Architecture MVC

Contenu

[I. MVC 3](#_Toc34037678)

[A. Qu’est-ce qu’un code pro ? 3](#_Toc34037679)

[B. L’architecture de base 4](#_Toc34037680)

[C. L’architecture MVC en détail. 6](#_Toc34037681)

[D. Création d’un Template 7](#_Toc34037682)

[E. Création d’un routeur 10](#_Toc34037683)

[1. Structure 10](#_Toc34037684)

[F. Gérer les erreurs 13](#_Toc34037685)

[G. Passage en programmation Objet 15](#_Toc34037686)

[1. Le modèle 15](#_Toc34037687)

[2. Le controller : 16](#_Toc34037688)

[3. L’héritage 17](#_Toc34037689)

[4. Les namespaces 17](#_Toc34037690)

# MVC

## Qu’est-ce qu’un code pro ?

Voici quelques caractéristiques d'un code professionnel que l'on entend souvent :

* **Il est modulaire** : généralement découpé en de nombreux fichiers, chaque fichier a un et un seul rôle à la fois.
* **Il est découplé :** les fichiers sont conçus pour fonctionner indépendamment les uns des autres.
* **Il est documenté** : la documentation prend généralement la forme de commentaires spéciaux placés au-dessus des méthodes et classes publiques, pouvant être réutilisées dans d'autres projets (renseignez-vous sur la PHPdoc). On peut générer automatiquement une page web de documentation à partir de ces commentaires.
* **Il est en anglais :** c'est la langue des développeurs et développeuses partout sur la planète. Les variables et les noms des fonctions sont en anglais et peuvent être compris par tous.
* **Il est clair :** chaque fichier est généralement assez court (jamais des fichiers de 1000 lignes !), et le code à l'intérieur peut se lire comme de la prose.

Ce genre de code a de nombreux avantages :

* Il est réutilisable : si un jour nous avons codé un fichier utile, nous pouvons nous en resservir dans un autre projet, ou dans un autre endroit du même projet. On gagne du temps en n'ayant pas à tout refaire à chaque fois !
* Il est facile d'y travailler à plusieurs : chaque fichier étant indépendant (et généralement de petite taille), on peut travailler en équipe de 5, 10, voire 100 personnes sur un même projet. Si tout était mélangé dans un seul et même gros fichier, il serait impossible de le modifier en même temps !
* Il est évolutif : quand quelqu'un vient vous demander une nouvelle fonctionnalité, il est facile de l'ajouter. Vous n'avez pas peur de tout casser. Vous savez que ça va marcher, et votre code ne sera pas plus compliqué.

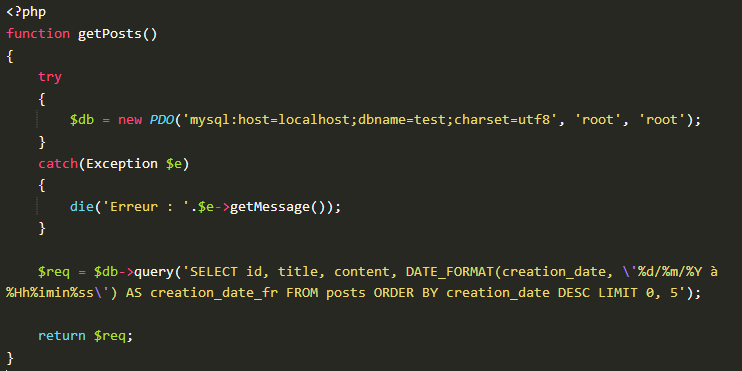
## L’architecture de base

Il faut :

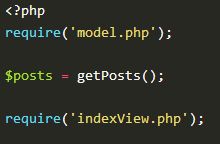
* Isoler l’accès aux données : **Modèle**
* Isoler l’affichage du traitement PHP : **Vue**
* Utilisé un fichier qui fait l’intermédiaire : **Controleur**

Exemple :

Modèle :



Controleur :



On appel toujours le controleur en premier. C’est lui qui fait le lien entre le modèle et la vue.

On remarque qu’on ne ferme pas la balise php pour les fichier php qui n’utilise que du php c’est une convention.

Vue :



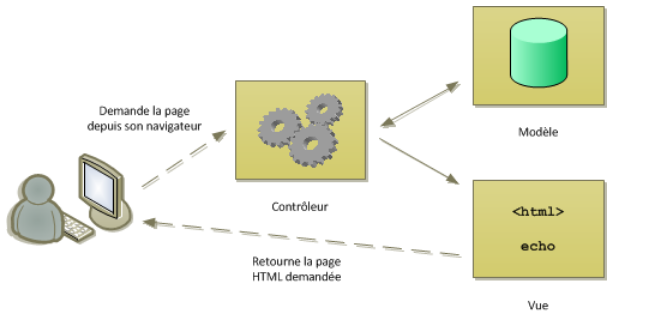
On utilise < ?= ?> au lieu de < ?php ?> cela permet de ne pas toujours mettre la fonction echo. On gagne en lisibilité.

## L’architecture MVC en détail.

L’architecture MVC fonctionne grâce à trois éléments :

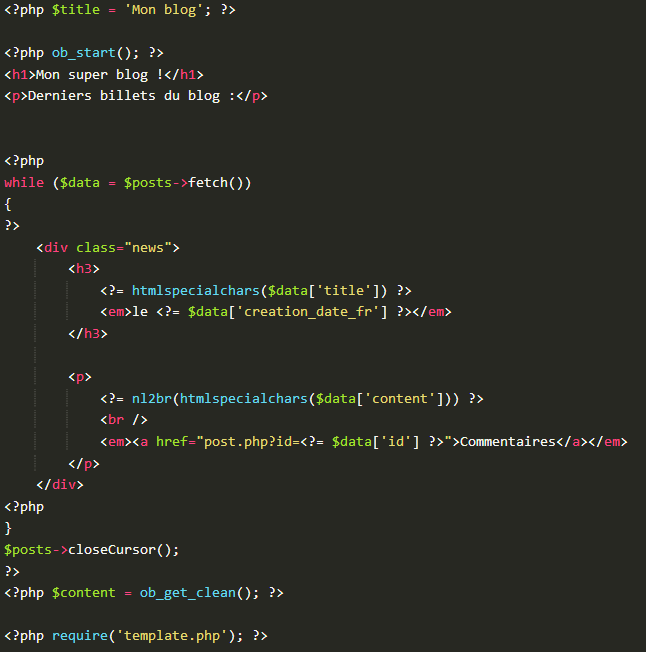


* **Modèle :** cette partie gère les données de votre site. Son rôle est d'aller récupérer les informations « brutes » dans la base de données, de les organiser et de les assembler pour qu'elles puissent ensuite être traitées par le contrôleur. On y trouve donc entre autres les requêtes SQL.
* **Vue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



## Création d’un Template

Utilisation de require et ou d include : la diffrence entre les deux est que require va renvoyer une erreur si jamais le fichier n’existe pas, c’est la seule différence.



Ce code fait 3 choses :

1. Il définit le titre de la page dans $title. Celui-ci sera intégré dans la balise <title> dans le template.
2. Il définit le contenu de la page dans $content. Il sera intégré dans la balise <body> du template.  
   Comme ce contenu est un peu gros, on utilise une astuce pour le mettre dans une variable. On appelle la fonction ob\_start() (ligne 3) qui "mémorise" toute la sortie HTML qui suit, puis, à la fin, on récupère le contenu généré avec ob\_get\_clean()  (ligne 28) et on met le tout dans $content .
3. Enfin, il appelle le template avec un require. Celui-ci va récupérer les variables $title et $content qu'on vient de créer... pour afficher la page !

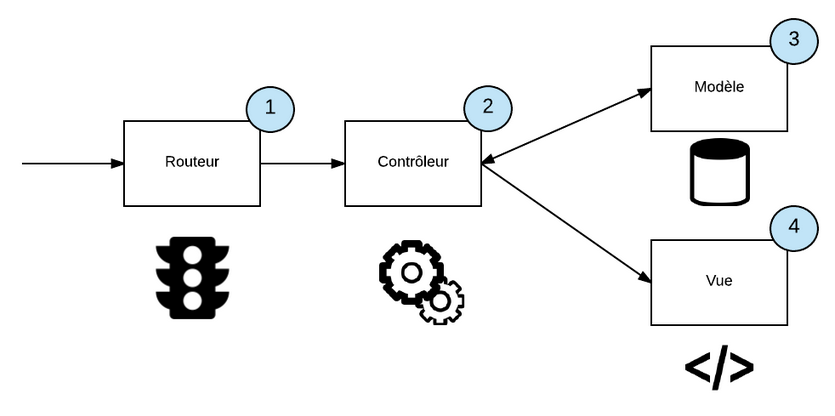
Le second point vous paraît sûrement un peu délicat. Laissez-moi vous proposer une explication en image (j'adore les schémas !) :

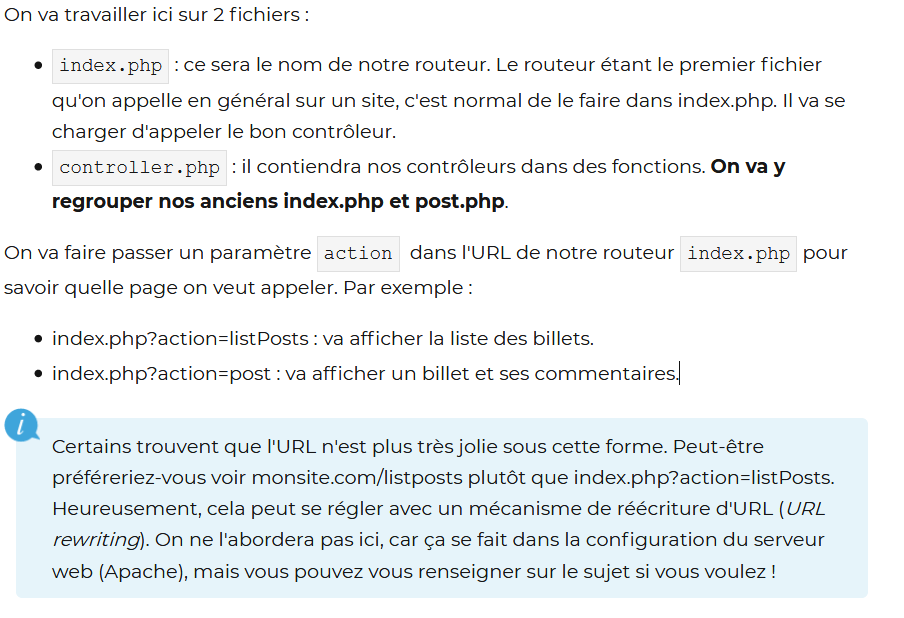


À part l'astuce du ob\_start() et ob\_get\_clean() (qui nous sert juste à mettre facilement beaucoup de code HTML dans une variable), le principe est simple. En inversant l'approche, on a un code bien plus flexible pour définir des "morceaux" de page dans des variables.

## Création d’un routeur

### Structure





Controller :



Routeur :



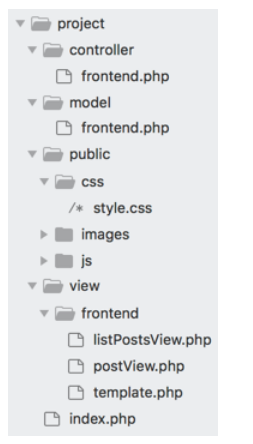
Je pense que ça serait bien d'adopter déjà au minimum l'organisation suivante, que l'on peut retrouver dans un certain nombre projets :

* **controller/ :** le dossier qui contient nos contrôleurs.
* **view/ :** nos vues.
* **model/** : notre modèle.
* **public/ :** tous nos fichiers statiques publics. On pourra y mettre à l'intérieur un dossier css/, images/, js/, etc.

Si sur mon site j'ai un espace "blog", un espace "forum", un espace "members", je pourrais regrouper les fonctions dans des fichiers au nom de ces sections.

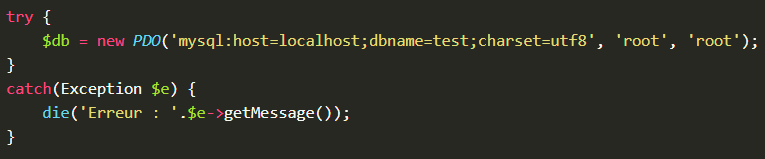
Pour notre blog, je vous propose un autre découpage :

* **frontend :** tout ce qui est côté utilisateur. Affichage des billets, ajout et liste des commentaires...
* **backend :** tout ce qui est pour les administrateurs. Création de billets, modération des commentaires...



## Gérer les erreurs

Pour gérer les erreurs nous utilisons la fonction : Try et catch



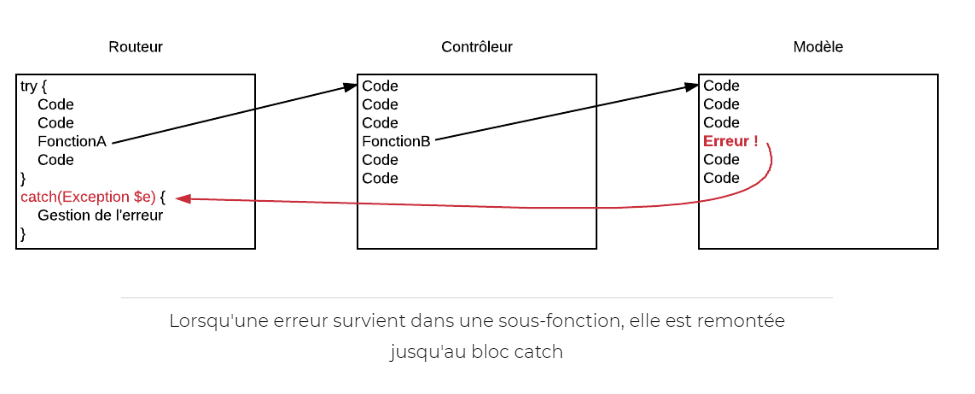
On affiche le message d’erreur grâce à la méthode getMessage de la classe Exception.

Il est possible d’étendre la fonction sur l’ensemble d’un code et non pas seulement sur quelque instruction.

Pour cela on utilise l’instruction throw new Exception ( ‘Message d’erreur’ ) ;

Ici nous l’utilisons dans le routeur.

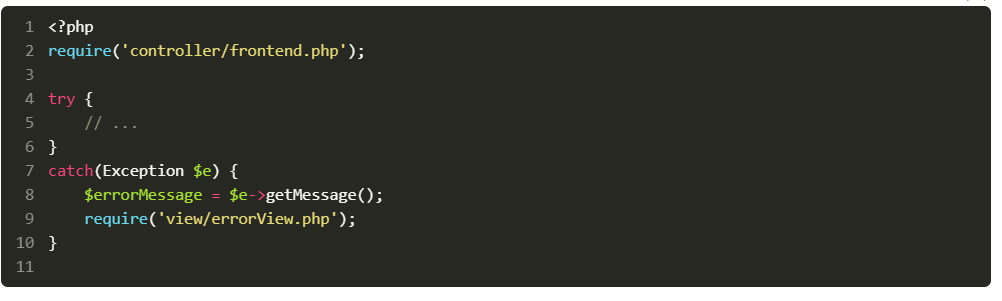






Pour l'instant, notre bloc catch affiche une erreur avec un simple echo . Si nous voulons faire quelque chose de plus joli, nous pouvons appeler une vue errorView.php qui affiche joliment le message d'erreur.

Il faudrait faire quelque chose dans ce goût-là :



## Passage en programmation Objet

### Le modèle

Histoire de bien faire les choses, on va créer 2 classes car on peut considérer qu'on a 2 types d'objets différents :

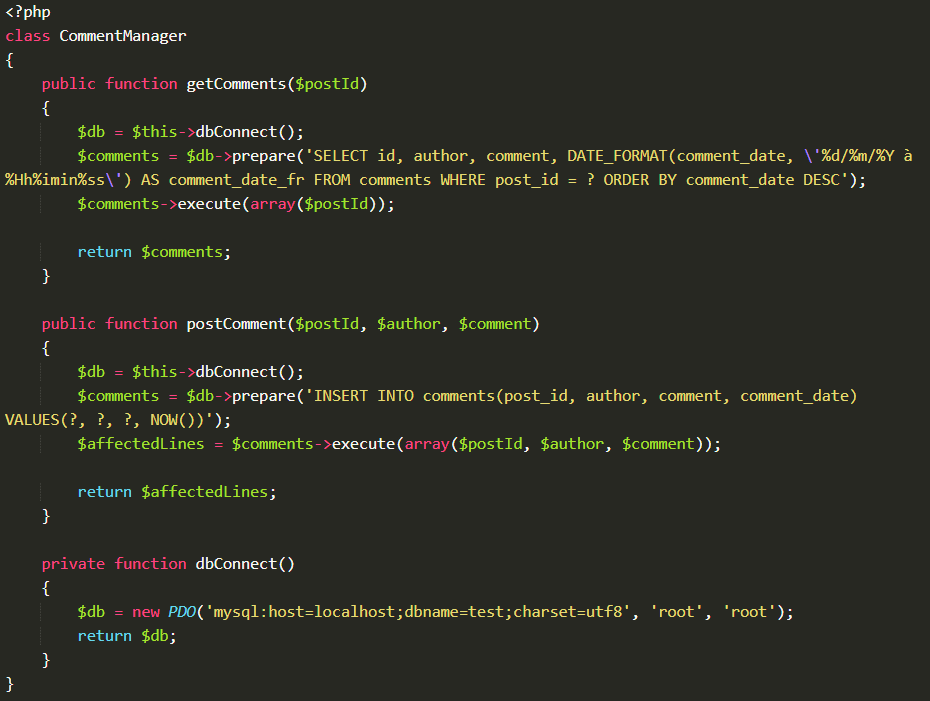
* PostManager  : un gestionnaire de post de blog
* CommentManager  : un gestionnaire de commentaire

On va donc avoir 2 fichiers :

* model/PostManager.php
* model/CommentManager.php

Par convention on vas appeler les Classe Manager car elles vonts nous aider à manager les données. Effectuer des actions de lecture et d’écriture.





### Le controller :



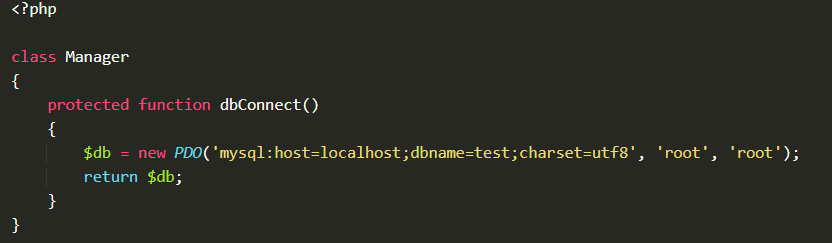
\*

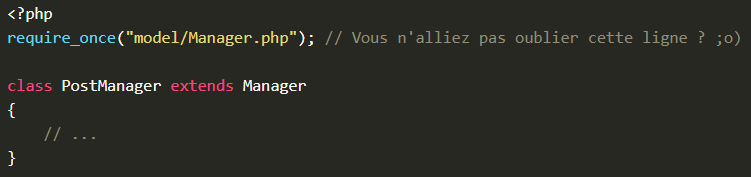
### L’héritage

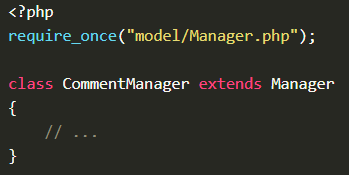
L’héritage se matérialise pas le mot extends qui signifie «hérite de la classe »

Les variables public peuvent s’utiliser en dehors de la classe, les variables private ne s’utilise seulement dans la classe, et protected c’est comme private sauf que les classe fille qui ont hértié de la classe peuvent utilisé la variable.

On va factorisé la connection à la base de donné grâce à l’héritage :





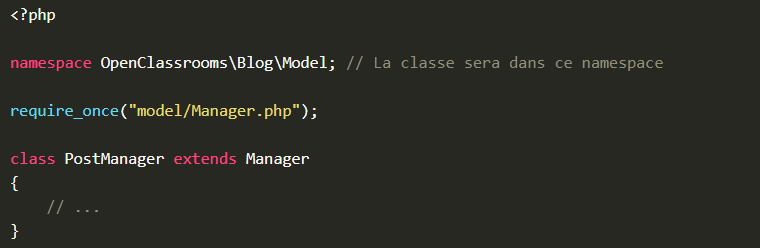


### Les namespaces

Leur rôle ? Éviter les collisions de noms de classes. Les namespaces ont cette forme :

* Entreprise\Projet\Section\SousSection\SousSousSection

Pour définir un namespace, rien de plus simple. On va déclarer un  namespace  juste avant la définition de la classe :



Cela a un impact : tous les fichiers qui font appel à cette classe doivent maintenant ajouter le namespace en préfixe. Voilà par exemple à quoi va ressembler la fonction  post()  du contrôleur :





Il est possible d’utiliser use afin d’éviter ça :

