ORGANISER SON CODE SELON L'ARCHITECTURE MVC

Beaucoup de débutants en PHP disent avoir des difficultés à organiser le code de leur site web. Ils savent créer des scripts, comme un mini-chat par exemple, mais ne se sentent pas capables de concevoir leur site web de manière globale. « Quels dossiers dois-je créer ? », « Comment dois-je organiser mes fichiers ? », « Ai-je besoin d'un dossier admin ? », etc.

L'objectif de ce chapitre est de vous faire découvrir l'architecture MVC, une bonne pratique de programmation qui va vous aider à bien concevoir votre futur site web. Après sa lecture, vous vous sentirez largement plus capables de créer un site web de qualité et facile à maintenir. ;-)

Qu'est-ce que l'architecture MVC ?

Vous vous posez certainement beaucoup de questions sur la bonne façon de concevoir votre site web. Laissez-moi vous rassurer à ce sujet : ces questions, nous nous les sommes tous posées un jour. En fait, il y a des problèmes en programmation qui reviennent tellement souvent qu'on a créé toute une série de bonnes pratiques que l'on a réunies sous le nom de *design patterns*.

Un des plus célèbres *design patterns* s'appelle MVC, qui signifie **Modèle - Vue - Contrôleur**. C'est celui que nous allons découvrir dans ce chapitre.

Le pattern MVC permet de bien organiser son code source. Il va vous aider à savoir quels fichiers créer, mais surtout à définir leur rôle. Le but de MVC est justement de séparer la logique du code en trois parties que l'on retrouve dans des fichiers distincts, comme l'explique la description qui suit.

- **Modèle** : cette partie gère les *données* de votre site. Son rôle est d'aller récupérer les informations « brutes » dans la base de données, de les organiser et de les assembler pour qu'elles puissent ensuite être traitées par le contrôleur. On y trouve donc les requêtes SQL.

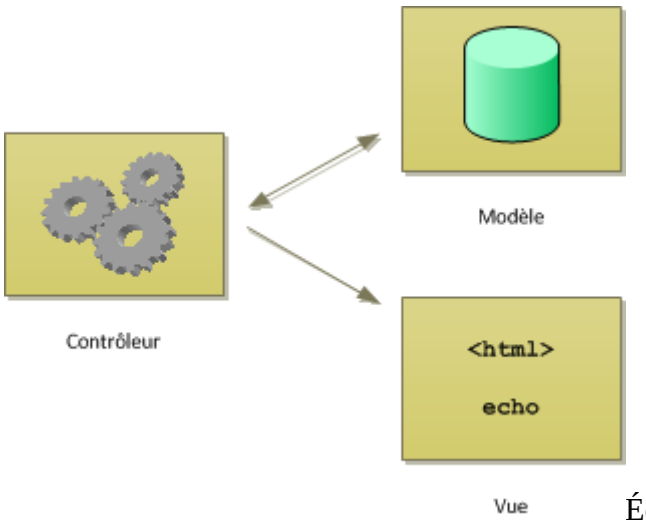
  Parfois, les données ne sont pas stockées dans une base de données. C'est plus rare, mais on peut être amené à aller chercher des données dans des fichiers. Dans ce cas, le rôle du modèle est de faire les opérations d'ouverture, de lecture et d'écriture de fichiers (fonctions `fopen`,`fgets`, etc.).
- **Vue** : cette partie se concentre sur l'*affichage*. Elle ne fait presque aucun calcul et se contente de récupérer des variables pour savoir ce qu'elle doit afficher. On y trouve essentiellement du code HTML mais aussi quelques boucles et conditions PHP très simples, pour afficher par exemple la liste des messages des forums.

- **Contrôleur** : cette partie gère la logique du code qui prend des *décisions*. C'est en quelque sorte l'intermédiaire entre le modèle et la vue : le contrôleur va demander au modèle les données, les analyser, prendre des décisions et renvoyer le texte à afficher à la vue. Le contrôleur contient exclusivement du PHP. C'est notamment lui qui détermine si le visiteur a le droit de voir la page ou non (gestion des droits d'accès).

La figure suivante schématise le rôle de chacun de ces éléments.

L'architecture MVC

Il est important de bien comprendre comment ces éléments s'agencent et communiquent entre eux. Regardez bien la figure suivante.



Échange d'informations entre les éléments

Il faut tout d'abord retenir que le contrôleur est le chef d'orchestre : c'est lui qui reçoit la requête du visiteur et qui contacte d'autres fichiers (le modèle et la vue) pour échanger des informations avec eux.

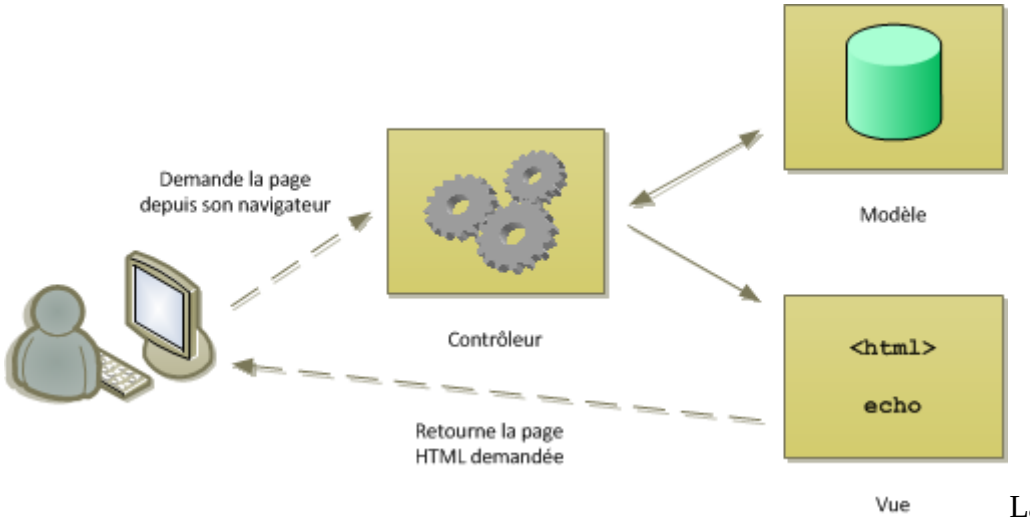
Le fichier du contrôleur demande les données au modèle sans se soucier de la façon dont celui-ci va les récupérer. Par exemple : « Donne-moi la liste des 30 derniers messages du forum no 5 ». Le modèle traduit cette demande en une requête SQL, récupère les informations et les renvoie au contrôleur.

Une fois les données récupérées, le contrôleur les transmet à la vue qui se chargera d'afficher la liste des messages.

Le contrôleur sert seulement à faire la jonction entre le modèle et la vue finalement, non ?

Dans les cas les plus simples, ce sera probablement le cas. Mais comme je vous le disais, le rôle du contrôleur ne se limite pas à cela : s'il y a des calculs ou des vérifications d'autorisations à faire, des images à miniaturiser, c'est lui qui s'en chargera.

Concrètement, le visiteur demandera la page au contrôleur et c'est la vue qui lui sera retournée, comme schématisé sur la figure suivante. Bien entendu, tout cela est transparent pour lui, il ne voit pas tout ce qui se passe sur le serveur. C'est un schéma plus complexe que ce à quoi vous avez été habitués, bien évidemment : c'est pourtant sur ce type d'architecture que repose un très grand nombre de sites professionnels !



La requête du client arrive au contrôleur et celui-ci lui retourne la vue
Tout ce que je vous présente là doit vous paraître bien beau, mais encore très flou. Je vais vous montrer dans la pratique comment on peut respecter ce principe en PHP.

Le code du TP blog et ses défauts

Nous allons reprendre le TP blog sur lequel nous avons travaillé un peu plus tôt dans le cours.

Nous allons nous intéresser au code de la page qui affiche les derniers billets du blog. Voici ce que nous avions produit à la fin de ce TP :

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Mon blog</title>
    <link href="style.css" rel="stylesheet" />
    </head>

    <body>
        <h1>Mon super blog !</h1>
        <p>Derniers billets du blog :</p>

<?php
// Connexion à la base de données
try
{
```

```
    $bdd = new PDO('mysql:host=localhost;dbname=test;charset=utf8', 'root', '');
}
catch(Exception $e)
{
        die('Erreur : '.$e->getMessage());
}

// On récupère les 5 derniers billets
$req = $bdd->query('SELECT id, titre, contenu, DATE_FORMAT(date_creation, \'%d/%m/%Y à
%Hh%imin%ss\') AS date_creation_fr FROM billets ORDER BY date_creation DESC LIMIT 0, 5');

while ($donnees = $req->fetch())
{
?>
<div class="news">
    <h3>
        <?php echo htmlspecialchars($donnees['titre']); ?>
        <em>le <?php echo $donnees['date_creation_fr']; ?></em>
    </h3>

    <p>
    <?php
    // On affiche le contenu du billet
    echo nl2br(htmlspecialchars($donnees['contenu']));
    ?>
    <br />
    <em><a href="commentaires.php?billet=<?php echo $donnees['id'];
?>">Commentaires</a></em>
    </p>
</div>
<?php
} // Fin de la boucle des billets
$req->closeCursor();
?>
</body>
</html>
```

Et alors ? Ce code marche bien, non ? On repère les requêtes SQL, le HTML, etc. Moi je le comprends bien en tout cas, je ne vois pas ce qu'il faut améliorer !

À l'époque, nous avions écrit le script PHP de façon intuitive sans trop nous soucier de son organisation. Résultat : notre code était un joyeux mélange d'instructions PHP, de requêtes SQL et de HTML. Sur une page simple comme celle-là, cela ne posait pas trop de problèmes car il n'y avait pas beaucoup de lignes de code. En revanche, si vous ajoutez plus tard des fonctionnalités et des requêtes SQL à cette page… cela va vite devenir un vrai capharnaüm !

Si on prend la peine de construire cette page en respectant l'architecture MVC, cela va nous prendre un peu plus de temps : il y aura probablement trois fichiers au lieu d'un seul, mais au final le code sera beaucoup plus clair et mieux découpé.

L'intérêt de respecter l'architecture MVC ne se voit pas forcément de suite. En revanche, si vous améliorez votre site plus tard, vous serez bien heureux d'avoir pris la peine de vous organiser avant ! De plus, si vous travaillez à plusieurs, cela vous permet de séparer les tâches : une personne peut s'occuper du contrôleur, une autre de la vue et la dernière du modèle. Si vous vous mettez d'accord au préalable sur les noms des variables et fonctions à

appeler entre les fichiers, vous pouvez travailler ensemble en parallèle et avancer ainsi beaucoup plus vite !

Amélioration du TP blog en respectant l'architecture MVC

Je vais maintenant vous montrer comment on pourrait découper le code précédent selon une architecture MVC.
Je vais vous proposer *une* façon de faire, mais n'allez pas croire que c'est la seule méthode qui existe. On peut respecter l'architecture MVC de différentes manières ; tout dépend de la façon dont on l'interprète. D'ailleurs, la théorie « pure » de MVC est bien souvent inapplicable en pratique. Il faut faire des concessions, savoir être pragmatique : on prend les bonnes idées et on met de côté celles qui se révèlent trop contraignantes.

À la racine de votre site, je vous propose de créer trois répertoires :

- `modele` ;
- `vue` ;
- `controleur`.

Dans chacun d'eux, vous pouvez créer un sous-répertoire pour chaque « module » de votre site : `forums`,`blog`, `minichat`, etc. Pour le moment, créez un répertoire `blog` dans chacun de ces dossiers. On aura ainsi l'architecture suivante :

- `modele/blog` : contient les fichiers gérant l'accès à la base de données du blog ;
- `vue/blog` : contient les fichiers gérant l'affichage du blog ;
- `controleur/blog` : contient les fichiers contrôlant le fonctionnement global du blog.

On peut commencer par travailler sur l'élément que l'on veut ; il n'y a pas d'ordre particulier à suivre (comme je vous le disais, on peut travailler à trois en parallèle sur chacun de ces éléments). Je vous propose de commencer par le modèle, puis de voir le contrôleur et enfin la vue.

Le modèle

Créez un fichier `get_billets.php` dans `modele/blog`. Ce fichier contiendra une fonction dont le rôle sera de retourner un certain nombre de billets depuis la base de données. C'est tout ce qu'elle fera.

```php
<?php
function get_billets($offset, $limit)
{
    global $bdd;
    $offset = (int) $offset;
    $limit = (int) $limit;
```

```
    $req = $bdd->prepare('SELECT id, titre, contenu, DATE_FORMAT(date_creation, \'%d/%m/%Y
à %Hh%imin%ss\') AS date_creation_fr FROM billets ORDER BY date_creation DESC LIMIT
:offset, :limit');
    $req->bindParam(':offset', $offset, PDO::PARAM_INT);
    $req->bindParam(':limit', $limit, PDO::PARAM_INT);
    $req->execute();
    $billets = $req->fetchAll();


    return $billets;
}
```

Ce code source ne contient pas de réelles nouveautés. Il s'agit d'une fonction qui prend en paramètre un *offset*et une *limite*. Elle retourne le nombre de billets demandés à partir du billet no*offset*. Ces paramètres sont transmis à MySQL via une requête préparée.

Je n'ai pas utilisé la méthode traditionnelle à laquelle vous avez été habitués pour transmettre les paramètres à la requête SQL. En effet, j'ai utilisé ici la fonction `bindParam` qui me permet de préciser que le paramètre est un entier (`PDO$\colon\colon$PARAM_INT`). Cette méthode alternative est obligatoire dans le cas où les paramètres sont situés dans la clause `LIMIT` car il faut préciser qu'il s'agit d'entiers.
La connexion à la base de données aura été faite précédemment. On récupère l'objet `$bdd` global représentant la connexion à la base et on l'utilise pour effectuer notre requête SQL. Cela nous évite d'avoir à recréer une connexion à la base de données dans chaque fonction, ce qui serait très mauvais pour les performances du site (et très laid dans le code !).

Il existe une meilleure méthode pour récupérer l'objet `$bdd` et qui est basée sur le design pattern singleton. Elle consiste à créer une classe qui retourne toujours le même objet. Nous ne détaillerons pas cette méthode ici, sensiblement plus complexe, mais je vous invite à vous renseigner sur le sujet.
Plutôt que de faire une boucle de `fetch()`, j'appelle ici la fonction `fetchAll()` qui assemble toutes les données dans un grand array. La fonction retourne donc un array contenant les billets demandés.

Vous noterez que ce fichier PHP ne contient pas la balise de fermeture `?>`. Celle-ci n'est en effet pas obligatoire, comme je vous l'ai dit plus tôt dans le cours. Je vous recommande de ne pas l'écrire surtout dans le modèle et le contrôleur d'une architecture MVC. Cela permet d'éviter de fâcheux problèmes liés à l'envoi de HTML avant l'utilisation de fonctions comme `setCookie` qui nécessitent d'être appelées avant tout code HTML.

Le contrôleur

Le contrôleur est le chef d'orchestre de notre application. Il demande au modèle les données, les traite et appelle la vue qui utilisera ces données pour afficher la page.

Dans `controleur/blog`, créez un fichier `index.php` qui représentera la page d'accueil du blog.

```php
<?php

// On demande les 5 derniers billets (modèle)
include_once('modele/blog/get_billets.php');
$billets = get_billets(0, 5);

// On effectue du traitement sur les données (contrôleur)
// Ici, on doit surtout sécuriser l'affichage
foreach($billets as $cle => $billet)
{
    $billets[$cle]['titre'] = htmlspecialchars($billet['titre']);
    $billets[$cle]['contenu'] = nl2br(htmlspecialchars($billet['contenu']));
}

// On affiche la page (vue)
include_once('vue/blog/index.php');
```

On retrouve le cheminement simple que je vous avais décrit. Le rôle de la page d'accueil du blog est d'afficher les cinq derniers billets. On appelle donc la fonction `get_billets()` du modèle, on récupère la liste « brute » de ces billets que l'on traite dans un `foreach` pour protéger l'affichage avec `htmlspecialchars()` et créer les retours à la ligne du contenu avec `nl2br()`.
S'il y avait d'autres opérations à faire avant l'appel de la vue, comme la gestion des droits d'accès, ce serait le bon moment. En l'occurrence, il n'est pas nécessaire d'effectuer d'autres opérations dans le cas présent.

Notez que la présence des fonctions de sécurisation des données dans le contrôleur est discutable. On pourrait laisser cette tâche à la vue, qui ferait donc les `htmlspecialchars`, ou bien à une couche intermédiaire entre le contrôleur et la vue (c'est d'ailleurs ce que proposent certains *frameworks* dont on parlera plus loin). Comme vous le voyez, il n'y a pas une seule bonne approche mais plusieurs, chacune ayant ses avantages et inconvénients.
Vous noterez qu'on opère sur les clés du tableau plutôt que sur `$billet` (sans *s*) directement. En effet, `$billet` est une copie du tableau `$billets` créée par le `foreach`. `$billet` n'existe qu'à l'intérieur du `foreach`, il est ensuite supprimé. Pour éviter les failles XSS, il faut agir sur le tableau utilisé à l'affichage, c'est-à-dire `$billets`.
Ce qui est intéressant, c'est que la fonction `get_billets()` pourra être réutilisée à d'autres occasions. Ici, on l'appelle toujours avec les mêmes paramètres, mais on pourrait en avoir besoin dans d'autres contrôleurs. On pourrait aussi améliorer ce contrôleur-ci pour gérer la pagination des billets du blog et afficher un nombre différent de billets par page en fonction des préférences du visiteur.

La vue

Il ne nous reste plus qu'à créer la vue correspondant à la page d'accueil des derniers billets du blog. Ce sera très simple : le fichier de la vue devra simplement afficher le contenu de

l'array `$billets` sans se soucier de la sécurité ou des requêtes SQL. Tout aura été préparé avant.

Vous pouvez créer un fichier `index.php` dans `vue/blog` et y insérer le code suivant :

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Mon blog</title>
    <link href="vue/blog/style.css" rel="stylesheet" />
    </head>

    <body>
        <h1>Mon super blog !</h1>
        <p>Derniers billets du blog :</p>

<?php
foreach($billets as $billet)
{
?>
<div class="news">
    <h3>
        <?php echo $billet['titre']; ?>
        <em>le <?php echo $billet['date_creation_fr']; ?></em>
    </h3>

    <p>
    <?php echo $billet['contenu']; ?>
    <br />
    <em><a href="commentaires.php?billet=<?php echo $billet['id'];
?>">Commentaires</a></em>
    </p>
</div>
<?php
}
?>
</body>
</html>
```

Ce code source se passe réellement de commentaires. Il contient essentiellement du HTML et quelques morceaux de PHP pour afficher le contenu des variables et effectuer la boucle nécessaire.

L'intérêt est que ce fichier est débarrassé de toute « logique » du code : vous pouvez aisément le donner à un spécialiste de la mise en page web pour qu'il améliore la présentation. Celui-ci n'aura pas besoin de connaître le PHP pour travailler sur la mise en page du blog. Il suffit de lui expliquer le principe de la boucle et comment on affiche une variable, c'est vraiment peu de choses.

Le contrôleur global du blog

Bien qu'en théorie ce ne soit pas obligatoire, je vous recommande de créer un fichier contrôleur « global » par module à la racine de votre site. Son rôle sera essentiellement de traiter les paramètres `$_GET` et d'appeler le contrôleur correspondant en fonction de la page

demandée. On peut aussi profiter de ce point « central » pour créer la connexion à la base de données.
J'ai ici choisi d'inclure un fichier `connexion_sql.php` qui crée l'objet `$bdd`. Ce fichier pourra ainsi être partagé entre les différents modules de mon site.

Vous pouvez donc créer ce fichier `blog.php` à la racine de votre site :

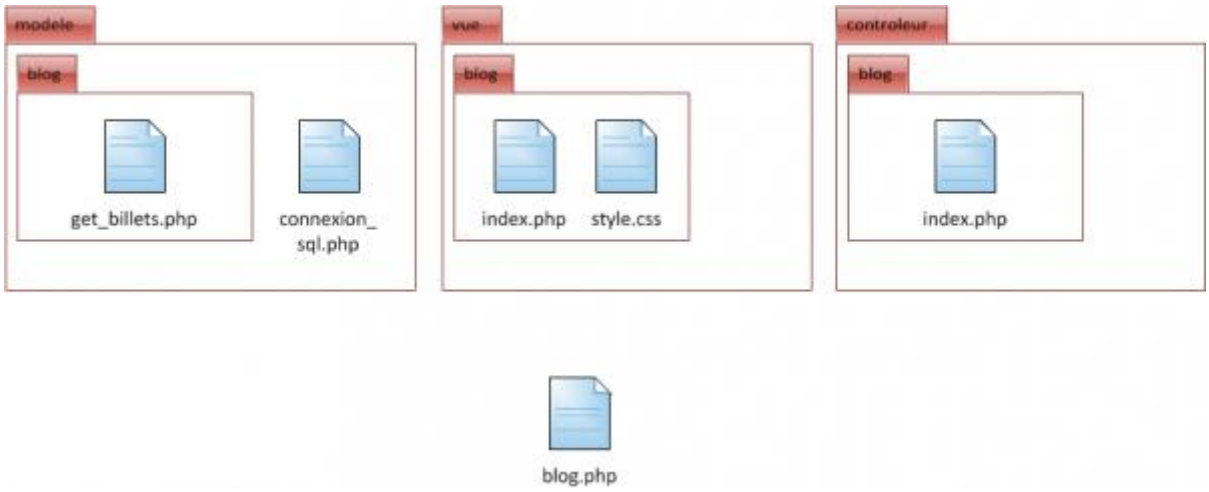
```php
<?php

include_once('modele/connexion_sql.php');

if (!isset($_GET['section']) OR $_GET['section'] == 'index')
{
    include_once('controleur/blog/index.php');
}
```

Pour accéder à la page d'accueil du blog, il suffit maintenant d'ouvrir la page `blog.php` !

L'avantage de cette page est aussi qu'elle permet de masquer l'architecture de votre site au visiteur. Sans elle, ce dernier aurait dû aller sur la page `controleur/blog/index.php` pour afficher votre blog. Cela aurait pu marcher, mais aurait probablement créé de la confusion chez vos visiteurs, en plus de complexifier inutilement l'URL.

Résumé des fichiers créés

Je pense qu'un schéma synthétisant l'architecture des fichiers que nous venons de créer ne sera pas de refus ! La figure suivante devrait vous permettre de mieux vous repérer.



L'architecture des fichiers créés pour adapter notre blog en MVC
Cela fait beaucoup de fichiers alors qu'auparavant tout tenait dans un seul fichier ! La construction de son site selon une architecture MVC est à ce prix. Cela multiplie les fichiers, mais clarifie dans le même temps leur rôle. Si vous les regroupez intelligemment dans des dossiers, il n'y aura pas de problème. ;-)

Désormais, si vous avez un problème de requête SQL, vous savez précisément quel fichier ouvrir : le modèle correspondant. Si c'est un problème d'affichage, vous ouvrirez la vue. Tout cela rend votre site plus facile à maintenir et à faire évoluer !

Je vous invite à télécharger les fichiers d'exemple de ce chapitre pour que vous puissiez vous familiariser avec l'architecture MVC.

Profitez-en pour améliorer le code ! Je vous proposais un peu plus tôt de gérer par exemple la pagination du blog sur la page d'accueil. Vous pouvez aussi porter le reste des pages du blog selon cette même architecture : ce sera le meilleur exercice pour vous former que vous trouverez !

Aller plus loin : les frameworks MVC

L'organisation en fichiers que je vous ai proposée dans ce chapitre n'est qu'une façon de faire parmi beaucoup d'autres. Vous pouvez vous en inspirer comme d'un modèle pour créer votre site, mais rien ne vous y oblige. En fait, l'idéal serait de créer votre site en vous basant sur un *framework* PHP de qualité (mais cela vous demandera du temps, car il faut apprendre à se servir du *framework* en question !).

Un ***framework*** est un ensemble de bibliothèques, une sorte de *kit* prêt à l'emploi pour créer plus rapidement son site web, tout en respectant des règles de qualité. Vous vous souvenez de la bibliothèque GD qui nous permettait de créer des images ? Les *frameworks* sont des assemblages de bibliothèques comme celle-ci. C'est dire si ce sont des outils puissants et complets !

Voici quelques *frameworks* PHP célèbres qu'il faut connaître :

- CodeIgniter ;
- CakePHP ;
- Symfony ;
- Jelix ;
- Zend Framework.

Ils sont tous basés sur une architecture MVC et proposent en outre de nombreux outils pour faciliter le développement de son site web.

Il en existe d'autres, mais ceux-là méritent déjà le coup d'œil. Parmi eux, Symfony2 et le Zend Framework sont probablement les plus célèbres. Ils sont utilisés par de nombreux sites, petits comme grands. Prenez le site de partage de vidéos Dailymotion : il utilise Symfony2. Le Site du Zéro ? Il est basé sur Symfony2 lui aussi !

Ces frameworks ont prouvé leur robustesse et leur sérieux, ce qui en fait des outils de choix pour concevoir des sites de qualité professionnelle.

Pour débuter sur Symfony2 par exemple, le cours d'Alexandre Bacco est un très bon point de départ que je vous invite à consulter.

Ce type de *framework* nécessite de bonnes connaissances en PHP et en programmation orientée objet. Si vous débutez complètement, attendez d'avoir un peu plus d'expérience avant de vous y mettre car il risque de vous paraître un peu complexe.

En résumé

- MVC est un *design pattern*, une bonne pratique de programmation qui recommande de découper son code en trois parties qui gèrent les éléments suivants :
  - **M**odèle : stockage des données ;
  - **V**ue : affichage de la page ;
  - **C**ontrôleur : logique, calculs et décisions.
- Utiliser l'architecture MVC sur son site web en PHP est recommandé, car cela permet d'avoir un code plus facile à maintenir et à faire évoluer.
- De nombreux *frameworks* PHP, tels que Symfony et le Zend Framework, vous permettent de mettre rapidement en place les bases d'une architecture MVC sur votre site. Ce sont des outils appréciés des professionnels qui nécessitent cependant un certain temps d'adaptation.