Recherche & préparation au développement PHP

Adrien Lecomte

2026

Table des matières

1 Architecture MVC

1.1 Introduction

L'architecture **MVC** est un patron de conception (*design pattern* en anglais). Il a été créé par Trygve Reenskaug dans les années 70-80 afin d'organiser une interface graphique.

Aujourd'hui, l'architecture MVC est utilisée partout (Laravel, AngularJS, Spring, etc.). Les rôles des trois entités sont les suivants :

— **Modèle** : données ;
— **Vue** : interface utilisateur ;
— **Contrôleur** : gestion des événements et synchronisation.

1.2 Modèle

Le modèle contient les **données** qui vont être manipulées par le programme. C'est dans cette partie que l'on assure l'intégrité et la gestion des données. C'est ici que l'on va retrouver des algorithmes (recherche de mots, calculs, chiffrement, etc.). On y retrouve aussi des requêtes SQL par exemple. C'est ce qu'on appelle la **logique métier** de l'application.

1.3 Vue

La vue va s'occuper de l'affichage utilisateur. C'est-à-dire qu'elle va afficher les données qu'on lui donne. Cela peut être un formulaire, un bouton, la gestion du responsive, etc.

1.4 Contrôleur

Le contrôleur est l'intermédiaire entre le modèle et la vue. Il va gérer les échanges de données, les événements utilisateurs, etc.

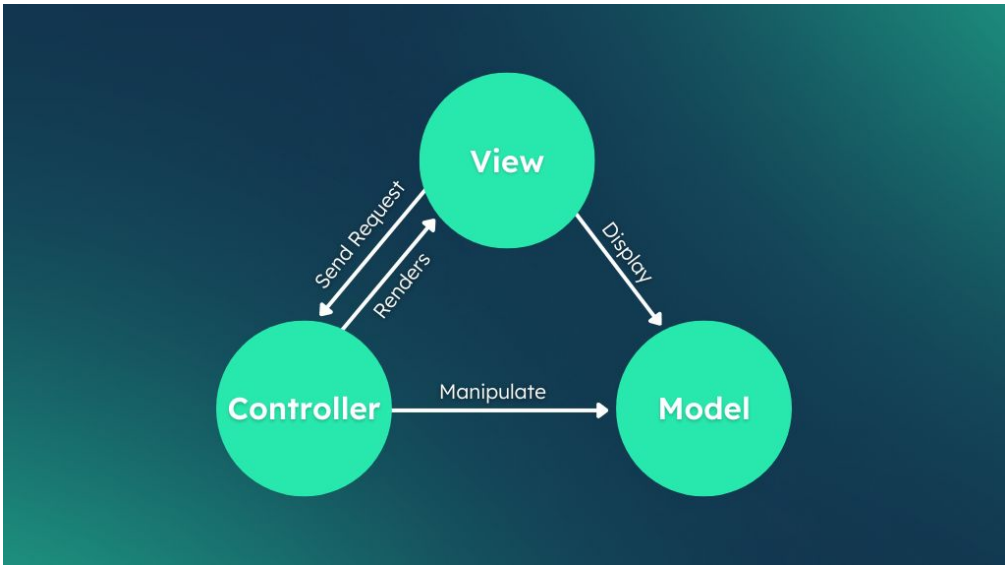

FIGURE 1 – Architecture MVC

1.5 Cheminement d'une requête HTTP

Le cheminement d'une requête HTTP se fait en plusieurs étapes. Dans ce cas précis, ce sera une requête POST pour s'authentifier.

1. L'utilisateur remplit le formulaire d'authentification (email et mot de passe) et clique sur le bouton **Se connecter**.
2. La **Vue** récupère le formulaire et l'envoie au contrôleur.
3. Le **Contrôleur** récupère le formulaire et l'envoie au modèle.
4. Le **Modèle** récupère le formulaire (il peut faire une analyse des données pour la sécurité de l'application) puis fait une requête SQL pour vérifier si l'utilisateur existe et si le mot de passe est correct.
5. Si l'utilisateur est correct, le modèle va créer une session utilisateur et envoyer l'ID de la session au contrôleur.
6. Le **Contrôleur** va stocker l'ID de la session dans un cookie (bien configuré), l'insérer dans la réponse de la requête HTTP, puis rediriger la vue vers la page d'accueil.

1.6 Pourquoi les requêtes SQL ne doivent-elles pas se trouver dans les vues ?

Il y a plusieurs raisons fondamentales pour lesquelles l'accès aux données doit rester exclusif au Modèle :

1. **La sécurité**
   - **Risque d'injection SQL :** Les vues sont destinées à l'affichage et concatènent souvent des variables. Si les requêtes y sont écrites directement, il est plus difficile de garantir qu'elles sont sécurisées (préparées).
   - **Exposition des données sensibles :** Si une erreur SQL survient dans une vue, le message d'erreur technique (révélant souvent le nom des tables ou des colonnes) risque de s'afficher directement à l'écran de l'utilisateur final.
2. **La maintenabilité**
   - **Principe DRY (Don't Repeat Yourself) :** Si une même requête est écrite dans plusieurs vues différentes, la moindre modification de la base de données (ex : renommage d'une table) obligera à modifier manuellement tous les fichiers de vues. En MVC, on ne modifie le code qu'une seule fois dans le Modèle.

2 Front Controller et Routage

2.1 Le Front Controller (Contrôleur Frontal)

Le **Front Controller** est un patron de conception (*design pattern*).. Il désigne un **point d'entrée unique** pour l'ensemble de l'application.

Concrètement, au lieu d'avoir un fichier PHP par page (ex : `accueil.php`, `contact.php`, `produit.php`), toutes les requêtes HTTP sont redirigées vers un seul fichier principal, généralement nommé `index.php`.

Son rôle est d'initialiser l'application. C'est le fichier principale qui :

— Charge la configuration globale (connexion à la base de données, chargement des classes, .env, etc)

— Filtre les requêtes

— Appelle le **Routeur** pour déterminer quelle page afficher.

2.2 Pourquoi centraliser les requêtes dans index.php ?

Centraliser le flux dans un fichier `index.php` a plusieurs avantages :

1. **Factorisation du code (DRY) :** On ne répète pas le code de connexion à la base de données dans chaque fichier. Tout est fait une seule fois au démarrage.
2. **Sécurité :** C'est le seul fichier exposé publiquement. Les autres fichiers PHP peuvent être stockés en dehors du dossier accessible par le navigateur, empêchant leur exécution directe.
3. **Gestion des URLs :** Cela permet d'utiliser des "URLs propres" (URL rewriting), plus agréables pour l'utilisateur (et le dev) et le référencement (SEO).

2.3 Le Routeur

Le rôle du **Routeur** est d'analyser l'URL demandée par l'utilisateur (l'URI) et de décider quel **Contrôleur** et quelle méthode appeler pour traiter la demande.

2.4 Exemple de routage simple en PHP

Voici une méthode basique pour mettre en place un routage sans utiliser de framework complexe. On utilise un paramètre GET dans l'URL pour déterminer la page.

```
<?php
$page = $_GET['page'] ?? 'accueil';

switch ($page) {
 case 'accueil':
 require_once 'controllers/HomeController.php';
 break;

 case 'contact':
 require_once 'controllers/ContactController.php';
 break;

 case 'login':
 require_once 'controllers/AuthController.php';
 break;

 default:
 require_once 'views/404.php';
 break;
}
?>
```

3 Accès aux données avec PDO

3.1 Introduction

PDO est un acronyme pour **PHP Data Object**. Il s'agit d'une extension définissant l'interface pour accéder à une base de données. PDO est disponible depuis PHP 5 et est recommendé parce qu'elle **orienté objet**.

3.2 Quelle est la différence entre query() et prepare() / execute() ?

3.2.1 Query

La fonction **PDO : :query** prépare et éxécute une requête SQL en un seul appel de fonction tout en retournant la reqête en objet.

3.2.2 Prepare

La fonction **PDO : :prepare** sert à préparer une requête SQL en utilisant sois des paramètres nommé (:name) ou des marqueurs (?) pour lesquels les valeurs réelles seront substituées lorsque la requête sera exécutée. Anoter que l'utilisation à la fois des paramètres nommés ainsi que les marqueurs est impossible dans un modèle de déclaration. La fonction renvoie un objet **PDOStatement**, si le serveur de base de données ne réussit pas à préparer la requête, la fonction renvoie **false** ou une exception PDOException.

3.2.3 Execute

La fonction **PDOStatement : :execute** éxécute une requête préparé (voir Prepare). Si la requête préparé inclut des marqueurs (?), on doit utiliser PDOStatement : :bindParam() et/ou PDOStatement : :bindValue() pour lier des variables ou des valeurs aux marqueurs.

3.2.4 Conclusion

Il est plus conseillé de préparer la requête puis de l'éxécuter, c'est plus performant et si les marqueurs (?) sont utilisé, cela bloque les injection SQL.

3.3 Pourquoi les requêtes préparées sont-elles indispensables dans une application web ?

Une application web a bessoin des requêtes préparées pour plusieurs raisons :
- **Sécurité** : Pour se protéger des injections SQL
- **Performance** : avec les requêtes préparées, on autorise le pilote à négocier coté client et/ou serveur avec le cache des requêtes et les meta-informations
- **DRY** : On évite aussi d'écrire plusieurs fois la même requête dans toute l'application

3.4 Quelle est la différence entre fetch() et fetchAll() ?

3.4.1 Fetch

La fonction **PDOStatement : :fetch** s'utilise après avoir éxécuté une requête. Elle permet de lire le résultat après l'éxécution et de passer à la ligne suivante si on rappelle la fonction après. Si il n'y a plus de ligne, la fonction renvoie **false**. La fonction dispose de plusieurs mode de lecture.

Source : [1]

3.4.2 FetchAll

La fonction **PDOStatement : :fetchAll** permet de récupérer toutes les lignes d'une requête exécutée sous la forme d'un tableau (*array*). Cela peut être très gourmand en ressources (mémoire vive), car l'intégralité du jeu de résultats est chargée en PHP. Il est fortement conseillé d'utiliser des clauses SQL (`LIMIT`, `WHERE`) pour réduire le nombre de résultats retournés.

Source : [2]

3.5 Gestion des erreurs lors de l'exécution d'une requête SQL avec PDO

Pour la gestion d'erreur, on utilise des **try** et des **catch** afin de récuperer les warnings et les erreurs. Cela permet de loe **logs** et de faire en sortte que l'application continue de fonctionner malgrés les erreurs.

4 Gestion des requêtes et couche Model

4.1 Rôle de la couche Model dans une architecture MVC

Le rôle de la couche model est d'avoir toute la logqie métier de l'application, c'est dans cette couche que l'on va retrouver les algorithmes, les requêtes SQL. C'est l'équivalent du cerveau dans le corp humain.

4.2 Pourquoi le contrôleur ne doit-il pas contenir de requêtes SQL ?

Le rôle du contrôleur est exclusivement de faire la liaison (*glue code*) entre la Vue et le Modèle. Il reçoit la demande de l'utilisateur, demande les données au Modèle, et choisit la Vue à afficher.

Si le contrôleur contenait des requêtes SQL, cela poserait plusieurs problèmes :

- **Violation du principe de responsabilité unique :** Le contrôleur doit gérer le flux de l'application (requêtes HTTP, redirections), pas la logique de données.
- **Duplication du code :** Si deux contrôleurs différents ont besoin des mêmes données (par exemple, la liste des utilisateurs), on serait obligé de copier-coller la requête SQL dans les deux fichiers.

4.3 Comment organiser les méthodes d'un Model pour interagir avec la base de données ?

Avoir un dossier **database** qui contient une classe parent qui contient les information de la base de données (connection, attribut, etc) puis utiliser le mot clés **extends** pour chaque entités. Cela permet d'éviter de faire plusieurs connexion pour chaque entités.

1. `https://www.php.net/manual/fr/pdostatement.fetch.php`
2. `https://www.php.net/manual/fr/pdostatement.fetchall.php`

4.4 Exemple

```
class Model {
 protected $db;

 public function __construct($databaseConnection) {
  $this->db = $databaseConnection;
 }
}

class User extends Model {
 protected $table = 'users';

 public function find($id) {
  // Logique SQL : SELECT * FROM users WHERE id = :id
 }

 public function getPremiumUsers() {
  // Logique SQL : SELECT * FROM users WHERE status = 'premium'
 }

 public function getPosts() {
  // Récupérer les articles liés à cet utilisateur
 }
}
```

5 Pattern Singleton

5.1 Qu'est-ce que le pattern Singleton ?

Le **Singleton** est un patron de conception (design pattern) de création qui garantit qu'une classe n'est instanciée qu'une seule fois au cours de l'exécution d'un script. Il fournit un point d'accès global à cette instance unique.

Pour implémenter ce pattern en PHP, trois éléments sont indispensables :

— Une **propriété statique privée** qui stockera l'instance unique.

— Un **constructeur privé** pour empêcher la création d'objets via le mot-clé `new` depuis l'extérieur de la classe.

— Une **méthode statique publique** (généralement nommée `getInstance()`) qui crée l'instance si elle n'existe pas, ou retourne l'instance existante.

5.2 Pourquoi utilise-t-on souvent le Singleton pour la connexion à la BDD ?

L'usage du Singleton est particulièrement répandu pour les connexions aux bases de données pour les raisons suivantes :

1. **Économie de ressources :** L'ouverture d'une connexion SQL est une opération lourde. Le Singleton évite d'ouvrir plusieurs connexions identiques lors d'une même requête HTTP, économisant ainsi la mémoire et les sockets du serveur.
2. **Unicité de l'état :** Il garantit que toutes les parties de l'application utilisent la même instance de connexion, ce qui est crucial pour la gestion des transactions (un `COMMIT` ou `ROLLBACK` doit s'appliquer sur la même connexion).
3. **Accès simplifié :** Il permet d'accéder à la base de données depuis n'importe quel Model sans avoir à passer l'objet de connexion manuellement de constructeur en constructeur.

5.3 Avantage et limite du pattern Singleton

Avantage : Contrôle strict de l'accès. Le Singleton permet de s'assurer que l'instance est initialisée uniquement au moment où on en a besoin (lazy loading) et qu'aucune autre instance ne viendra surcharger le système inutilement.

Limite : Difficulté des tests unitaires. Le Singleton introduit un état global dans l'application. Cela crée un couplage fort entre les classes, rendant difficile l'isolation d'un composant pour le tester sans dépendre réellement de la base de données.