Définition du besoin et scénario d'usage du gestionnaire d'images

Sommaire

Les repères de lecture


Retour au chapitre


Définition


Objectif(s)


Espace Élèves


Vidéo / Audio

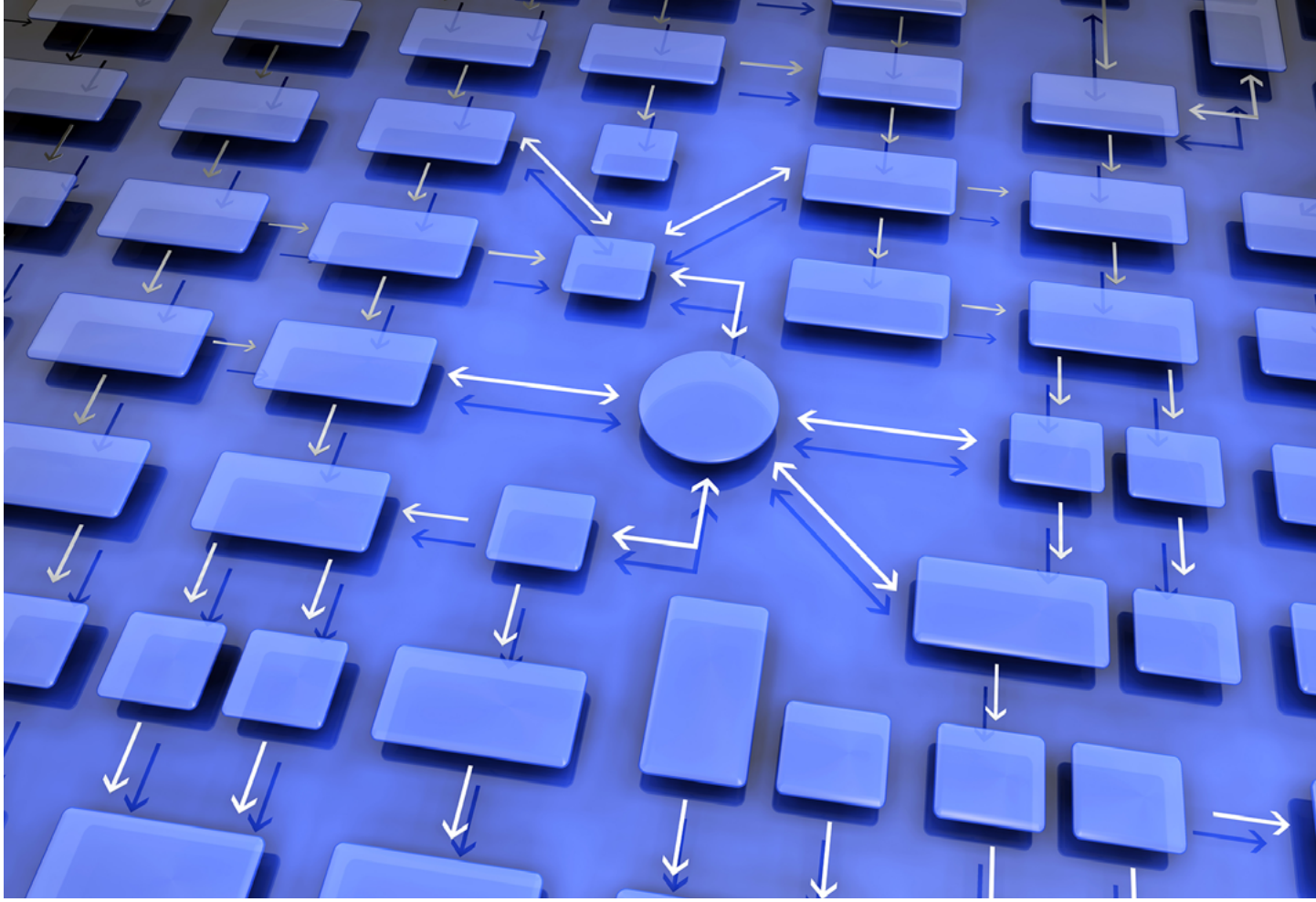

Point important / À retenir


Remarque(s)


Pour aller plus loin


Normes et lois


Quiz



Introduction

Nous allons intentionnellement limiter les fonctionnalités à mettre en œuvre et nous nous contenterons pour le moment des besoins *front-office* et *back-office* suivants :

— **front-office** : afficher des images au format vignette (petite dimension). Les vignettes sont cliquables et affichent l'image dans le format initial plus grand ;

— **back-office** : associer un titre et une description aux images.

Le scénario d'usage est le suivant : l'administrateur du site fournit un titre et une description.

L'internaute consulte les images et clique sur les images de son choix pour les afficher en grand format.

Dans ce contexte et afin de démarrer, nous préparons donc un lot de deux images que nous pourrons utiliser pour notre développement. Ces images au format JPG, doivent être d'une dimension relativement petite pour des raisons de mise en page (si possible moins de 400 pixels de large et/ou de haut). Nous les nommerons **image1.jpg** et **image2.jpg**.

Sur notre serveur local, nous créons un répertoire *projet_image* avec un sous-répertoire « image » et nous plaçons nos deux images dans celui-ci.

image1.jpg

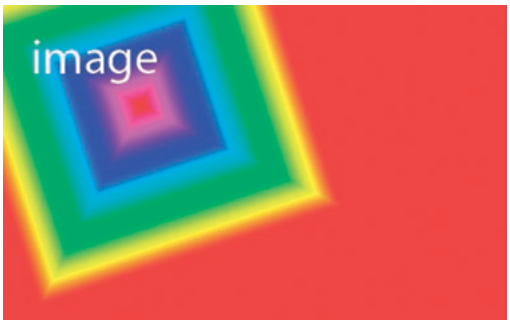


image2.jpg

I. Conception du projet

Nous allons donc avoir besoin :
– d'une base de données dans laquelle nous enregistrerons les noms et les descriptions de chaque image ;
– de lister toutes les images contenues dans le répertoire *image*.

À cette fin, nous créerons une classe possédant une méthode dédiée à cette tâche.

Nous préparons donc la base de données *projet_image* contenant une table intitulée *images*. Cette table est structurée avec les champs suivants :

```
-- Structure de la table `image`
CREATE TABLE IF NOT EXISTS `image` (
 `id` int(11) NOT NULL AUTO_INCREMENT,
 `title` varchar(255) NOT NULL,
 `description` text NOT NULL,
 `filename` varchar(200) NOT NULL,
 PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 AUTO_INCREMENT=1 ;
```

Fig. 1

Nous élaborons ensuite la classe que nous appelons simplement *Image* et qui sera développée dans le fichier **class/Image.php**. Pour mémoire, rappelons qu'une majuscule est donnée à ce nom de fichier afin qu'il soit nommé de la même manière que la classe *Image*, cette concordance des noms classes/fichiers étant une bonne pratique du développement qui nous permet de nous retrouver rapidement et aisément parmi les classes et les fichiers.

II. Principe de mise en œuvre : *opendir* et *readdir*

Pour lister les images, le principe de mise en œuvre s'effectue en deux temps principaux : ouverture du dossier puis identification des fichiers contenus dans le dossier (dossier et répertoire sont des termes synonymes).

A. Ouverture du répertoire avec la fonction *opendir*

Le **pointeur** (*pointer* en anglais) est à considérer comme une sorte de variable permettant à PHP d'accéder, par un mécanisme interne qui lui est propre, aux informations disponibles pour ce répertoire ouvert sans avoir à stocker l'ensemble de ces informations. Celles-ci peuvent en effet être nombreuses et leur poids total pourrait engendrer une lenteur de traitement lors de leurs manipulations par PHP.

En l'occurrence et comme stipulé dans le manuel, *opendir* retourne la ressource de dossier en cas de succès ou *FALSE* en cas d'échec.

Il est recommandé dans le cadre de l'étude de PHP, de tester avant de les intégrer les fonctions que nous rencontrons, afin de bien comprendre leur fonctionnement. Un répertoire dédié aux tests dans *localhost* fait parfaitement l'affaire pour regrouper l'ensemble des tests que nous sommes amenés à conduire.

La définition donnée est qu'*opendir* « **ouvre un dossier, et récupère un pointeur dessus** ».

Page du manuel PHP : http://www.php.net/manual/fr/function.opendir.php

Testons donc la fonction *opendir* sur le répertoire *projet_image* (nous passons en argument le chemin complet du dossier) et affectons le résultat à la variable *$handle* (gérer, manipuler, traiter) qui est le nom traditionnel donné aux variables contenant des informations de répertoires.

```
$handle = opendir('C:\wamp\www\projet_image'); // chemin complet du
//dossier
var_dump ($handle);
```

Fig. 2

La fonction *opendir* a ouvert avec succès le dossier et le *var_dump* affiche ceci :

```
resource(3, stream)
```

Fig. 3

Comme nous le constatons, aucune information précise n'est fournie et si nous ne savons pas en lisant le résultat de quel dossier il s'agit, PHP en revanche sait très bien grâce au mécanisme du pointeur qu'il s'agit du répertoire *projet_image*. Ce résultat est également nommé une **ressource**, c'est-à-dire une variable spéciale exploitable par d'autres fonctions PHP.

Si *opendir* ne peut pas ouvrir un dossier, au cas où par exemple le chemin d'accès ne soit pas correct, le résultat retourné serait alors le booléen *FALSE* (précédé naturellement des erreurs de type *warning* : le fichier spécifié est introuvable).

```
boolean false
```

Fig. 4

B. Identification des fichiers contenus avec la fonction *readdir*

La fonction *readdir* permet d'identifier les entrées d'un répertoire ouvert avec *opendir*.

Page du manuel PHP : http://www.php.net/manual/fr/function.readdir.php

Une entrée est un fichier au sens où l'entend UNIX, c'est-à-dire un fichier ou un dossier. De plus, les valeurs point « . » et double point « .. » qui sous UNIX définissent une arborescence de fichiers sont également des entrées.

Continuons le test précédent :

```
$handle = opendir('C:\wamp\www\projet_image');
$entry = readdir($handle);
var_dump ($entry);
```

Fig. 5

Le *var_dump* affiche le point :

```
string '.' (length=1)
```

Fig. 6

Si nous écrivons une deuxième fois *readdir* comme ceci :

```
$handle = opendir('C:\wamp\www\projet_image');
$entry = readdir($handle);
var_dump ($entry);
$entry = readdir($handle);
var_dump ($entry);
```

Fig. 7

Nous obtenons :

```
string '.' (length=1)
string '..' (length=2)
```

Fig. 8

En l'écrivant une troisième fois nous obtenons :

```
string '.' (length=1)
string '..' (length=2)
string 'admin.php' (length=9)
```

Fig. 9

Nous constatons bien que comme le dit le manuel PHP que *readdir* retourne **le nom de la prochaine entrée** à chaque fois que cette fonction est appelée.

Une utilisation de la fonction *readdir* avec une boucle *while* semble alors être une idée judicieuse pour afficher dynamiquement l'ensemble des fichiers contenus, en excluant les points et double points. C'est d'ailleurs la méthode préconisée par le manuel PHP dont nous allons étudier l'exemple *#1* pour l'adapter et l'intégrer dans une méthode de notre classe *Image*.

Comprenons bien la syntaxe de l'exemple défini dans le manuel PHP comme étant la façon correcte de parcourir un dossier :

```
while (false !== ($entry = readdir($handle)))
```

Fig. 10

La boucle *while* signifie ici : tant qu'il est possible de lire les fichiers du répertoire avec *readdir* et d'affecter ce résultat à une variable *$entry*. L'utilisation de *false* est pratique pour caractériser le résultat d'une fonction car, nous l'avons vu, dans un contexte booléen c'est-à-dire en logique, *false* est faux et *true* est vrai, et nous pouvons alors définir si le résultat d'une fonction est un succès (*true*) ou un échec (*false*).

Donc si lire un fichier est possible, alors l'affectation à *$handle* retourne vrai. D'un point de vue logique, la formulation « tant que c'est vrai » revient à énoncer « tant que ce n'est pas faux » ce qui justifie la syntaxe « *false !==* » qui signifie que la proposition *($entry = readdir($handle))* n'est pas fausse, et que donc la boucle peut continuer.

Rappelons que :
- si une variable est vraie (ou fausse) nous écrivons *$var === true* (ou *$var === false*), avec trois fois le signe égal ;
- nous pouvons écrire indifféremment *TRUE* ou *true* et *FALSE* ou *false*.

Nous intégrons donc maintenant à notre classe *Image* la méthode que nous appellerons *getImages*.

```
<?php
class Image
{
 public function __construct()
 {
// le constructeur est vide pour ce projet
 }
/* methode retournant les fichiers présents dans le repertoire où
nous   avons placé nos images et que nous définissons au moyen de la
variable $image_dir
*/
public function getImages($image_dir)
{
// nous ouvrons le dossier $image_dir avec opendir
// et affectons le résultat à la variable $handle
if ($handle = opendir($image_dir))
{
//
while (false !== ($entry = readdir($handle)))
{
/* la variable $entry ne pourra pas se voir affecté les . et les ..
*/
if ($entry != "." && $entry != "..")
{
/* nous affectons le resultat dans un array */
$images[] = $entry;
}
}
}
closedir($handle); // nous fermons le repertoire avec closedir
return $images ; // nous retournons le tableau de données
}
}
```

Fig. 11

Script d'affichage, fichier **contenu.php** :

```
<?php
require('class/Image.php');
$image = new Image();
// définition du chemin et de l'URL du répertoire image
// chemin (path) du repertoire images
$image_dir_path = $_SERVER['DOCUMENT_ROOT'] . 'projet_image/
images/';
$image_dir_url = 'http://'.$_SERVER['HTTP_HOST'] . '/projet_image/
images/';
// affectation dans la variable $images du résultat de la méthode
getImages
$images = $image->getImages($image_dir_path);
// affichage;
?>
<?php foreach ($images as $image): ?>
<li><img src="<?php echo $image_dir_url. $image ?>" /></li>
<?php endforeach ?>
```

Fig. 12

Pour mémoire la variable *$_SERVER* déjà rencontrée fournit de nombreuses informations pratiques sur l'environnement utilisé : http://php.net/manual/fr/reserved.variables.server.php. Dans le cadre du projet, il est pertinent de définir de façon permanente, avec des constantes, certaines données, telles que le nom du répertoire image par exemple. Ces constantes seront judicieusement définies dans un **fichier de configuration** usuellement nommé **config.php** qui sera placé à la racine du site et inclus dans les principaux fichiers.

Nous définirons pour le moment les constantes pour les informations suivantes :

- nom du dossier principal : *WEB_DIR_NAME* ;
- nom du dossier contenant les images et qui est donc un sous-dossier du répertoire principal : *IMAGE _DIR_NAME* ;
- le chemin (*path*) complet du répertoire image : *IMAGE _DIR_PATH* ;
- l'URL du répertoire image : *IMAGE _DIR_URL*.

config.php

```
<?php
// fichier de configuration
define ('WEB_DIR_NAME', 'projet_image');
define ('IMAGE_DIR_NAME', 'images');
define ('IMAGE_DIR_PATH', $_SERVER['DOCUMENT_ROOT'] . '/' . WEB_DIR_
NAME .'/'. IMAGE_DIR_NAME .'/');
define ('IMAGE_DIR_URL', 'http://' . $_SERVER['HTTP_HOST'] . '/'.
WEB_DIR_NAME .'/'. IMAGE_DIR_NAME .'/');
```

Fig. 13

Nous avons donc profité de ce fichier de configuration pour mettre en constantes le nom du dossier web local *projet_images*. Cette utilisation des constantes est un cas typique de l'intérêt des constantes : le nom du dossier local ne changera pas tout au long du projet, et en incluant le fichier *config.php* dans toutes les pages du site, les valeurs des constantes sont directement utilisables dans tous nos scripts. Ce fichier **config.php** sera mis à jour régulièrement tout au long du projet au fur et à mesure de l'évolution de nos besoins.

Le code de la page **index.php** est de fait devenu très simple :

index.php

```
<?php
require('config.php');
require('class/Image.php');
require('contenu.php');
```

Fig. 14

Notons qu'il n'est pas nécessaire de fermer le bloc PHP avec ?> puisqu'il s'agit d'un fichier contenant uniquement du code PHP. Du reste, cela constitue une bonne pratique en permettant notamment de s'assurer qu'aucun caractère vide (un espace par exemple) ne s'insinue dans nos scripts et n'occasionne des erreurs.

Le code de la page contenu devient donc également plus simple grâce aux constantes nouvellement créées :

```
<?php
$image = new Image();
$images = $image->getImages(IMAGE_DIR_PATH);
?>
<ul>
<?php foreach ($images as $image): ?>
<li><img src="<?php echo IMAGE_DIR_URL . $image ?>" /></li>
<?php endforeach ?>
</ul>
```

Fig. 15

III. La fonction *get_defined_constants*

Dans une perspective de suivi et de débuggage (correction) et/ou de maintenance du code, à l'instar des fonctions déjà maitrisées comme *var_dump*, *print_r* ou *get_object_vars*, la fonction **get_defined_constants** liste toutes les constantes utilisées. Cette fonction très pratique retourne aussi bien les constantes natives du cœur de PHP (ainsi que celles fournies par les extensions PHP) que les constantes utilisateurs, c'est-à-dire celles créées par le développeur.

Si l'argument est *TRUE* la fonction retourne un tableau associatif dont les clefs sont les catégories d'appartenance : *core*, c'est-à-dire cœur de PHP, *extensions* ou *user*.

Code pour lister toutes les constantes :

```
print_r(get_defined_constants(true));
```

Fig. 16

Code pour lister les constantes *user* :

```
$constants = get_defined_constants(true);
print_r($constants['user']);
```

Fig. 17

Ce code retourne :

```
Array
(
  [WEB_DIR_NAME] => projet_image
  [IMAGE_DIR_NAME] => images
  [IMAGE_DIR_PATH] => C:/wamp/www/projet_image/images/
  [IMAGE_DIR_URL] => http://localhost/projet_image/images/
)
```

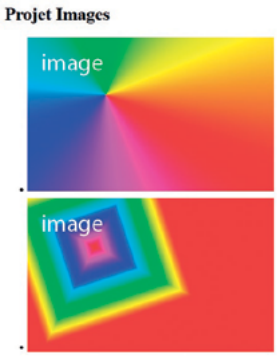
Fig. 18

Nous pouvons également afficher le nom du site *'Projet Images'* et l'enregistrer dans une constante *WEB_TITLE* pour être intégré dans les éléments html <*title*> et <*h1*>.

Nous constatons que l'affichage des images pour l'internaute et pour l'administrateur sont semblables à la différence notable que l'affichage pour l'administrateur doit comporter une interface permettant de saisir puis valider un titre et une description afin de les enregistrer dans la base de données.

Le formulaire étant le moyen de saisir et valider des informations, nous allons ajouter un formulaire pour chaque image affichée.

Page contenu (front-office)



Page *admin*, avec formulaire (back-office)

La page *admin* est similaire à la page contenu à la différence près que nous ajoutons le formulaire pour chaque image.

admin.php

```
<?php
require('config.php');
require('class/Image.php');
$image = new Image();
$images = $image->getImages(IMAGE_DIR_PATH);
?>
<h1><?php echo WEB_TITLE ?></h1>
<ul>
<?php foreach ($images as $image): ?>
<li><img src="<?php echo IMAGE_DIR_URL . $image ?>" />
<form method="post" action="process_image.php">
<p>Titre : <input type="text" name="title" /></p>
<input type="hidden" name="filename" value="<?php echo $image ?>" />
<p>Description<br> <textarea name="descr" cols="50" rows="5"></
textarea></p>
<p><input type="submit" name="formImageSubmit" value="validez" /></
p>
</form>
</li>
<?php endforeach ?>
</ul>
```

WEB_T

Fig. 19

Nous incluons bien le fichier **Image.php** qui contient la classe *Image*.

Nous indiquons, dans un premier temps, que la cible du formulaire (action) est le fichier **process_image.php** que nous créerons ensuite et dont le nom de fichier est préfixé avec *process_* pour indiquer qu'un processus va y être élaboré, en l'occurrence le contrôle des valeurs soumises par le formulaire et le traitement de ces valeurs.

```
<form method="post" action="process_image.php">
```

Fig. 20

Après avoir récupéré les valeurs postées par le formulaire, le script de **process_image.php** les vérifie puis s'occupe de leur enregistrement dans la base de données. Nous allons dans un premier temps effectuer un test sans nous soucier pour le moment d'afficher les messages d'erreur ou de confirmation que tout espace d'administration se doit d'afficher suite aux actions des utilisateurs.

En premier lieu, il est toujours intéressant de vérifier par un *print_r($_POST)* ou un *var_dump($_POST)* que les valeurs sont bien postées et ce qu'elles contiennent :

```
Array
(
    [title] => test titre
    [filename] => image1.jpg
    [descr] => test description
    [formImageSubmit] => validez
)
```

Fig. 21

Les valeurs sont bien récupérées par le fichier **process_image.php** :
– **title** : saisie du champ correspondant au titre ;
– **filename** : nom de fichier de l'image ;
– **descr** : saisie du *textarea* correspondant à la description ;
– **formImageSubmit** : la valeur du bouton *Submit*.

Nous pouvons donc continuer les étapes suivantes :
– **étape 1** : vérification classique des valeurs (sont-elles postées ? sont-elles vides ?) ;
– **étape 2** : enregistrement dans la base de données.

1. process_image.php – étape 1

```
require('config.php');
// les valeurs sont elles postées ?
if(!isset($_POST['formImageSubmit']))
{
$error_msg = 'Aucune donnée n\'est fournie.
<a href="' . WEB_DIR_URL .'admin.php">retour</a>';
}
if(isset($_POST['formImageSubmit'])) // cas où les valeurs sont
//postées
{
  // si une des valeurs est vide
  if( (empty($_POST['title'])) OR (empty($_POST['descr']))
OR (empty($_POST['filename'])) )
 {
$error_msg = 'une des informations est manquante.
<a href="' . WEB_DIR_URL .'admin.php">retour</a>';
 }
else
 {
// enregistrement dans la base - reste à faire
 }
}
```

Fig. 22

À ce stade, nous gérons bien les cas où les valeurs sont postées ou vides. Il ne reste qu'à effectuer l'enregistrement si les valeurs sont correctes.

Pour ce faire, nous allons créer une nouvelle méthode dédiée à cette action dans la classe *Image*. Nous nommons simplement cette méthode *insertImage* (Insérer une image).

Les trois arguments utiles correspondront aux informations que nous souhaitons enregistrer dans la base et qui sont naturellement celles que nous saisissons via le formulaire :
– le titre ;
– la description ;
– le nom de fichier.

La méthode s'écrira donc avec les trois arguments : *insertImage($title, $descr, $filename)*.

La classe *Image* permettra donc d'effectuer deux opérations :
– afficher les images contenus dans le répertoire avec la méthode : *getImages($image_dir)* ;
– enregistrer dans la base les informations postées depuis l'administration : *insertImage($title, $descr, $filename*).

Le code de la méthode *insertImage* est le suivant :

```
public function insertImage($title, $descr, $filename)
{

$mysqli = new mysqli('localhost', 'root', '', 'projet_image');

$mysqli->set_charset("utf8"); // encodage utf8


// Vérification de la connexion à la base

if ($mysqli->connect_errno) {

echo 'Echec de la connexion ' .$mysqli->connect_error ;

exit();

}



if (!$mysqli->query('INSERT INTO image (title, description,
filename) VALUES ("'. $title .'", "'. $descr .'", "'. $filename
.'")'))

{

echo 'Une erreur est survenue lors de l\'insertion des données dans
la base. Message d\'erreur : ' . $mysqli->error;

return false;

}

else

return true;


$mysqli->close();
}
```

Fig. 23

Nous pourrons si besoin nous reporter au cours précédent pour nous remémorer la syntaxe et les principes de mise en œuvre des requêtes MySQL en PHP.

Nous pouvons toutefois noter que la syntaxe connue *$mysqli = new mysqli* contient le mot clef *new* ce qui est typique d'une syntaxe POO. Le fonctionnement de *mysqli* repose donc sur une gestion objet.

La classe Image contient désormais la nouvelle méthode *insertImage* et nous allons donc pouvoir l'utiliser dans le fichier **process_image.php** qui a justement pour rôle d'enregistrer les images dans la base de données.

Nous finalisons donc le fichier **process_image.php** tout d'abord en incluant le fichier comportant la classe *Image* (**Image.php**) puis nous intégrons la méthode *insertImage*.

2. process_image.php – étape 2

```
require('config.php');
require('Image.php');
if(!isset($_POST['formImageSubmit']))
{
     $error_msg = 'Aucune donnée n\'est fournie. <a href="' .
     WEB_DIR_URL .'admin.php">retour</a>';
}

if(isset($_POST['formImageSubmit']))
{
  if( (empty($_POST['title'])) OR (empty($_POST['descr']))
     OR   ( empty($_POST['filename'])) )
     {
          $error_msg = 'une des informations est manquante. <a href="' .
          WEB_DIR_URL .'admin.php">retour</a>';
     }
     else
     {
          $title = trim ($_POST['title']);
          $descr = trim ($_POST['descr']);
          $filename   = trim ($_POST['filename']);

          $image = new Image();
          $insertImage = $image->insertImage($title, $descr, $filename) ;

          if(true === $insertImage)
          {
            header('location:' . WEB_DIR_URL .'admin.php?insertImage=ok');
          }
          else
          {
               $error_msg = '<br><a href="' .
               WEB_DIR_URL .'admin.php">retour</a>';
          }
     }
}
```

Fig. 24

La fonction *trim* est utile car elle permet d'enlever les espaces blancs au début et à la fin des chaîne de caractère. Il est conseillé de l'utiliser systématiquement lors du traitement des valeurs soumises par un utilisateur.

Page sur la fonction *trim* : http://fr2.php.net/manual/fr/function.trim.php

Nous l'utilisons ici dans *$title = trim ($_POST['title']);* lorsque nous affectons la valeur du champ postée à la variable *$title*.

$image = new Image() nous permet classiquement de créer une instance de la classe *Image* et d'accéder ainsi aux méthodes de la classe.

Puis, nous utilisons la fonction *insertImage* avec la syntaxe vue précédemment (opérateur flèche). Nous affectons ensuite le résultat de la fonction à une variable *$insertImage* que nous avons créé spécialement pour gérer les résultats.

```
$insertImage = $image->insertImage($title, $descr, $filename);
```

Fig. 25

Page du manuel à propos de la fonction *header* : http://fr2.php.net/manual/fr/function.header.php

Si l'enregistrement dans la base est effectué, la méthode retourne *TRUE* et donc la variable *$insertImage* aura pour valeur *TRUE*. Si elle l'est, alors nous revenons sur la page *admin* au moyen de la fonction *header()* et de son argument location. Sinon nous créons un message d'erreur spécifique.

À ce stade, et puisque nous avons décidé dans un premier temps d'effectuer un simple test, nous ne gérons pas encore l'affichage des messages d'erreurs et nous testons notre code.

Nous vérifions donc les points suivants :
- le formulaire permet bien de saisir un titre et une description pour chacune des images ;
- la validation du formulaire poste les trois valeurs souhaitées (titre, description, nom de fichier de l'image) ;
- nous revenons après validation sur la page d'administration ;
- les données (valeurs) postées sont bien enregistrées dans la base de données pour les deux images. Nous utilisons *phpMyAdmin* pour cette vérification.

Fig. 26

Une fois ces points vérifiés, nous pouvons envisager de restituer les messages d'erreur et de confirmation qui participent de l'ergonomie indispensable à une interface d'administration.

Nous constatons par ailleurs que l'utilisation de la fonction *insertImage* permet d'insérer à chaque validation du formulaire des valeurs différentes pour la même image. Il est donc naturel que nous cherchions soit à insérer un contenu si l'image n'en contient pas, soit à modifier ce contenu si l'image en contient déjà un, au lieu de nous retrouver avec plusieurs informations inutiles. Nous allons donc vérifier pour chaque image si elle contient du contenu ou non. En fonction de ce que nous constaterons, nous utiliserons soit la méthode *insertImage* soit une méthode *updateImage* (mise à jour) que nous devons alors créer.

Par ailleurs, il semble souhaitable d'obtenir la confirmation que l'enregistrement a bien été effectué, par exemple en affichant en dessous des images les informations titre et description que nous venons de saisir. Lors de l'affichage des images nous serons donc amenés à utiliser les informations issues de la base de données, ce qui nous permettra donc de traiter le besoin évoqué ci-dessous c'est-à-dire de vérifier dans la base de données l'existence ou l'absence de contenu pour l'image.

Enfin il serait très utile d'ajouter directement des nouvelles images au moyen d'une interface de chargement (*upload*).