangeant, on obtient inévitablement un code illisible. De plus, cette séparation permet de développer et de tester chacune des parties séparément, en confiant éventuellement les tâches à des équipes différentes.
- ensuite, en isolant chacune des parties, on la rend plus souple. Supposons par exemple un site de recettes de cuisines. Si la partie qui permet de sauvegarder une recette dans la base est indépendante du formulaire correspondant, il devient alors facile de créer des fonctionnalités supplémentaires, permettant par exemple d'importer d'un bloc une liste de recettes stoquée dans un fichier XML.

L'architecture MVC est un des moyens de parvenir à cette séparation.

2.2 Premières définitions

Interface utilisateur partie du programme qui s'occupe d'afficher les données vues par l'utilisateur, et de récupérer les données fournies par celui-ci ;

Contrôle partie de l'interface utilisateur qui reçoit les requêtes de l'utilisateur, et déclenche les traitements nécessaires.

Les traitements eux-mêmes ne font normallement pas partie du contrôle.

Logique applicative traitements liés à une application particulière.

Logique métier/Modèle représentation du domaine sur lequel travaille le programme et traitements associés. Par exemple, dans un logiciel de commerce électronique, la représentation des commandes, des produits à commander, etc... sera dans le modèle. En orienté objet, une grande partie des traitements est effectuée par le modèle lui-même.

Persistance partie du programme qui a pour tâche de sauvegarder et relire les informations du modèle que l'on souhaite conserver, typiquement dans une base de données.

2.3 Séparation du modèle et de la vue : première approche

Le premier point, et sans doute le plus important, est de bien séparer la partie qui réalise les traitements de celle qui fait l'affichage.

Considérons le programme suivant :

```
1  <html>
2  <head><title>exemple sans séparation</title></head>
3  <body>
4
5  <p> liste des articles : </p>
6  <?php
7  $db= new PDO("mysql:host=localhost;dbname=publi", 'xxxx', 'xxxx');
8  $st = $db->query("SELECT * FROM article");
9
10 while ($obj= $st->fetchObject()) : ?>
11   <h3><em><?php echo htmlspecialchars($obj->id_article); ?></em>
   <?php echo htmlspecialchars($obj->titre); ?></h3>
12   <p> <?php echo htmlspecialchars($obj->texte); ?>
13   </p>
14 <?php endwhile; ?>
15 </body>
16 </html>
```

Il affiche une liste d'articles (textes avec un titre). Le programme est court, mais difficile à lire. Il mélange le HTML et le PHP. De plus, la partie mise en page n'est pas réutilisable. Dans notre cas précis, la requête liste tous les articles. Si nous voulons, dans une autre page, lister uniquement une partie de ceux-ci, il faudra recopier le code HTML, ce qui nous posera des problèmes quand il s'agira de donner au site un aspect uniforme.

Le principe pour séparer mise en page et traitements (au sens large du terme) est de stocker dans un tableau les données à afficher. L'affichage se fera alors en deux temps :

1. calcul des données à afficher. Création du tableau global `$VUE` ;
2. affichage des données contenues dans `$VUE`.

En suivant ces principes, le code devient :

```
1  <?php
2  $db= new PDO("mysql:host=localhost;dbname=publi", 'xxxx', 'xxxx');
3  $st = $db->query("SELECT * FROM article");
4  $VUE= array();
5
6  while ($obj= $st->fetchObject()) {
7    $VUE['articles'][]= array(
8      'id_article' => htmlspecialchars($obj->id_article),
9       'titre' =>  htmlspecialchars($obj->titre),
10      'texte' => htmlspecialchars($obj->texte)
11   );
12 }
13 ?>
14 <html>
15 <head><title>exemple avec séparation</title></head>
16 <body>
17
18 <p> liste des articles : </p>
19 <?php foreach ($VUE['articles'] as $art): ?>
20
21 <h3><em><?php echo $art['id_article']; ?></em>  <?php echo $art['titre']; ?></h3>
```

```
22    <p> <?php echo $art['texte']; ?>
23     </p>
24  <?php endforeach; ?>
25  </body>
26  </html>
```

La partie HTML ne contient plus que le strict minimum de php. Par ailleurs, dans cet exemple, nous avons veillé à ce que le tableau `$VUE` ne comporte que du texte directement affichable en HTML (protégé par `htmlspecialchars()`. La personne qui écrit le HTML peut donc le faire en toute confiance.

L'étape suivante, c'est la séparation physique des deux codes. On aura donc

```
1   <?php
2   $db= new PDO("mysql:host=localhost;dbname=publi", 'xxxx', 'xxxx');
3   $st = $db->query("SELECT * FROM article");
4   $VUE= array();
5
6   while ($obj= $st->fetchObject()) {
7     $VUE['articles'][]= array(
8       'id_article' => htmlspecialchars($obj->id_article),
9        'titre' =>  htmlspecialchars($obj->titre),
10       'texte' => htmlspecialchars($obj->texte)
11    );
12  }
13
14  require('pageListeArticle.php');
15  ?>
```

```
1   <html>
2   <head><title>exemple avec séparation</title></head>
3   <body>
4
5   <p> liste des articles : </p>
6   <?php foreach ($VUE['articles'] as $art): ?>
7
8   <h3><em><?php echo $art['id_article']; ?></em>  <?php echo $art['titre']; ?></h3>
9     <p> <?php echo $art['texte']; ?>
10     </p>
11  <?php endforeach; ?>
12  </body>
13  </html>
```

le fichier "`pageListeArticle.php`" est alors utilisable par d'autres scripts php, si nécessaire.

On peut facilement le tester en construisant une liste d'articles « à la main » :

```
1   <?php
2   $VUE['articles'][]= array('id_article'=> 1, 'titre' => 'premier',
3   'texte' => "un texte");
4   $VUE['articles'][]= array('id_article'=> 2, 'titre' => 'premier',
5   'texte' => "un texte");
6   $VUE['articles'][]= array('id_article'=> 3, 'titre' => 'premier',
7   'texte' => "un texte");
8
```

```
9  require('pageListeArticle.php');
10 ?>
```

Enfin, ce mécanisme permet de choisir facilement plusieurs affichages différents en fonction du résultat d'une opération. Pour donner un exemple simple, supposons que nous voulions avertir l'utilisateur d'un problème éventuel de connexion à la base de données.

```
1  <?php
2
3  try {
4    $db= new PDO("mysql:host=localhost;dbname=publi", 'xxxx', 'xxxx');
5    $st = $db->query("SELECT * FROM article");
6    $VUE= array();
7
8    while ($obj= $st->fetchObject()) {
9      $VUE['articles'][]= array(
10                                'id_article' => htmlspecialchars($obj->id_article),
11                                'titre' =>  htmlspecialchars($obj->titre),
12                                'texte' => htmlspecialchars($obj->texte));
13   }
14   $ok= true;
15 } catch (PDOException $e) {
16   $ok= false;
17   $VUE['message']= htmlspecialchars("Désolé, erreur de connexion à la base");
18 }
19
20 if ($ok)
21   require('pageListeArticle.php');
22 else
23   require('erreur.php');
24 ?>
```

```
1  <html>
2  <head><title>Désolé nous avons un problème</title></head>
3  <body>
4
5  <h1> Problème </h1>
6  <p> <?php echo $VUE['message']; ?> </p>
7
8  </body>
9  </html>
```

2.4 Intégration du contrôle

La tâche du contrôle est
- de récupérer les données transmises par l'utilisateur ;
- de choisir l'opération à réaliser ;
- d'appeler l'affichage correct.

Dans les applications simples, le choix de l'opération se fait uniquement à partir de l'URL. On peut cependant rencontrer des cas plus complexes. Par exemple, supposons un forum en ligne. Si je clique sur le lien « répondre », je suis envoyé sur une page qui me permet d'éditer un commentaire.

Supposons maintenant que seuls les utilisateurs enregistrés puissent « répondre. » Il serait intéressant qu'au lieu de recevoir un refus, je sois envoyé sur une page me permettant de me connecter, et, si besoin, de créer un nouveau compte.

On voit donc que pour un maximum de souplesse, il est intéressant qu'à une page donnée puissent correspondre plusieurs traitements différents.

D'autre part, dans toute application un peu complexe, on va souhaiter qu'un certain nombre d'opérations soit *toujours* réalisées : tests pour vérifier si l'utilisateur est bien enregistré, vérification de la durée de vie de la session, etc...

Il existe plusieurs solutions pour répondre à ces problèmes.

La plus simple, une fois séparés les traitements et l'affichage, est d'adopter la séparation suivante :

- la logique (sous forme de fonctions à appeler ou de classes) est stockée dans des fichiers php qui ne sont pas directement accessibles par les clients, afin d'empêcher leur chargement non contrôlé.
- les traitements à réaliser dans tous les cas sont dans un fichier php à part, qui sera chargé par tous les contrôleurs (par `require_once`.
- les pages directement accessibles (le contrôle) en font le moins possible. Elles incluent le fichier précédent, traitent les paramètres, appellent les fonctions de la couche logique, remplissent en conséquence la vue, et passent les données à l'affichage.
- l'affichage, comme vu précédemment, se contente d'afficher les données contenu dans VUE (on peut bien évidemment raffiner un peu les choses, mais c'est le principe.)

Pour empêcher qu'un fichier php soit accessible par les clients, il y a plusieurs méthodes. La plus sûre est de le placer en dehors de l'arborescence accessible par le serveur HTTP.

On aura alors, par exemple, un fichier `toto.php`, accessible dans `htdocs`, et, ailleurs (par exemple dans `/home/moi/php`) un fichier `logique.php`. Ce dernier n'est pas accessible par le serveur, c'est à dire qu'un client ne peut l'atteindre en tapant une URL.

En revanche, il est possible à `toto.php` d'inclure `logique.php` :

```
1 require_once('/home/moi/php/logique.php');
```

Une autre technique est de placer les fichiers à protéger dans l'arborescence du serveur, mais d'en interdire l'accès direct en y plaçant un fichier `.htaccess` :

```
1 deny from all
```

2.5 Le mode POST et la redirection

Il vous est peut-être arrivé sur un site web de valider un formulaire, puis de recharger la page. Le navigateur vous demande alors s'il faut ou nous réexpédier les données.

Dans certains cas, ce peut être très gênant. Supposons par exemple qu'en validant deux fois le formulaire, on expédie deux fois la même commande.

Pour éviter ce genre de problèmes, on peut adopter la position suivante :

- une requête destinée à obtenir des informations devrait, dans la mesure du possible, être une requête GET. On peut ainsi la placer dans les marques-pages, l'indexer, etc...

– une requête POST est *a priori* destinée à modifier les données. Si on veut éviter qu'elle soit répétée accidentellement, il est conseillé de terminer le traitement de la requête par une redirection http.
Au lieu d'afficher un résultat directement, on indique au navigateur de charger une autre page. Si l'utilisateur clique sur le bouton *recharger*, c'est cette page-résultat qui sera chargée, et non le formulaire qui sera réexpédié.

Un petit exemple :

Supposons un blog qui permette d'enregistrer des commentaires.

La page `nouveauCommentaire.php` (un contrôleur) contient le code :

```
1  <?php
2    require_once('code/logiqueCommentaires');
3    $VUE['message']= "";
4    if (! isset($_POST['titre']) {
5      // Première visite : on affiche simplement le formulaire.
6       $VUE['titre']= "";
7       $VUE['texte']= "";
8       $suivant= "recharger";
9    } else if (isempty($_POST['titre'] || isempty($_POST['texte']))) {
10      $VUE['message']= htmlspecialchars("Il faut remplir titre et texte");
11      $VUE['titre']= $_POST['titre'];
12      $VUE['texte']= $_POST['texte'];
13      $suivant= "recharger";
14   } else {
15      // une fonction de logiqueCommentaire. Sauve le commentaire
16      // renvoie : -1 si échec
17      // l'identifiant du commentaire sinon.
18      $idComment= sauverCommentaire($_POST['titre'], $_POST['texte']);
19      if ($idComment == -1) {
20        // une autre fonction de la couche logique...
21        $VUE['message']= htmlspecialchars(getMessageErreurCommentaire());
22        $suivant= "erreur";
23      } else {
24        $suivant= "montrer";
25      }
26   }
27
28   switch ($suivant) {
29     case "recharger":
30       require_once("formulaireCommentaire.php"); break;
31     case "erreur":
32       require_once("erreur.php"); break;
33     case "montrer":
34       // Redirection vers une page qui affiche le nouveau commentaire (mode GET)
35       header("Location: afficherCommentaire.php?id=".$idComment);
36   }
37 ?>
```

Le fichier `formulaireCommentaire.php` :

```
1  <html><head><title></title><body>
2  <div class="message"> <?php echo $VUE['message'];?> </div>
3  <form action="nouveauCommentaire.php" method="POST">
4    <p> titre <input type="text" name="titre"
```

```
5                 value="<?php␣echo␣$VUE['titre'];?>"/></p>
6     <p> texte <input type="text" name="texte"
7                 value="<?php␣echo␣$VUE['texte'];?>"/></p>
8     <p> <input type="submit"/></p>
9   </form>
10  </body></html>
```

Le fichier `message.php` :

```
1  <html><head><title></title><body>
2  <div class="message"> <?php echo $VUE['message'];?> </div>
3  </body></html>
```

Incidemment, on voit comment ce système permet de gérer les formulaires avec le comportement habituel :

- à la première visite, on affiche le formulaire ;
- ensuite, on vérifie que les données sont correctes. Si ce n'est pas le cas, on affiche à nouveau le formulaire, en conservant ce que l'utilisateur avait saisi ;
- enfin, si tout est correct, l'action demandée est exécutée, et nous sommes redirigés vers une page qui affiche son résultat.

2.6 L'architecture Modèle/Vue/Contrôleur, récapitulation

Développée d'abord pour les interfaces graphiques, en particulier pour le langage objet smalltalk, l'architecture MVC comporte de nombreuses variantes.

Dans tous les cas, on sépare les données (le modèle) de leur affichage (la vue). Dans la version de MVC utilisée pour les applications « non web », le principe est que :

- le contrôle reçoit un événement (par exemple un clic souris) ; il modifie le modèle en conséquence.
- le modèle *prévient **ses** vues* qu'il a été modifié ;
- les vues se redessinent en fonction du contenu actuel du modèle.

Cette architecture, qui permet entre autres de présenter plusieurs visions simultanées d'un même modèle, n'est pas adaptée aux applications web.

Le MVC « web » a donc la structure :

- le contrôle reçoit une requête ;
- il la fait traiter par la vue, qui s'en trouve modifiée ;
- le contrôle extrait les données pertinentes, et les passe à la vue pour affichage.

2.7 La persistance

On va généralement isoler l'accès aux bases de données dans une couche à part. Il s'agit d'éviter, d'une part que la complexité des opérations d'écriture, de recherche et de sauvegarde ne se mèle à celle du modèle.

Typiquement, on a intérêt à « faire simple ». Soit une information sauvegardée dans la base, par exemple un commentaire.

Le commentaire a un identifiant, un auteur (identifié par un numéro), un titre et un texte.

On va donc créer dans le modèle une représentation du commentaire, sous forme de classe (ou de tableau, mais pourquoi se priver des classes ?)

```
1  class Commentaire {
2    public $id_commentaire,
```

```
3     public $id_auteur,
4     public $titre,
5     public $texte;
6   }
```

En orienté objet, on créerait ensuite un `DAO` (data access object) responsable de la sauvegarde et de la récupération des commentaires dans la base de données.

On peut aussi se contenter d'écrire les fonctions nécessaires, sans les placer dans une classe.

Les fonctions en question fournissent typiquement les fonctionnalités décrites par l'abbréviation `CRUD` :

Create créer, à partir d'un objet Commentaire, une nouvelle entrée dans la base de données ; la fonction

```
1  function daoCreerCommentaire($commentaire) {...}
```

peut éventuellement replir certaines variables d'instance (par exemple `$id_commentaire`) ;

Retrieve on fournit généralement une ou plusieurs fonction qui permettent d'aller récupérer des données sur un commentaire dans la base. En gros : une fonction par requête SQL. Ces fonctions retournent, soit un Commentaire, soit un tableau de Commentaires ;

Update met à jour une entrée déjà existante de la base (modification d'un commentaire déjà existant) ;

Delete destruction d'un commentaire existant.

On pourrait donc avoir le fichier suivant :

```
1
2  function daoCreerCommmentaire($commentaire) {....}
3  function daoUpdateCommmentaire($commentaire) {....}
4  function daoDeleteCommmentaire($id_commentaire) {....}
5
6  // retourne le commentaire d'id $id_commentaire
7  function daoTrouverCommmentaire($id_commentaire) {....}
8
9  // retourne tous les commentaires
10 function daoTrouverCommmentaires() {....}
11
12 // retourne tous les commentaires d'un certain auteur
13 function daoTrouverCommmentairesParAuteur($id_auteur) {....}
```

2.8 Un exemple

Nous fournissons en archive zip un exemple. Il s'agit d'un petit site de blog, avec les contraintes suivantes :

- Il y a un administrateur, qui seul peut ajouter de nouveaux utilisateurs enregistrés ;
- les utilisateurs enregistrés peuvent écrire des articles ;
- les articles peuvent être privés ou publics. Un article privé n'est visible que des utilisateurs enregistrés ;
- quand on liste les articles, on ne voit que ceux que l'on a le droit de lire.

L'architecture choisie utilise un « contrôleur en frontal », c'est à dire que toutes les requêtes passent par la même URL. Cela permet de garantir qu'un certain nombre de traitements seront toujours effectués.

Ce poly intégrera prochainement une description plus détaillée du logiciel d'exemple. En attendant, quelques pistes :

1. Pour installer le logiciel, créez la base de données, le sql est dans `base.sql`. Créez manuellement l'administrateur (l'utilisateur d'identifiant 1.
2. allez modifier le fichier `code/Codes.php`. Il contient le login et le mot de passe de la base.
3. visitez le site... ça devrait fonctionner.
4. ensuite, l'architecture du site. Chaque URL a la forme `index.php?action=.....` La valeur d'`action` détermine ce qui sera fait, plus exactement quel contrôleur sera utilisé.

   Pour ajouter une nouvelle action, il faut donc créer le contrôleur correspondant, et modifier `index.php` en conséquence.
5. Le contrôleur a pour tâche de récupérer les données, de les traiter, et de remplir le tableau `$vue`. Contrairement à la première partie de ce cours, nous avons laissé aux pages d'affichage le soin de protéger les variables de vue par `htmlspecialchars` (il n'y a pas de bonne raison pour cela. Juste que nous avons écrit le logiciel avant le poly).

   Les contrôleurs héritent de la classe `Controle`. ils peuvent remplir une des deux variables d'instance `page` ou `redirect`. Si `redirect` n'est pas `null`, l'affichage sera réalisé par redirection. Sinon, on visualisera la page `$this->page`, qui doit se trouver dans le dossier `ui`. Ce dernier contient toutes les pages html.
6. `Controle` comporte aussi quelques méthodes dédiées à la sécurité, qui permettent de tester que l'utilisateur a bien le droit d'effectuer l'action en cours.
7. Pour un contrôle typique, nous avons séparé la partie « intéraction web » à proprement parler des actions plus liées à la logique du programme, en deux classes. Ainsi les contrôles sur les articles (`CreerArticle, LireControle, ListerControle` utilisent-ils la classe `ArticleLogique`, qui elle-même fait appel à `ArticleDAO`.
8. Pour aborder ce programme, nous suggérons de commencer par comprendre le fonctionnement de l'action de déconnexion, puis de connexion, et ensuite de passer, par exemple, à l'action `lire`, qui reste relativement simple.

Le cadre applicatif est probablement trop complexe par rapport à ce que nous désirons faire. Mais :

- il est extensible facilement : une fois compris le principe, le mode de programmation est très répétitif ;
- il gère beaucoup de points « annexes », tels que la vérification de la connexion, un affichage systématique des messages d'erreurs, etc.

Des améliorations sont possibles :

- nous pourrions, comme dans les exemples du début du cours, placer dans `$vue` uniquement des données protégées par `htmlspecialchars` ;
- comme nous manipulons aussi des liens, un tableau de liens similaire au tableau des vues pourrait être utile ;
- le menu est ici donné statiquement. Il serait simple (et c'est un exercice laissé au lecteur intéressé) de n'afficher que les actions réalisables par l'utilisateur ;
- nous nous sommes peut-être un peu trop inspirés des architectures java... certains objets ne sont sans doute pas nécessaires en PHP.

3 Dernier conseils

Faut-il impérativement éclater votre application en une trentaine de fichiers comme nous l'avons fait ?

En fait, cela dépend de la complexité et de l'évolution du programme. Il nous semble que la séparation traitements/affichage est souhaitable dans pratiquement tous les cas, sauf si votre programme est très court, et qu'il n'est pas destiné à évoluer.

Dès qu'on commence à avoir plusieurs pages, entre lesquelles on veut assurer une homogénéïté de présentation, la séparation au moins entre modèles et vues est nécessaire. On aura aussi intérêt à bâtir la vue à partir de composants distincts comme dans notre exemple.

Isoler le code SQL est aussi une bonne idée. Si la base change, on saura immédiatement où opérer des modifications.

Assez rapidement, on s'aperçoit qu'un certain nombre d'opérations doivent toujours être réalisées (par exemple récupérer les données de l'utilisateur). Plus les pages seront complexes, plus une architecture MVC sera utile.

Dans la réalité, notre petit programme est un embryon de *framework*. Il en existe de beaucoup plus complets et plus sûrs. Pour de gros développements, ils apportent une infrastructure solide, au prix d'une certaine complexité d'accès. Ils sont trop nombreux pour tous les citer : Zend framework, Symphony, CakePHP par exemple. Ce dernier nous semble être une bonne solution pour démarrer, car il reste raisonnablement simple.