

Upload, chargement et sécurisation d'un fichier via un formulaire

Sommaire

I. Upload, chargement d'un fichier via un formulaire	3
A. La variable <code>\$_FILES</code>	3
B. Mise en oeuvre de l' <i>upload</i>	5
C. Intégration du script dans l'application	8
II. Sécurisation	13
A. Empêcher le chargement de certains types de fichiers	13
B. Sécurité : limiter le poids des fichiers à uploader	14

Crédits des illustrations : © Fotolia

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

I. Upload, chargement d'un fichier via un formulaire

A. La variable \$_FILES

Comme demandé, nous souhaitons maintenant pouvoir charger des images sur le site.

Comme nous le savons, un formulaire de chargement ou d'*upload* en HTML 5 possède la syntaxe suivante :

```
<form id="uploadForm" action="" method="post" enctype="multipart/form-data">
<p>Ajoutez des images</p>
<input type="file" value="" name="upload[]" multiple="multiple">
<input id="uploadFormSubmit" name="uploadFormSubmit" type="submit">
</form>
```

Fig. 1

Il s'agit d'un formulaire classique d'*upload* avec les boutons « Parcourir » et « Envoyer ».

Le bouton « Parcourir » permet de sélectionner les fichiers et le bouton « Envoyer » de transmettre les fichiers au serveur. C'est l'attribut *enctype* avec la valeur *multipart/form-data* qui permet ce fonctionnement.

Ajoutez des images

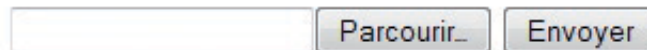


Fig. 2

En HTML 5, l'attribut *multiple* du formulaire permettra l'*upload* de masse, c'est-à-dire que nous pourrions sélectionner plusieurs images en même temps, comme le montre la capture ci-dessous, le rectangle bleu étant ici la zone de sélection standard de Windows.

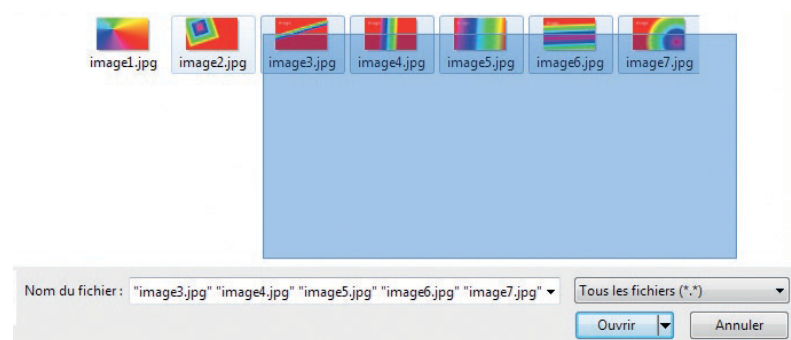


Fig. 3 Sélection multiple sous Windows

Du fait que plusieurs fichiers peuvent être transmis, nous définirons la valeur du champ *file* qui va contenir les fichiers avec des crochets, de manière à définir un tableau de données lors du traitement.

Nous créons donc un fichier **upload.php** pour y écrire le code du formulaire. Nous allons dans un premier temps en tester le fonctionnement puis l'intégrer pas à pas dans notre application de façon à ce que les fichiers uploadés soient placés dans le répertoire images et que les informations puissent être enregistrées dans la base de données.

Testons le formulaire en chargeant deux fichiers. Les fichiers **sont postés** mais **seront transmis par la variable \$_FILES** et non pas par la variable \$_POST. La variable \$_FILES est un tableau associatif des **valeurs téléchargées**.

Page du manuel PHP : <http://www.php.net/manual/fr/reserved.variables.files.php>

Un `print_r($_FILES)` nous permettra donc d'étudier ce qui est transmis par le formulaire :

```
Array
(
    [upload] => Array
        (
            [name] => Array
                (
                    [0] => image4.jpg
                    [1] => image5.jpg
                )
            [type] => Array
                (
                    [0] => image/jpeg
                    [1] => image/jpeg
                )
            [tmp_name] => Array
                (
                    [0] => C:\wamp\tmp\php744F.tmp
                    [1] => C:\wamp\tmp\php746F.tmp
                )
            [error] => Array
                (
                    [0] => 0
                    [1] => 0
                )
            [size] => Array
                (
                    [0] => 14597
                    [1] => 18336
                )
        )
)
```

Fig.4

Ce tableau associatif a comme première clef *upload* : il s'agit du *name* donné au champ du formulaire. Cette clef a pour valeur le tableau contenant les valeurs téléchargées par le formulaire.

Le tableau des valeurs possède les clefs suivantes :

- **name** : le nom de fichier initial des fichiers postés (exemple : **image4.jpg**) ;
- **type** : le *type*. Ici les fichiers sont des images au format JPG. C'est grâce à cette valeur que nous pourrions si besoin interdire l'*upload* de certains types de fichiers ;
- **tmp_name** : c'est le chemin temporaire donné par le serveur aux fichiers postés ;

Le principe du téléchargement est en effet que **les fichiers postés sont immédiatement placés dans un répertoire temporaire puis immédiatement supprimés par le serveur** par mesure de sécurité. L'objectif du développement consiste donc à récupérer les fichiers avant qu'ils ne soient supprimés.



La liste des erreurs est disponible dans le manuel PHP : <http://php.net/manual/fr/features.fileupload.errors.php>
Des codes erreurs sont également fournis par PHP sous forme de constante.

- **error** : indique le numéro des erreurs survenues. La valeur 0 obtenue dans notre exemple correspond à « Aucune erreur, le téléchargement est correct ». Par exemple `UPLOAD_ERR_OK` est la constante associée à la valeur 0. L'intérêt de `UPLOAD_ERR_OK` est que nous pouvons l'utiliser directement comme nous le verrons ci-dessous ;
- **size** : il s'agit du poids des fichiers transmis en octet. Il faudra donc diviser cette valeur par 1 024 pour obtenir un poids en kilo-octets (Ko).

B. Mise en oeuvre de l'*upload*

Nous venons de réussir un test de chargement de fichiers, mais nous ne pouvons pas voir les fichiers puisque ceux-ci sont immédiatement supprimés du serveur. Nous allons développer un script permettant de récupérer les fichiers postés avant cette suppression. Ce script nous permettra de les placer dans le répertoire de notre choix et de les renommer si nécessaire. Pour commencer, nous poserons les premiers éléments du script en procédural puis nous le finaliserons en **programmation orientée objet (POO)** afin de l'intégrer dans notre application.

Nous nous servons du manuel PHP pour étudier la gestion du téléchargement. La page du manuel relative à l'*upload* est : <http://www.php.net/manual/fr/features.file-upload.post-method.php>

L'exemple #2 intitulé « **Validation de téléchargement de fichiers** » nous fait découvrir le principe et nous indique l'intérêt des deux fonctions `is_uploaded_file` et `move_uploaded_file`.

- la fonction `is_uploaded_file` indique si le fichier a été téléchargé (le nom de la fonction est explicite) ;
- <http://www.php.net/manual/fr/function.is-uploaded-file.php>

C'est la fonction que nous utiliserons afin de déterminer si un fichier a bien été posté et si aucune erreur n'est survenue. Nous l'utiliserons dans un second temps, lors de l'intégration dans l'application, afin de ne pas déclencher le script de gestion du chargement si aucun fichier n'est posté.

- la fonction *move_uploaded_file* déplace un fichier téléchargé (le nom de la fonction est explicite) ;
- <http://www.php.net/manual/fr/function.move-uploaded-file.php>

C'est ensuite la fonction qui nous permettra de récupérer avant sa suppression le fichier téléchargé et de le placer dans le répertoire de notre choix.

Puisque nous chargeons plusieurs fichiers, nous utiliserons le code de l'exemple proposé dans la page de présentation de *move_uploaded_file()* : Exemple #1 « **Téléchargement de plusieurs fichiers** ».

Il est tout à fait permis d'utiliser un exemple de script du manuel PHP **à condition que l'effort soit fait de le comprendre ligne par ligne, intégralement.**

Ce script est intéressant car il prend le parti de démarrer par la gestion des erreurs. Qu'il y ait ou non des erreurs, un code « *error* » est toujours fourni lors du chargement, et donc il s'agit d'une démarche très pertinente qui va permettre de gérer tout de suite les différentes situations.

```
foreach ($_FILES["upload"]["error"] as $key => $error) {}
```

Fig. 5

Le tableau associatif de la variable *\$_FILES* est ainsi utilisé directement avec une boucle *foreach* qui va parcourir le tableau associé à la clef *\$_FILES["upload"]["error"]*. Rappelons qu'un tableau associatif fait correspondre un tableau de données à une clef du tableau *parent*.

La boucle utilise la pseudo-variable *\$error* et nous pouvons alors vérifier avec le code *UPLOAD_ERR_OK* qu'aucune erreur n'est survenue en effectuant la vérification suivante :

```
if ($error == UPLOAD_ERR_OK) {}
```

Fig. 6

Si aucune erreur n'est survenue, nous pouvons alors déplacer le fichier avec la fonction *move_uploaded_file()*. Très logiquement, cette fonction utilise deux arguments :

- **le chemin d'origine** : c'est-à-dire le **chemin temporaire** attribué par le serveur et qui va permettre de définir quel est le fichier chargé qu'il faut déplacer. Le chemin temporaire est la valeur du tableau *\$_FILES["upload"]["tmp_name"]* ;

- **le chemin de destination** : c'est-à-dire le répertoire de destination et le nom final du fichier. Un chemin (*path* en anglais) est, en effet, toujours constitué de ces deux données : le répertoire et le nom de fichier. Il revient au développeur de déterminer quel est le chemin des destinations. Dans notre exemple, nous utiliserons le répertoire « images » de notre application et nous conviendrons de garder le nom de fichier initial. Cette valeur sera affectée par commodité à une variable `$upload_dir`.

```
$upload_dir = 'C:\wamp\www\projet_image\images/';
foreach ($_FILES['upload']['error'] as $key => $error)
{
    if ($error == UPLOAD_ERR_OK) // aucune erreur
    {
        $tmp_name = $_FILES['upload']['tmp_name'][$key];
        $name = $_FILES['upload']['name'][$key];
        move_uploaded_file($tmp_name, $upload_dir . $name); // déplacement
    }
}
```

Fig.7

Nous ajouterons par la suite une gestion des erreurs, car notre gestionnaire est destiné uniquement à la gestion d'images et il sera donc nécessaire d'empêcher le chargement d'autres types de fichiers.

Nous vérifions que ce code fonctionne en nous rendant sur notre page d'administration, et en l'actualisant si nécessaire (touche F5 du clavier), afin de constater que deux nouvelles images ont été chargées.

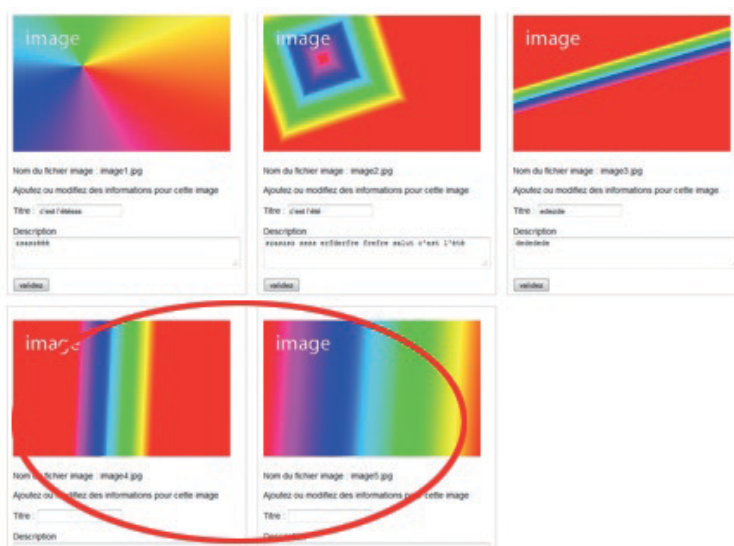


Fig.8

Les deux nouvelles images ont bien été téléchargées et sont prêtes à être éditées dans la base de données. Le script est donc bien fonctionnel et il ne nous reste plus qu'à l'intégrer dans notre application.

C. Intégration du script dans l'application

L'intégration nécessite généralement une réflexion en deux étapes :

- **l'ergonomie**, c'est-à-dire la manière dont l'utilisateur va pouvoir utiliser cette nouvelle fonctionnalité dans le cadre de l'interface d'administration ;
- **le fonctionnel**, il s'agit de définir les actions souhaitées par la fonctionnalité.

Le développement qui s'ensuivra consistera à mettre en œuvre ces deux points.

1. Ergonomie

L'intégration dans l'interface ergonomique est censée permettre un accès rapide à l'*upload* et de revenir soit dans l'*admin* soit dans la page front. Nous ajouterons donc un menu de navigation latéral contenant les éléments de liste suivants : *Administration*, *Upload* et *Site Front* qui seront des liens vers les pages respectives.

Les messages d'erreur ou de succès seront ajoutés. Le travail sera effectué en HTML et ne présente pas de difficulté particulière.

2. Fonctionnel

La fonctionnalité d'*upload* actuelle est intéressante en soi puisqu'elle permet de charger des fichiers sur le serveur. Cependant il peut être pertinent de considérer les besoins suivants :

- gérer la création de vignettes, c'est-à-dire les images miniatures, afin d'optimiser la composition des pages ;
- renommer les fichiers : afin que par exemple les noms de fichier ne contiennent pas d'espace et qu'ils contiennent le nom du projet ;
- empêcher l'*upload* de fichiers autres que de type image.

Nous intégrerons donc ces points dans le développement.

3. Réorganisation des répertoires et des fichiers – principe du MVC

Nous créons les répertoires suivants :

- **class** : l'ensemble des fichiers contenant nos classes (ici : **Image.php**) ;
- **process** : l'ensemble des fichiers destinés à effectuer les actions ;
- **la racine /admin/** : dans lequel nous placerons l'ensemble des fichiers qui permettent d'afficher les informations.

Cette organisation en trois parties est caractéristique de l'architecture dite **MVC** pour **Model**, **View**, **Controller**.

- **M : Model** (ou Modèle) c'est-à-dire les classes ;
- **V : View** (ou Vue) c'est-à-dire les pages qui affichent des informations ;
- **C : Controller** (Contrôleur) c'est-à-dire les fichiers qui appellent les classes et après calcul et/ou contrôle des informations, envoient les résultats pour affichage aux *Vues*.

Chaque type de fichier a un rôle précis à jouer, et le code gagne en lisibilité et en maintenabilité. Nous reviendrons plus avant sur l'architecture MVC dans le prochain cours, mais celle-ci doit nous guider dans toutes nos organisations de fichiers.



Les fichiers **index.php** seront créés ultérieurement afin de pouvoir continuer, dans le cadre de cette démonstration, à accéder de manière rapide et pratique aux listes des fichiers via les URLs des répertoires.

http://localhost/projet_image/admin/

http://localhost/projet_image/

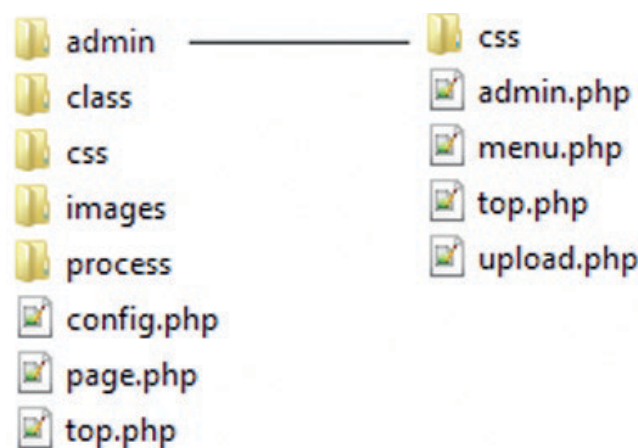


Fig.9 Réorganisation des fichiers et répertoires de l'application

Naturellement, les chemins d'accès aux fichiers inclus tiennent compte de cette restructuration comme le montre le fichier **admin.php** :

```
require ('../config.php') ;
require ('../class/Image.php') ;
require ('top.php') ;
require ('menu.php') ;
require ('../process/process_image.php') ;
```

Fig.10

Nous en profitons également pour ajouter une navigation à notre espace de gestion. Cette navigation est une simple liste contenue dans le fichier **menu.php**.

```
<ul class="menu">
<li><a href="admin.php">admin</a></li>
<li><a href="upload.php">Upload</a></li>
<li><a href="../page.php">site web</a></li>
</ul>
```

Fig.11

Quelques instructions CSS nous permettent ensuite d'obtenir un espace de gestion organisé nous permettant de naviguer entre l'accueil de l'administration, l'interface d'*upload* et le site *front*.



Fig. 12

Nous déportons ensuite le fonctionnement de l'*upload* dans la classe *Image* afin de ne pas la localiser dans le fichier **upload.php** mais au contraire d'en faire une méthode de l'objet *Image* qui pourra si besoin être réutilisée ailleurs, dans d'autres fichiers.

Nous créons donc la méthode en l'appelant simplement *upload* et en récupérant le code initial basé sur la variable `$_FILES` que nous utilisons comme argument. Puisque nous passons donc toutes les informations contenues dans `$_FILES`, nous respectons de fait la structure de ce tableau de données, sauf que, en tant qu'argument, nous la renommons en *\$files*.

```
public function upload ($files) // $files contiendra
```

Fig. 13

Nous gérons également les messages d'erreur ou de succès. Dans le cas présent, nous déclarons une variable `$error = 0` qui jouera le rôle d'un **itérateur** et que nous pourrions incrémenter à chaque erreur rencontrée :

- soit dans le cas d'une erreur d'*upload* ;
- soit dans le cas d'une erreur de *move_uploaded_file*.

Si la variable `$error` contient toujours zéro à la fin du processus, alors l'opération complète s'est bien passée et la méthode retournera *TRUE*. Sinon la méthode retournera *FALSE*. Nous utiliserons ensuite ce résultat dans le fichier **upload.php**.

Notons que nous pourrions et devrions affiner ces messages d'erreur en récupérant précisément le problème et en l'affichant correctement. Ceci constitue un excellent exercice que vous êtes invité à réaliser.

Voici le code de la méthode *upload* de la classe *Image* :

```
public function upload ($files)
{
    $upload_dir = IMAGE_DIR_PATH;
    foreach ($files['upload']['error'] as $key => $error)
    {
        $error = 0;
        if ($error == UPLOAD_ERR_OK)
        {
            $tmp_name = $files['upload']['tmp_name'][$key];
            $name = $files['upload']['name'][$key];
            if (move_uploaded_file($tmp_name, $upload_dir . $name)
            == false)
            $error++ ;
        }
        else
        {
            $error++ ;
        }
    }
    if($error == 0)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

Fig. 14

Voici le nouveau code de la page **upload.php**.

```
<?php
require('../config.php');
require('../class/Image.php');
require('top.php');
require('menu.php');
require('../process/process_image.php');
if (!empty($_FILES))
{
    $image = new Image();
    $images = $image->upload($_FILES);
    if( $images === true)
    {
        $msg_success = 'Le chargement a réussi';
    }
    else
    {
        $msg_error = 'Le chargement a échoué';
    }
}
?>
<h1>Upload</h1>
<?php if(isset($msg_success)): ?>
<p class="msg_success"><?php echo $msg_success ?></p>
<?php endif ?>
<?php if(isset($msg_error)): ?>
<p class="msg_error"><?php echo $msg_error ?></p>
<?php endif ?>
<form id="uploadForm" action="" method="post" enctype="multipart/
form-data">
<p>Ajoutez des images</p>
<input type="file" value="" name="upload[]" multiple="multiple">
<input id="uploadFormSubmit" name="uploadFormSubmit" type="submit">
</form>
```

Fig. 15

Upload

Le chargement a réussi

Ajoutez des images

- [admin](#)
- [Upload](#)
- [site web](#)

Fig. 16

Notons qu'une optimisation intéressante d'un point de vue ergonomie serait d'afficher les nouvelles images que nous venons d'uploader avec un lien pour accéder à leur gestion, ou encore mieux, de restituer directement le formulaire d'édition pour chacune des nouvelles images de manière à les éditer aussitôt le chargement effectué.

II. Sécurisation

A. Empêcher le chargement de certains types de fichiers

Il est important de prendre en compte la partie sécurité de l'application. Compte tenu du fait qu'un formulaire d'*upload* permet à la base de charger tout type de fichiers, y compris des fichiers PHP.

Une personne mal intentionnée peut charger un fichier PHP contenant un code malicieux destiné à pirater l'application.

Il faut donc toujours empêcher que certains types de fichiers puissent être chargés. Partant du principe que tout ce qui n'est pas interdit est autorisé, une bonne pratique consiste à autoriser seulement les types de fichiers attendus et à interdire tous les autres. Nous allons donc autoriser uniquement les fichiers de type JPG.

Pour ce faire, une première approche consisterait à vérifier dans le nom du fichier que le fichier possède bien l'extension .jpg ou .jpeg. Mais cela serait incomplet car il est en effet très simple de modifier l'extension d'un fichier en le renommant avant l'*upload*.

Nous utiliserons donc l'analyse du fichier telle qu'elle est restituée par la variable `$_FILES` lors de l'*upload*. Rappelons-nous que cette variable contient la donnée type qui définit justement le type des fichiers uploadés. C'est grâce à cette donnée que nous pourrions vérifier si les fichiers sont réellement de type JPG ou non.

Le type fourni s'appelle le **type MIME** (*Multipurpose Internet Mail Extensions*) et il est défini de façon standard. Ainsi un fichier .jpg possède le type MIME suivant : *image/jpeg*.

Le Type MIME (ou simplement : « le MIME ») est synonyme de Content-type. La liste des MIME est accessible et mise à jour sur cette page : <http://svn.apache.org/viewvc/httpd/httpd/branches/2.2.x/docs/conf/mime.types?view=annotate>

Il suffit donc de vérifier que le type est *image/jpeg* pour autoriser l'*upload*. A noter que s'agissant également d'images, nous pourrions autoriser les images de type GIF (MIME : *image/gif*) ou PNG (MIME : *image/png*).

Nous ajouterons donc dans la boucle *foreach* le code suivant :

```
$type = $_FILES['upload']['type'][$key];
if($type == 'image/jpeg') // seules les fichiers jpg sont autorisés
{
    # le code existant
}
else
{
    $error++ ;
}
```

Fig. 17

B. Sécurité : limiter le poids des fichiers à uploader

De la même manière il est souhaitable de limiter le poids des fichiers que les utilisateurs peuvent charger. En effet plus le poids du fichier est lourd, plus le temps de chargement pris par PHP est long et plus les ressources du serveur sont consommées. Afin d'empêcher un dysfonctionnement du serveur en cas de chargement trop long, il est recommandé de limiter la taille des fichiers chargés.

En HTML nous pouvons ajouter l'attribut `MAX_FILE_SIZE` qui permet de limiter le poids des fichiers, mais d'un point de vue sécurité, il s'agit d'une mesure insuffisante car une personne mal intentionnée connaissant le HTML n'aura aucune difficulté à dupliquer le formulaire et à supprimer cet attribut. Il faut donc nécessairement effectuer cette limitation en PHP, côté serveur.

Nous utiliserons là encore les informations fournies par la variable `$_FILES`, notamment la donnée `size` dont nous avons vu qu'elle contenait le poids des fichiers transmis en octet. Très simplement nous fixerons donc une valeur limite de 1 Méga-octet soit 10 000 000 octets (106, c'est-à-dire un « 10 » suivi de 6 zéros) et ajouterons la ligne de code idoine.