

Le modèle MVC

Table des matières

I. Contexte	3
II. Définition du MVC	3
III. Exercice : Appliquez la notion	4
IV. Principe de base	4
V. Exercice : Appliquez la notion	9
VI. Organisation des trois couches	10
VII. Exercice : Appliquez la notion	19
VIII. Analyse de l'architecture	19
IX. Exercice : Appliquez la notion	21
X. Auto-évaluation	22
A. Exercice final	22
B. Exercice : Défi	23
Solutions des exercices	24

I. Contexte

Durée : 1 h 30

Environnement de travail : XAMPP

Pré-requis : Connaissances avancées en HTML et PHP objet, connaissances de base en SQL

Contexte

La création d'une application web va forcément amener à s'interroger sur la façon d'organiser le code source. Or, cette organisation est primordiale pour garantir une maintenabilité du projet sur les moyen et long termes.

- Comment éviter la duplication de code et donc gagner en efficacité ?
- Comment minimiser les changements à effectuer si l'interface graphique du site web doit évoluer ?
- Quelle organisation du code choisir pour faciliter la collaboration entre plusieurs développeurs sur un même projet ?

L'architecture MVC, très populaire pour les applications web, permet de répondre efficacement à ces questions en proposant un modèle robuste et éprouvé.

II. Définition du MVC

Objectif

- Comprendre la définition de l'architecture MVC

Mise en situation

Lorsqu'on s'intéresse à la conception d'une application web, on va rapidement trouver le terme MVC. Mais, concrètement, d'où vient ce terme et que signifie-t-il ? Quel est le concept général de cette architecture ?

Définition du MVC

Une architecture logicielle décrit la manière de concevoir et d'organiser le code source d'une application. L'architecture MVC a été conçu par Trygve Reenskaug en 1978 et fait aujourd'hui partie des modèles les plus utilisés pour la conception des applications web. Cette architecture est divisée en trois couches distinctes, dans lesquelles sera organisé le code source. MVC correspond à l'acronyme du nom de chacune des trois couches :

- **Modèle**
- **Vue**
- **Contrôleur**

Remarque

Dans la littérature informatique, on trouvera souvent les noms des couches en anglais : *Model, View, Controller*.

Chaque couche est chargée d'un rôle bien précis.

Modèle :

- Il contient les données et la logique applicative.
- Exemple : les données avec leur validation et les calculs qui peuvent être associés.

Vue :

- Elle correspond à l'interface graphique, donc la partie visible par l'utilisateur de l'application web.
- Exemple : le code HTML des pages et des formulaires.

Contrôleur :

- Il est en charge du traitement des actions de l'utilisateur. Il sert également à faire le lien entre le modèle et la vue.
- Exemple : le code appelé directement lorsqu'un utilisateur demande l'affichage d'une page web et qui va devoir faire appel au modèle et à la vue pour retourner le résultat à afficher.

Cette division correspond très bien aux besoins actuels des applications web modernes, ce qui explique la popularité de cette architecture.

Syntaxe	À retenir
<ul style="list-style-type: none"> • L'architecture MVC propose une organisation du code source divisée en trois couches nommées respectivement : Modèle, Vue et Contrôleur. • Le modèle contient la logique applicative, la vue correspond à l'interface graphique et le contrôleur traite les actions de l'utilisateur en faisant le lien avec les deux autres couches. 	

Complément
Modèle-vue-contrôleur ¹

Exercice : Appliquez la notion

[solution n°1 p.25]

Exercice

L'architecture MVC divise l'application en combien de couches ?

- ☐ 2
- ☐ 3
- ☐ 4

Exercice

Que signifie le V dans l'acronyme MVC ?

IV. Principe de base

¹ <https://fr.wikipedia.org/wiki/Mod%C3%A8le-vue-contr%C3%B4leur>

Objectif

- Mettre en pratique l'architecture MVC sur un site avec une seule page

Mise en situation

Connaître la définition théorique de l'architecture MVC est intéressant. Mais le plus utile au quotidien est de savoir la mettre en pratique lors de la conception d'un site web.

C'est ce que nous allons faire ici, et nous aurons besoin pour cela d'installer XAMPP afin de disposer d'un serveur Apache, de PHP 7 et d'une base de données MySQL.

Utiliser un cas pratique

Nous proposons de concevoir un *book* en ligne pour un photographe qui va nous servir de base pour comprendre et appliquer l'architecture MVC. Ce site a pour objectif de présenter les photographies réalisées par l'artiste afin de démarcher des galeries d'art.

La page d'accueil du site doit afficher le nom du *book*, les photos et le copyright. Les informations sur les photos sont stockées dans une base de données MySQL.

Ci-dessous, la structure de la table `photo` utilisée avec quelques données :

Nom de la colonne	Type	Null	Extra
id	INT(11)	Non	AUTO_INCREMENT
fichier	VARCHAR(100)	Non	
titre	VARCHAR(100)	Non	

Le code source de la page d'accueil est rangé dans le fichier `index.php` présenté ci-dessous.

Exemple index.php

```
1 <!DOCTYPE html>
2 <html lang="fr">
3   <head>
4     <title>Mon book</title>
5     <meta charset="utf-8">
6     <meta name="viewport" content="width=device-width, initial-scale=1">
7   </head>
8   <body>
9     <?php
10       try {
11         $pdo = new PDO('mysql:host=localhost;dbname=book;charset=utf8', 'root',
12 'root');
13       } catch (PDOException $e) {
14         exit('Erreur : '.$e->getMessage());
15       }
16       $stmt = $pdo->query('SELECT * FROM photo');
17       $photos = [];
18       while ($photo = $stmt->fetchObject()) {
19         $photos[] = $photo;
20       }
21     <?>
22     <header>
23       <h1>Mon book</h1>
```

```

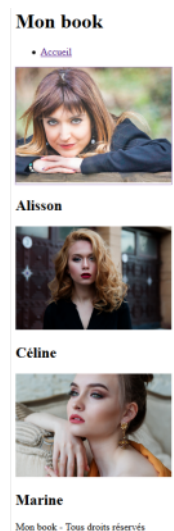
23     </header>
24     <section>
25         <nav>
26             <ul>
27                 <li><a href="/">Accueil</a></li>
28             </ul>
29         </nav>
30         <article>
31             <?php foreach ($photos as $photo): ?>
32                 
33                 <h2><?= $photo->titre ?></h2>
34             <?php endforeach; ?>
35         </article>
36     </section>
37     <footer>
38         <p>Mon book - Tous droits réservés</p>
39     </footer>
40 </body>
41 </html>

```

Le résultat graphique affiché dans le navigateur correspond à la capture d'écran ci-dessous.

En l'état, la page est parfaitement fonctionnelle. Pour autant, regrouper l'ensemble du code source dans un seul fichier est peu lisible. Nous allons vite atteindre les limites de cette organisation lorsqu'il s'agira de créer de nouvelles pages.

Une bonne pratique consiste donc à séparer au maximum le code PHP du code HTML. C'est d'ailleurs ce concept que l'on retrouve dans l'architecture MVC avec la couche vue, destinée uniquement à la présentation.



Isoler la présentation

L'idée consiste donc à isoler la présentation, en grande majorité en HTML, du reste du code source du site. Pour cela, nous allons conserver l'ensemble du code PHP dans le fichier `index.php`, à l'exception de la boucle `foreach` qui est utilisée uniquement pour de l'affichage. Puis nous allons déplacer le code HTML et la boucle `foreach` dans un fichier séparé qui sera inclus depuis `index.php`. Voyons ci-dessous le code du fichier PHP qui a gagné en lisibilité :

Exemple index.php

```

1 <?php
2 try {
3     $pdo = new PDO('mysql:host=localhost;dbname=book;charset=utf8', 'root', 'root');
4 } catch (PDOException $e) {
5     exit('Erreur : '.$e->getMessage());
6 }

```

```

7 $stmt = $pdo->query('SELECT * FROM photo');
8 $photos = [];
9 while ($photo = $stmt->fetchObject()) {
10     $photos[] = $photo;
11 }
12 require_once('vues/liste-photos.php');

```

Remarque

La fonction PHP `require_once()`¹ permet de charger de façon unique le code source présent dans un autre fichier. À l'exécution, le résultat est donc similaire à un copier-coller du contenu du fichier `liste-photos.php` à la fin du fichier `index.php`.

Puis le code chargé de l'affichage de la page d'accueil a été déplacé dans le fichier `liste-photos.php`, lui-même rangé dans un dossier `vues`.

Exemple `vues/liste-photos.php`

```

1 <!DOCTYPE html>
2 <html lang="fr">
3 <head>
4     <title>Mon book</title>
5     <meta charset="utf-8">
6     <meta name="viewport" content="width=device-width, initial-scale=1">
7 </head>
8 <body>
9 <header>
10     <h1>Mon book</h1>
11 </header>
12 <section>
13     <nav>
14         <ul>
15             <li><a href="/">Accueil</a></li>
16         </ul>
17     </nav>
18     <article>
19         <?php foreach ($photos as $photo): ?>
20             
21             <h2><?=$photo->titre ?></h2>
22         <?php endforeach; ?>
23     </article>
24 </section>
25 <footer>
26     <p>Mon book - Tous droits réservés</p>
27 </footer>
28 </body>
29 </html>

```

1 <https://www.php.net/manual/fr/function.require-once.php>

Finalement, pour le visiteur, l'aspect esthétique du site n'a pas évolué. En revanche, la séparation du code a permis de rendre celui-ci plus lisible et maintenable.

Pour autant, nous pourrions encore améliorer cette architecture. En effet, le fichier `index.php` contient l'accès à la base de données et une requête permettant de récupérer les données à afficher. Or, la définition de l'architecture MVC nous indique que le contrôleur (ici `index.php`) devrait être uniquement chargé de traiter les actions de l'utilisateur. Il faudrait donc déplacer dans une nouvelle couche les fonctionnalités relatives à l'accès aux données, c'est-à-dire séparer la logique applicative.

Isoler la logique applicative

Déplaçons le code du fichier `index.php` chargé de la connexion à la base de données et la requête à la base de données dans un fichier séparé. Afin de bénéficier des avantages de la programmation orientée objet, nous allons construire une classe `Photos`.

Le constructeur sera chargé de la connexion à la base de données qui, une fois initialisée, sera stockée dans un attribut privé. Puis une méthode `listerPhotos()` aura pour fonction de récupérer les photos dans la base de données et de les retourner sous forme de tableau.

Ci-dessous le code source de la classe `Photos` avec les deux méthodes présentées :

Exemple modeles/Photos.php

```

1 <?php
2
3 class Photos
4 {
5     private $pdo = null;
6
7     public function __construct()
8     {
9         try {
10             $this->pdo = new PDO('mysql:host=localhost;dbname=book;charset=utf8', 'root',
11 'root');
12         } catch (PDOException $e) {
13             exit('Erreur : '.$e->getMessage());
14         }
15
16     public function listerPhotos()
17     {
18         if (!is_null($this->pdo)) {
19             $stmt = $this->pdo->query('SELECT * FROM photo');
20         }
21         $photos = [];
22         while ($photo = $stmt->fetchObject()) {
23             $photos[] = $photo;
24         }
25
26         return $photos;
27     }
28 }
```

Le fichier `index.php` s'allège donc une fois de plus pour ne contenir que le chargement des fichiers `modeles/Photos.php` et `vues/liste-photos.php`, ainsi que l'initialisation d'un objet `Photos` et l'appel à la méthode `listerPhotos()`.

Exemple **index.php**

```

1 <?php
2 require_once('modeles/Photos.php');
3 $photos = new Photos();
4 $photos = $photos->listPhotos();
5 require_once('vues/Liste-photos.php');

```

Nous venons de structurer notre *book* en ligne avec l'architecture MVC.

- Le fichier `modeles/Photos.php` correspond au modèle et regroupe la logique métier.
- Le fichier `vues/liste-photos.php` contient uniquement le code chargé de l'interface graphique.
- Le fichier `index.php` correspond, lui, au contrôleur et ne fait que répondre aux actions de l'utilisateur en faisant appel au modèle et à la vue.

Le principe de base de l'architecture MVC est donc relativement simple à mettre en pratique. Cependant, dès que l'application va s'étoffer, avec notamment plusieurs pages à gérer, il faudra structurer encore plus le code de chacune des trois couches.

Syntaxe **À retenir**

- Le code chargé d'accéder à la base de données et de récupérer les données doit être stocké dans une classe dédiée correspondant à la couche modèle.
- Le code HTML et le code PHP chargé uniquement de l'affichage sont regroupés dans un fichier qui correspond à la couche vue.
- Le contrôleur est identifié par le fichier appelé par l'internaute, qui va traiter son action en faisant appel au modèle et à la vue.

V. Exercice : Appliquez la notion

Pour réaliser cet exercice, vous pouvez travailler sur l'environnement de travail :


Question

[solution n°2 p.25]

À vous de jouer ! Vous venez d'écrire la première page de votre blog, mais vous trouvez qu'elle n'est pas très lisible. Séparez-la donc en plusieurs fichiers :

- un fichier `models/Posts.php` qui va s'occuper des accès à la base de données et qui sera notre modèle,
- un fichier `views/posts-list.php` qui va s'occuper de l'affichage et qui sera notre vue,
- un fichier `index.php` qui va s'occuper de la glue entre le modèle et la vue et qui sera notre contrôleur.

```

1 <?php
2     try {
3         $pdo = new PDO('mysql:host=localhost;dbname=blog;charset=utf8', 'root', 'root');
4     } catch (PDOException $e) {
5         exit('Erreur : '.$e->getMessage());
6     }
7     $stmt = $pdo->query('SELECT title, image FROM post');

```

1 <https://repl.it/>

```

8      $posts = [];
9      while ($post = $stmt->fetchObject()) {
10         $posts[] = $post;
11     }
12 ?>
13 <!DOCTYPE html>
14 <html lang="fr">
15     <head>
16         <title>Mon blog</title>
17         <meta charset="utf-8">
18         <meta name="viewport" content="width=device-width, initial-scale=1">
19     </head>
20     <body>
21         <header>
22             <h1>Mon blog</h1>
23         </header>
24         <section>
25             <nav>
26                 <ul>
27                     <li><a href="/">Accueil</a></li>
28                 </ul>
29             </nav>
30             <article>
31                 <?php foreach ($posts as $post): ?>
32                     
33                     <h2><?=$post->title ?></h2>
34                 <?php endforeach; ?>
35             </article>
36         </section>
37         <footer>
38             <p>Mon blog - Tous droits réservés</p>
39         </footer>
40     </body>
41 </html>

```

VI. Organisation des trois couches

Objectifs

- Appliquer l'architecture MVC sur un site avec plusieurs pages
- Organiser le code à l'intérieur des différentes couches

Mise en situation

Une fois que le concept de base de l'architecture MVC a été compris, il convient de l'appliquer sur une application web plus importante. Pour cela, nous allons faire évoluer notre *book* en ligne. Il sera alors nécessaire de s'interroger sur l'organisation du code au sein même de chaque couche. Nous verrons alors que de bonnes pratiques existent et permettent d'optimiser le code et sa maintenabilité.

Organiser les vues

Commençons par faire évoluer notre *book* en ligne de photographe. Pour cela, faisons en sorte que le visiteur qui clique sur une photo de la page d'accueil soit dirigé vers une nouvelle page, qui présente la photo en grand format avec le titre dessous.

L'aspect graphique de cette nouvelle page ressemblerait donc au visuel ci-dessous.

Mon book

• [Accueil](#)

Pour afficher cette page, l'URL serait de la forme `photo.php?id=2` dans lequel le chiffre (ici, 2) correspondrait à l'identifiant de la photo à afficher.

Il faut donc faire évoluer le modèle et la classe `Photo` précédente en ajoutant une méthode `afficherPhoto($id)`. Celle-ci va récupérer les informations d'une photo à partir de son identifiant, et les retourner.

En effet, il serait inutile et moins performant de récupérer toutes les photos avec la méthode `listPhotos()` existante simplement pour afficher l'une d'entre elles.



Alisson

Mon book - Tous droits réservés

Ci-dessous, la nouvelle version de la classe `Photos` qui comporte la nouvelle méthode `afficherPhoto($id)`.

Exemple	modeles/Photos.php
1	<code><?php</code>
2	
3	<code>class Photos</code>
4	<code>{</code>
5	<code> private \$pdo = null;</code>
6	
7	<code> public function __construct()</code>
8	<code> {</code>
9	<code> try {</code>
10	<code> \$this->pdo = new PDO('mysql:host=localhost;dbname=book;charset=utf8', 'root',</code>
11	<code>'root');</code>
12	<code> } catch (PDOException \$e) {</code>
13	<code> exit('Erreur : '.\$e->getMessage());</code>
14	<code> }</code>
15	<code> }</code>
16	<code> public function listPhotos()</code>
17	<code> {</code>
18	<code> if (!is_null(\$this->pdo)) {</code>
19	<code> \$stmt = \$this->pdo->query('SELECT * FROM photo');</code>
20	<code> }</code>
21	<code> \$photos = [];</code>
22	<code> while (\$photo = \$stmt->fetchObject()) {</code>
23	<code> \$photos[] = \$photo;</code>
24	<code> }</code>
25	
26	<code> return \$photos;</code>
27	<code> }</code>
28	
29	<code> public function afficherPhoto(\$id)</code>
30	<code> {</code>
31	<code> if (!is_null(\$this->pdo)) {</code>
32	<code> \$stmt = \$this->pdo->prepare('SELECT * FROM photo WHERE id = ?');</code>
33	<code> }</code>

```

34     $photo = null;
35     if ($stmt->execute([$id])) {
36         $photo = $stmt->fetchObject();
37         if (!is_object($photo)) {
38             $photo = null;
39         }
40     }
41
42     return $photo;
43 }
44 }

```

Il convient ensuite de créer un nouveau fichier `photo.php` à la racine du site, en se basant sur le code du fichier `index.php`. Il faut alors remplacer l'appel à la méthode `listPhotos()` par `afficherPhoto($_GET['id'])`. Avant d'effectuer cet appel, il convient de s'assurer que la variable `id` passée en GET existe et correspond bien à un entier numérique. Dernière modification : il faut appeler une nouvelle vue que nous allons nommer `vue/affiche-photo.php`.

Exemple photo.php

```

1 <?php
2 require_once('modeles/Photos.php');
3 $photos = new Photos();
4 $photo = null;
5 if (isset($_GET['id']) && is_numeric($_GET['id'])) {
6     $photo = $photos->afficherPhoto($_GET['id']);
7 }
8 require_once('vues/affiche-photo.php');

```

Si nous conservons la logique actuelle, nous allons copier le contenu (essentiellement HTML) du fichier `vues/liste-photos.php` précédent pour le coller dans `vues/affiche-photo.php` et l'adapter. Mais, si nous procédons ainsi, nous allons nous retrouver avec ces deux fichiers qui auront beaucoup de code dupliqué, comme la structure HTML, le header, le menu et le footer.

Si par la suite il nous est demandé de modifier, par exemple le footer pour ajouter l'année courante, nous allons devoir modifier les deux fichiers. Puis, si le site évolue et comporte une centaine de pages, il va falloir répéter cette modification autant de fois, ce qui va prendre du temps, sans compter le risque d'oubli ou d'erreur. Ce qu'il faudrait mettre en place, c'est un nouveau fichier qui contient la structure HTML commune à toutes les pages du site. C'est ce que l'on appelle le layout (gabarit de page) : il va justement nous permettre d'éviter cette duplication de code.

Ci-dessous le fichier `vues/layout.php` dans lequel la structure HTML a été ajoutée. Le titre et le contenu du corps de la page sont remplacés par des variables qui seront définies dans les templates correspondant aux pages.

Exemple vues/layout.php

```

1 <!DOCTYPE html>
2 <html lang="fr">
3 <head>
4     <title><?= $titre ?></title>
5     <meta charset="utf-8">
6     <meta name="viewport" content="width=device-width, initial-scale=1">
7 </head>
8 <body>
9 <header>
10     <h1>Mon book</h1>

```

```

11 </header>
12 <section>
13     <nav>
14         <ul>
15             <li><a href="/">Accueil</a></li>
16         </ul>
17     </nav>
18     <?= $contenu ?>
19 </section>
20 <footer>
21     <p>Mon book - Tous droits réservés</p>
22 </footer>
23 </body>
24 </html>

```

Il faut donc modifier le template `vues/liste-photos.php` afin de supprimer la structure HTML de la page et conserver uniquement le contenu de la zone centrale (balise `<article>`). Le contenu a légèrement évolué pour ajouter le lien sur la photo afin de visualiser celle-ci en grand format. Puis, il faut initialiser les variables `$titre` et `$contenu`.

Exemple `vues/liste-photos.php`

```

1 <?php
2 $titre = 'Mon book';
3 ob_start();
4 ?>
5     <article>
6         <?php foreach ($photos as $photo): ?>
7             <a href="photo.php?id=<?= $photo->id ?>">
8                 
9             </a>
10            <h2><?= $photo->titre ?></h2>
11        <?php endforeach; ?>
12    </article>
13 <?php
14 $contenu = ob_get_clean();
15 require_once('layout.php');

```

Remarque

La fonction `ob_start()`¹ permet de temporiser la sortie en la plaçant dans une mémoire tampon. Puis la fonction `ob_get_clean()`² retourne le contenu du tampon qui peut être récupéré dans une variable et l'efface. Ce processus permet de mettre le code HTML généré dans la variable `contenu` afin de l'afficher dans l'ordre souhaité, c'est-à-dire au milieu des balises `<section>` du layout.

Il ne reste plus qu'à appliquer le même principe pour la création du template `vues/affiche-photo.php`, dont le code est détaillé ci-dessous.

1 <https://www.php.net/manual/fr/function.ob-start.php>

2 <https://www.php.net/manual/fr/function.ob-get-clean.php>

Exemple vues/affiche-photo.php

```

1 <?php
2 $titre = 'Une photo de mon book';
3 if (is_null($photo)):
4     $contenu = "Cette photo n'existe pas.";
5 else:
6     ob_start();
7 ?>
8     <article>
9         
10        <h2><?= $photo->titre ?></h2>
11    </article>
12 <?php
13     $contenu = ob_get_clean();
14 endif;
15 require_once('layout.php');
```

Finalement, cette évolution a amené une organisation de la couche vue avec un layout et plusieurs templates. Cela nous permet une structure de page HTML commune et de bénéficier pour chaque page d'un fichier dédié pour le contenu.

Mais les deux contrôleurs que sont `index.php` et `photo.php` ont beaucoup de similitudes au niveau du code. Il est temps d'améliorer l'organisation de la couche contrôleur.

Organiser les contrôleurs

À ce stade, nous avons deux contrôleurs distincts qui sont appelés directement par le visiteur via l'URL. Or, cela va poser rapidement plusieurs problèmes :

- `index.php`
- `photo.php`

Lorsqu'on a plusieurs contrôleurs appelés directement via l'URL, il est difficile de renommer ces fichiers, car cela va avoir un impact sur tous les liens du site, son référencement, ou encore les éventuels favoris qu'auraient pu enregistrer les visiteurs.

De plus, c'est un besoin courant que d'intégrer une bibliothèque pour la rendre accessible partout dans le code de l'application. C'est par exemple le cas si nous souhaitons intégrer une bibliothèque chargée de l'écriture des logs : il faudra alors répéter son inclusion dans nos deux contrôleurs.

Or, quand le nombre de pages de notre *book* en ligne va augmenter, le nombre de contrôleurs va également suivre cette hausse et accentuer encore plus les deux problématiques soulevées.

Une bonne pratique consiste donc à créer un seul contrôleur frontal (`index.php`) qui va servir de point d'entrée unique pour les actions de l'utilisateur. Ce contrôleur peut alors déléguer le traitement à un contrôleur secondaire, voire plusieurs dans le cas d'une application web plus imposante.

Pour appliquer ce principe, nous allons créer une classe `Contrôleur`. Une méthode `listPhotos()` va être chargée de répondre à la demande d'affichage de la page d'accueil et une méthode `afficherPhoto()` va permettre l'affichage d'une seule photo en grand format.

Finalement, cela revient en grande partie à déplacer le code des fichiers `index.php` et `photo.php` dans cette nouvelle classe.

Exemple controleurs/Controleur.php

```

1 <?php
2
3 class Contrôleur
4 {
5     public function listerPhotos()
6     {
7         $photos = new Photos();
8         $photos = $photos->listerPhotos();
9         require_once('vues/liste-photos.php');
10    }
11
12    public function afficherPhoto()
13    {
14        $photos = new Photos();
15        if (isset($_GET['id']) && is_numeric($_GET['id'])) {
16            $photo = $photos->afficherPhoto($_GET['id']);
17        }
18        require_once('vues/affiche-photo.php');
19    }
20 }

```

Le code source du contrôleur frontal `index.php` est alors largement simplifié en faisant appel à la classe `Contrôleur`.

Exemple index.php

```

1 <?php
2 require_once('contrôleurs/Contrôleur.php');
3 require_once('modeles/Photos.php');
4 $contrôleur = new Contrôleur();
5 if (isset($_GET['page']) && 'photo' === $_GET['page']) {
6     $contrôleur->afficherPhoto();
7 } else {
8     $contrôleur->listerPhotos();
9 }

```

La réorganisation de la couche contrôleur nous a conduit à bénéficier d'un point d'entrée unique à notre application, appelé contrôleur frontal. Les actions de l'utilisateur sont ensuite déléguées à un contrôleur secondaire (ou plusieurs si la classe disposait de trop de lignes de code). Il ne reste plus qu'à optimiser la partie modèle.

Organiser les modèles

En l'état actuel, la classe `modeles/Photos.php` est assez lisible. Cependant, dès que les fonctionnalités du *book* en ligne vont évoluer, les méthodes vont se multiplier et risquer de nuire à la lisibilité. De plus, une méthode retourne un ensemble de photos, alors que l'autre ne retourne qu'une seule photo, ce qui n'est pas très cohérent.

Dans de nombreux frameworks MVC, on retrouve une pratique courante qui vise à créer, pour chaque table de la base de données, deux classes. Une qui représente un tuple (une ligne) de la base de données et qui aura donc le nom des colonnes comme propriétés, et une qui représente plusieurs tuples et qui s'occupe donc de gérer une collection d'objets de la première classe. Ainsi, suivant que le code est destiné à s'appliquer sur un tuple ou plusieurs, il sera placé dans l'une ou l'autre classe.

Appliquons cette bonne pratique à notre projet actuel en créant une classe `Photo` qui va représenter un tuple, et une classe `Photos` qui va représenter une collection de photos. Il est prévisible que ces deux classes auront des besoins communs, comme l'accès à la base de données.

Toujours dans le but d'éviter la duplication de code, nous allons définir un trait nommé `Modele` que nous allons utiliser dans les deux classes indiquées précédemment.

Remarque

Un trait est un morceau de code que l'on peut réutiliser dans plusieurs classes. Le fait d'utiliser le mot-clé `use` dans une classe pour utiliser un trait revient schématiquement à copier le code du trait dans l'endroit précis de la classe. Plus de détails sont précisés dans la documentation officielle de PHP¹.

Ci-dessous, la définition de ce trait :

Exemple modeles/Modele.php

```
1 <?php
2
3 trait Modele
4 {
5     private $pdo = null;
6
7     public function __construct()
8     {
9         try {
10             $this->pdo = new PDO('mysql:host=localhost;dbname=book;charset=utf8', 'root',
11 'root');
12         } catch (PDOException $e) {
13             exit('Erreur : '.$e->getMessage());
14         }
15 }
```

Définissons maintenant la classe `Photo` qui représente un tuple de la base de données avec le report de la méthode `afficherPhoto($id)`.

Exemple modeles/Photo.php

```
1 <?php
2
3 class Photo
4 {
5     use Modele;
6
7     private $id;
8
9     private $fichier;
10
11     private $titre;
12
13     public function afficherPhoto($id)
14     {
15         if (!is_null($this->pdo)) {
16             $stmt = $this->pdo->prepare('SELECT * FROM photo WHERE id = ?');
17         }
18         $photo = null;
19         if ($stmt->execute([$id])) {
```

¹ <https://www.php.net/manual/fr/language.oop5.traits.php>


```
20         $photo = $stmt->fetchObject('Photo');
21         if(!is_object($photo)) {
22             $photo = null;
23         }
24     }
25
26     return $photo;
27 }
28
29 public function getId()
30 {
31     return $this->id;
32 }
33
34 public function getFichier()
35 {
36     return $this->fichier;
37 }
38
39 public function getTitre()
40 {
41     return $this->titre;
42 }
43 }
```

Il est important de noter que la méthode `afficherPhoto($id)` retourne maintenant un objet de `Photo`.
La classe `Photos` avec la méthode `listerPhotos()` retourne elle une collection d'objets de `Photo`.

Exemple **modeles/Photos.php**

```
1 <?php
2
3 class Photos
4 {
5     use Modele;
6
7     public function listerPhotos()
8     {
9         if (!is_null($this->pdo)) {
10             $stmt = $this->pdo->query('SELECT * FROM photo');
11         }
12         $photos = [];
13         while ($photo = $stmt->fetchObject('Photo')) {
14             $photos[] = $photo;
15         }
16
17         return $photos;
18     }
19 }
```

Il reste à adapter la classe `Contrôleur` pour utiliser la nouvelle classe `Photo`, comme présenté ci-dessous.

Exemple controleurs/Controleur.php

```

1 <?php
2
3 class Contrôleur
4 {
5     public function listerPhotos()
6     {
7         $photos = new Photos();
8         $photos = $photos->listerPhotos();
9         require_once('vues/liste-photos.php');
10    }
11
12    public function afficherPhoto()
13    {
14        $photo = new Photo();
15        if (isset($_GET['id']) && is_numeric($_GET['id'])) {
16            $photo = $photo->afficherPhoto($_GET['id']);
17        }
18        require_once('vues/affiche-photo.php');
19    }
20 }

```

Puis à charger le trait `Modele` et les classes `Photos` et `Photo` dans le front contrôleur de l'application, soit le fichier `index.php`.

Exemple index.php

```

1 <?php
2 require_once('contrôleurs/Contrôleur.php');
3 require_once('modeles/Modele.php');
4 require_once('modeles/Photo.php');
5 require_once('modeles/Photos.php');
6 $contrôleur = new Contrôleur();
7 if (isset($_GET['page']) && 'photo' === $_GET['page']) {
8     $contrôleur->afficherPhoto();
9 } else {
10    $contrôleur->listerPhotos();
11 }

```

Remarque

À noter qu'il est possible de mettre en place un système de chargement automatique des classes afin d'éviter de multiplier les appels à la fonction `require_once()`¹, mais cela dépasse le cadre de l'application du modèle MVC. Plus d'informations dans la documentation officielle de PHP : <https://www.php.net/manual/fr/language.oop5.autoload.php>.

Syntaxe À retenir

- La couche vue s'organise généralement avec un layout (gabarit de la page) et plusieurs templates.
- Pour la partie contrôleur, on utilise un contrôleur frontal (souvent `index.php`) qui fait office de point d'entrée unique aux actions de l'utilisateur et qui délèguera les traitements à un ou plusieurs contrôleurs secondaires.

¹ <https://www.php.net/manual/fr/function.require-once.php>

- Une bonne pratique pour la couche modèle est de créer deux classes pour chaque table de la base de données : une qui correspond à un tuple (une ligne) et une autre à un ensemble de tuples.

VII. Exercice : Appliquez la notion

Pour réaliser cet exercice, vous pouvez travailler sur l'environnement de travail :



Question

[solution n°3 p.26]

Continuez à segmenter votre code en ajoutant la possibilité de voir le détail d'un article. Pour cela, vous aurez besoin de :

- un contrôleur frontal `index.php`
- un contrôleur `controllers/Controller.php` pour voir un article et lister des articles
- un trait générique `models/Model.php` contenant l'accès à la base de données
- deux modèles `models/Posts.php` et `models/Post.php` pour récupérer respectivement une liste d'articles et un unique article
- deux templates `views/posts-list.php` et `views/post-show.php` pour afficher respectivement une liste d'articles et un unique article
- un template global `views/layout.php` contenant l'ossature de vos pages HTML

VIII. Analyse de l'architecture

Objectifs

- Comprendre le schéma de l'architecture MVC
- Approfondir les bonnes pratiques, les avantages et les limites de cette architecture

Mise en situation

Après la mise en pratique de l'architecture MVC, il convient d'étudier son architecture, puis de s'interroger sur les différents cas que l'on peut rencontrer, et les bonnes pratiques à appliquer pour les traiter au mieux.

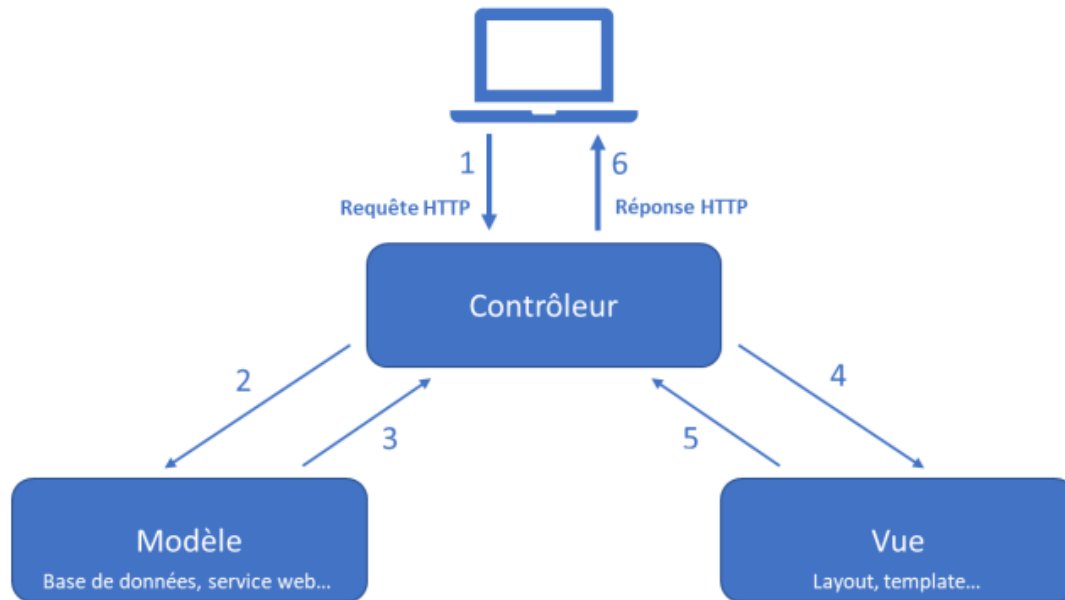
Comme toute architecture, MVC répond à des problématiques identifiées et apporte donc son lot d'avantages, mais potentiellement son lot d'inconvénients si elle n'est pas utilisée dans le bon contexte.

1 <https://repl.it/>

Schéma de fonctionnement

Ci-dessous, une vision schématique de l'architecture MVC telle que nous avons pu la mettre en pratique dans le chapitre précédent.

Schéma de l'architecture MVC



1. L'internaute demande la consultation d'une page web via une requête HTTP.
2. Le contrôleur réceptionne la demande et interroge le modèle pour obtenir les données nécessaires.
3. Le contrôleur récupère les données issues du modèle.
4. Le contrôleur appelle la vue correspondante en lui fournissant les données préalablement obtenues.
5. La vue retourne le résultat formaté au contrôleur.
6. Le contrôleur répond à l'internaute en lui fournissant le résultat correspondant à sa demande.

Bonnes pratiques

Une erreur courante dans l'application de l'architecture MVC consiste à insérer beaucoup de code au niveau de la couche contrôleur. Or, le code présent dans cette couche est difficilement réutilisable pour traiter d'autres actions de l'utilisateur.

Il est donc recommandé d'avoir une couche contrôleur la plus fine possible (avec le moins de code possible) pour déléguer les traitements métiers dans le modèle, et l'affichage dans la vue. Ainsi, on augmente la réutilisabilité du code.

Même s'il ne s'agit pas d'une règle stricte, de manière très pratique, une méthode d'un contrôleur ne dépasse généralement pas une vingtaine de lignes de code.

Lors de certaines applications de l'architecture MVC, on peut voir la vue appeler directement le modèle pour récupérer des données sans passer par le contrôleur. Bien que ce processus ne respecte pas strictement le modèle MVC, il est régulièrement utilisé.

Cela n'engendre pas de problèmes de conception du moment que l'on respecte ces deux conditions :

- La vue ne doit solliciter le modèle directement que pour un accès en lecture des données, jamais en écriture.
- La structure des données que va réceptionner la vue directement depuis le modèle doit rester simple et peu susceptible d'évoluer.

Avantages et limites

L'architecture MVC propose un modèle simple, robuste et largement utilisé pour le développement des applications web. Il permet de bien organiser le code source en maximisant sa réutilisabilité et en facilitant les évolutions.

L'immense majorité des frameworks PHP et des applications web ont aujourd'hui adoptés cette architecture.

Pour autant, dans certains cas particuliers, il est possible de conjuguer d'autres architectures à MVC, voire de la remplacer par une architecture plus spécifique.

À noter également que la mise en place de cette architecture nécessite plus de fichiers et plus de code, ce qui peut ne pas être justifié dans le cas de très petits projets mono page, par exemple.

Syntaxe	À retenir
---------	-----------

- | |
|--|
| <ul style="list-style-type: none">• La couche contrôleur réceptionne la demande de l'utilisateur, demande les données au modèle, sollicite la vue en passant ces données, puis répond à l'internaute.• Une bonne pratique consiste à avoir une couche contrôleur la plus fine possible (peu de code) afin de maximiser la réutilisabilité du modèle et de la vue. |
|--|

Exercice : Appliquez la notion

[solution n°4 p.30]

Exercice

Quelle est la couche qui échange directement avec l'internaute ?

- ☐ Modèle
- ☐ Vue
- ☐ Contrôleur

Exercice

Quelle est la couche qui doit normalement contenir le moins de code ?

- ☐ Modèle
- ☐ Vue
- ☐ Contrôleur

Exercice

Sélectionnez les avantages de l'architecture MVC.

- ☐ Simplicité
- ☐ Complexité
- ☐ Réutilisabilité
- ☐ Interactivité
- ☐ Évolutivité

X. Auto-évaluation

A. Exercice final

Exercice 1

[solution n°5 p.30]

Exercice

Dans l'exemple du *book* en ligne, si nous souhaitons ajouter une méthode `rechercherPhotos($motcle)` qui va retourner toutes les photos disposant du mot-clé, dans quelle classe doit-on intégrer cette méthode ?

Exercice

Quelle est la couche qui contient la logique applicative ?

- ☐ Modèle
- ☐ Vue
- ☐ Contrôleur

Exercice

Généralement le fichier `index.php` correspond au...

- ☐ Back controller
- ☐ Main controller
- ☐ Front controller
- ☐ Layout

Exercice

Que contient le layout dans notre exemple ?

- ☐ Les bibliothèques PHP
- ☐ Le gabarit HTML
- ☐ La configuration du projet

Exercice

Quel est l'intérêt d'avoir un contrôleur unique en entrée ?

- ☐ Minimiser le nombre de fichiers
- ☐ Faciliter la consultation du site par l'internaute
- ☐ Intégrer du code qui sera disponible dans toute l'application

Exercice

Quelle est l'utilité du fichier `Modele.php` dans notre exemple précédent ?

- ☐ Déléguer l'appel aux autres modèles
- ☐ Regrouper le code commun à tous les modèles
- ☐ Enregistrer les appels des modèles

Exercice

Pour des applications conséquentes, il est recommandé d'avoir plusieurs contrôleurs secondaires.

- ☐ Vrai
- ☐ Faux

Exercice

Les requêtes SQL à la base de données peuvent se trouver dans la ou les couches...

- ☐ Modèle
- ☐ Vue
- ☐ Contrôleur

Exercice

Dans quel dossier devrions-nous placer le fichier contenant le texte des mentions légales de notre *book* en ligne ?

Exercice

Il est envisageable de créer plusieurs layout pour une même application web.

- ☐ Vrai
- ☐ Faux

B. Exercice : Défi

Reprenons le code de notre *book* en ligne. L'objectif est de présenter les différentes séries photographiques réalisées. Une photographie appartient à une seule série.

Pour réaliser cet exercice, vous pouvez travailler sur l'environnement de travail :



1 <https://repl.it/>

Question

[solution n°6 p.32]

Pour cela, créons la nouvelle table **série**.

Table : `série`

Nom de la colonne	Type	Null	Extra
id	INT(11)	Non	AUTO_INCREMENT
titre	VARCHAR(100)	Non	
description	TEXT	Non	

Ensuite, il est nécessaire de créer une nouvelle page qui liste l'ensemble des séries et d'ajouter un lien dans le menu du site.

La page d'affichage des séries doit présenter le titre de la série dans une balise `<h2>` et sa description.

Bien sûr, le tout en respectant l'architecture MVC.

Solutions des exercices

Exercice p. 4 Solution n°1**Exercice**

L'architecture MVC divise l'application en combien de couches ?

☐ 2

☒ 3

☐ 4

 Trois couches, qui correspondent à l'acronyme MVC.

Exercice

Que signifie le V dans l'acronyme MVC ?

vue

 MVC signifie Modèle Vue Contrôleur.

p. 9 Solution n°2**Fichier models/Posts.php**

```
1 <?php
2
3 class Posts
4 {
5     private $pdo = null;
6
7     public function __construct()
8     {
9         try {
10             $this->pdo = new PDO('mysql:host=localhost;dbname=blog;charset=utf8', 'root',
11 'root');
12         } catch (PDOException $e) {
13             exit('Erreur : '.$e->getMessage());
14         }
15
16     public function getPosts()
17     {
18         if (!is_null($this->pdo)) {
19             $stmt = $this->pdo->query('SELECT title, image FROM post');
20         }
21         $posts = [];
22         while ($post = $stmt->fetchObject()) {
23             $posts[] = $post;
24         }
25
26         return $posts;
27     }
28 }
```

Fichier views/posts-list.php

```

1 <!DOCTYPE html>
2 <html lang="fr">
3 <head>
4     <title>Mon blog</title>
5     <meta charset="utf-8">
6     <meta name="viewport" content="width=device-width, initial-scale=1">
7 </head>
8 <body>
9 <header>
10     <h1>Mon blog</h1>
11 </header>
12 <section>
13     <nav>
14         <ul>
15             <li><a href="/">Accueil</a></li>
16         </ul>
17     </nav>
18     <article>
19         <?php foreach ($posts as $post): ?>
20             
21             <h2><?=$post->title ?></h2>
22         <?php endforeach; ?>
23     </article>
24 </section>
25 <footer>
26     <p>Mon blog - Tous droits réservés</p>
27 </footer>
28 </body>
29 </html>

```

Fichier index.php

```

1 <?php
2
3 require_once('models/Posts.php');
4 $posts = new Posts();
5 $posts = $posts->getPosts();
6 require_once('views/posts-list.php');

```

p. 19 Solution n°3

Fichier models/Model.php

```

1 <?php
2
3 trait Model
4 {
5     private $pdo = null;
6
7     public function __construct()
8     {
9         try {
10             $this->pdo = new PDO('mysql:host=localhost;dbname=blog;charset=utf8', 'root',
11 'root');
12         } catch (PDOException $e) {
13             exit('Erreur : '.$e->getMessage());
14         }
15     }
16 }

```

```

13     }
14 }
15 }

```

Fichier models/Posts.php

```

1 <?php
2
3 class Posts
4 {
5     use Model;
6
7     public function getPosts()
8     {
9         if (!is_null($this->pdo)) {
10             $stmt = $this->pdo->query('SELECT id, image, title FROM post');
11         }
12         $posts = [];
13         while ($post = $stmt->fetchObject('Post')) {
14             $posts[] = $post;
15         }
16
17         return $posts;
18     }
19 }

```

Fichier models/Post.php

```

1 <?php
2
3 class Post
4 {
5     use Model;
6
7     private $id;
8
9     private $image;
10
11     private $title;
12
13     public function getPost($id)
14     {
15         if (!is_null($this->pdo)) {
16             $stmt = $this->pdo->prepare('SELECT id, image, title FROM post WHERE id = ?');
17         }
18         $post = null;
19         if ($stmt->execute([$id])) {
20             $post = $stmt->fetchObject('Post');
21             if (!is_object($post)) {
22                 $post = null;
23             }
24         }
25
26         return $post;
27     }
28
29     public function getId()
30     {

```

```

31     return $this->id;
32 }
33
34 public function getImage()
35 {
36     return $this->image;
37 }
38
39 public function getTitle()
40 {
41     return $this->title;
42 }
43 }

```

Fichier views/posts-list.php

```

1 <?php
2 $titre = 'Mon blog';
3 ob_start();
4 ?>
5 <article>
6     <?php foreach ($posts as $post): ?>
7         <a href="?id=<?= $post->getId() ?>">
8             
9             </a>
10            <h2><?= $post->getTitle() ?></h2>
11        <?php endforeach; ?>
12 </article>
13 <?php
14 $content = ob_get_clean();
15 require_once('layout.php');

```

Fichier views/post-show.php

```

1 <?php
2 $titre = 'Un article de mon blog';
3 if (is_null($post)):
4     $content = "Cet article n'existe pas.";
5 else:
6     ob_start();
7     ?>
8     <article>
9         
10        <h2><?= $post->getTitle() ?></h2>
11    </article>
12    <?php
13        $content = ob_get_clean();
14    endif;
15    require_once('layout.php');

```

Fichier views/layout.php

```

1 <!DOCTYPE html>
2 <html lang="fr">
3 <head>
4     <title><?= $titre ?></title>
5     <meta charset="utf-8">
6     <meta name="viewport" content="width=device-width, initial-scale=1">

```

```

7 </head>
8 <body>
9 <header>
10     <h1>Mon blog</h1>
11 </header>
12 <section>
13     <nav>
14         <ul>
15             <li><a href="/">Accueil</a></li>
16         </ul>
17     </nav>
18     <?= $content ?>
19 </section>
20 <footer>
21     <p>Mon blog - Tous droits réservés</p>
22 </footer>
23 </body>
24 </html>

```

Fichier controllers/Controller.php

```

1 <?php
2
3 class Controller
4 {
5     public function getPosts()
6     {
7         $posts = new Posts();
8         $posts = $posts->getPosts();
9         require_once('views/posts-list.php');
10    }
11
12    public function getPost()
13    {
14        $post = new Post();
15        if (isset($_GET['id']) && is_numeric($_GET['id'])) {
16            $post = $post->getPost($_GET['id']);
17        }
18        require_once('views/post-show.php');
19    }
20 }

```

Fichier index.php

```


1 <?php
2
3 require_once('controllers/Controller.php');
4 require_once('models/Model.php');
5 require_once('models/Post.php');
6 require_once('models/Posts.php');
7 $controlleur = new Controller();
8 if (isset($_GET['id'])) {
9     $controlleur->getPost();
10 } else {
11     $controlleur->getPosts();
12 }

```

Exercice p. 21 Solution n°4


Exercice

Quelle est la couche qui échange directement avec l'internaute ?

- ☐ Modèle
- ☐ Vue
- ☒ Contrôleur
-  Le contrôleur frontal, souvent nommé `index.php`.


Exercice

Quelle est la couche qui doit normalement contenir le moins de code ?

- ☐ Modèle
- ☐ Vue
- ☒ Contrôleur
-  Le contrôleur, puisque le code présent dans cette couche est plus difficilement réutilisable.

Exercice

Sélectionnez les avantages de l'architecture MVC.


- ☒ Simplicité
- ☐ Complexité
- ☒ Réutilisabilité
- ☐ Interactivité
- ☒ Évolutivité
- 
 - Simplicité : trois couches facilement identifiables.
 - Réutilisabilité : cette organisation vise justement à éviter la duplication de code.
 - Évolutivité : si une partie de l'application doit changer, les conséquences sont minimisées (par exemple, si l'interface change, seule la couche vue devrait être impactée).

Exercice p. 22 Solution n°5

Exercice


Dans l'exemple du *book* en ligne, si nous souhaitons ajouter une méthode `rechercherPhotos($motcle)` qui va retourner toutes les photos disposant du mot-clé, dans quelle classe doit-on intégrer cette méthode ?

Photos

-  Cette méthode va retourner une collection de photos et sera donc intégrée à la classe `Photos`.


Exercice

Quelle est la couche qui contient la logique applicative ?

- ☒ Modèle
- ☐ Vue
- ☐ Contrôleur
-  Le modèle, avec les requêtes à la base de données, les calculs éventuels...


Exercice

Généralement le fichier `index.php` correspond au...

- ☐ Back controller
- ☐ Main controller
- ☒ Front controller
- ☐ Layout
-  Il s'agit du front controller, qui correspond au fichier appelé en premier par l'internaute.


Exercice

Que contient le layout dans notre exemple ?

- ☐ Les bibliothèques PHP
- ☒ Le gabarit HTML
- ☐ La configuration du projet
-  La structure HTML de la page de l'application, donc son gabarit.


Exercice

Quel est l'intérêt d'avoir un contrôleur unique en entrée ?

- ☐ Minimiser le nombre de fichiers
- ☐ Faciliter la consultation du site par l'internaute
- ☒ Intégrer du code qui sera disponible dans toute l'application
-  Par exemple intégrer une bibliothèque PHP qui doit être utilisable dans toute l'application.


Exercice

Quelle est l'utilité du fichier `Modele.php` dans notre exemple précédent ?

- ☐ Déléguer l'appel aux autres modèles
- ☒ Regrouper le code commun à tous les modèles
- ☐ Enregistrer les appels des modèles
-  Autrement dit, éviter de dupliquer du code dans chacune des classes du modèles (`Photos`, `Photo...`).


Exercice

Pour des applications conséquentes, il est recommandé d'avoir plusieurs contrôleurs secondaires.

- ☒ Vrai
- ☐ Faux
-  Vrai, sinon la classe contrôleur risque de devenir difficilement lisible avec des centaines de méthodes.


Exercice

Les requêtes SQL à la base de données peuvent se trouver dans la ou les couches...

- ☒ Modèle
- ☐ Vue
- ☐ Contrôleur
-  Uniquement dans le modèle, puisqu'il s'agit de logique applicative.


Exercice

Dans quel dossier devrions-nous placer le fichier contenant le texte des mentions légales de notre *book* en ligne ?

-  Par exemple : vues/mentions-legales.php.

Exercice

Il est envisageable de créer plusieurs layout pour une même application web.

- ☒ Vrai
- ☐ Faux
-  Vrai, dans le cas où certaines pages ont une structure assez différente. Par exemple, on peut imaginer que la page d'accueil est un menu différent et une organisation du contenu différente des autres pages.

p. 24 Solution n°6

Exemple index.php

```
1 <?php
2
3 require_once('contrôleurs/Contrôleur.php');
4 require_once('modeles/Modele.php');
5 require_once('modeles/Photo.php');
6 require_once('modeles/Photos.php');
7 require_once('modeles/Serie.php');
8 require_once('modeles/Series.php');
9 $contrôleur = new Contrôleur();
10 if (isset($_GET['page']) && 'photo' === $_GET['page']) {
11     $contrôleur->afficherPhoto();
12 } elseif (isset($_GET['page']) && 'series' === $_GET['page']) {
13     $contrôleur->listerSeries();
```



```

14 } else {
15     $controleur->listerPhotos();
16 }

```

Exemple controleurs/Controleur.php

```

1 <?php
2
3 class Controleur {
4
5     public function listerPhotos()
6     {
7         $photos = new Photos();
8         $photos = $photos->listerPhotos();
9         require_once('vues/liste-photos.php');
10    }
11
12    public function afficherPhoto()
13    {
14        $photo = new Photo();
15        $serie = new Serie();
16        if (isset($_GET['id']) && is_numeric($_GET['id'])) {
17            $photo = $photo->afficherPhoto($_GET['id']);
18        }
19        if (isset($photo) && !is_null($photo->getSerieid())) {
20            $serie = $serie->afficherSerie($photo->getSerieid());
21        }
22        require_once('vues/affiche-photo.php');
23    }
24
25    public function afficherMentions()
26    {
27        require_once('vues/affiche-mentions.php');
28    }
29
30    public function listerSeries()
31    {
32        $series = new Series();
33        $series = $series->listerSeries();
34        require_once('vues/liste-series.php');
35    }
36 }

```

Exemple vues/liste-series.php

```

1 <?php
2 $titre = 'Mes séries';
3 ob_start();
4 ?>
5 <article>
6     <?php foreach($series as $serie): ?>
7         <h2><?= $serie->getTitre() ?></h2>
8         <p><?= $serie->getDescription() ?></p>
9     <?php endforeach; ?>
10 </article>
11 <?php
12 $contenu = ob_get_clean();

```

```
13 require_once('layout.php');
```

Exemple vues/layout.php

```
1 <!DOCTYPE html>
2 <html lang="fr">
3 <head>
4     <title><?= $titre ?></title>
5     <meta charset="utf-8">
6     <meta name="viewport" content="width=device-width, initial-scale=1">
7 </head>
8 <body>
9 <header>
10     <h1>Mon book</h1>
11 </header>
12 <section>
13     <nav>
14         <ul>
15             <li><a href="/">Accueil</a></li>
16             <li><a href="?page=series">Séries</a></li>
17         </ul>
18     </nav>
19     <?= $contenu ?>
20 </section>
21 <footer>
22     <p>Mon book - Tous droits réservés</p>
23 </footer>
24 </body>
25 </html>
```

Exemple modeles/Serie.php

```
1 <?php
2
3 class Serie
4 {
5     use Modele;
6
7     private $id;
8
9     private $titre;
10
11     private $description;
12
13     public function afficherSerie($id)
14     {
15         if (!is_null($this->pdo)) {
16             $stmt = $this->pdo->prepare('SELECT * FROM serie WHERE id = ?');
17         }
18         $serie = null;
19         if ($stmt->execute([$id])) {
20             $serie = $stmt->fetchObject('Serie');
21             if (!is_object($serie)) {
22                 $serie = null;
23             }
24         }
25     }
```

```
26         return $serie;
27     }
28
29     public function getId()
30     {
31         return $this->id;
32     }
33
34     public function getTitre()
35     {
36         return $this->titre;
37     }
38
39     public function getDescription()
40     {
41         return $this->description;
42     }
43 }
```

Exemple modeles/Series.php

```
1 <?php
2
3 class Series
4 {
5     use Modele;
6
7     public function listerSeries()
8     {
9         if (!is_null($this->pdo)) {
10             $stmt = $this->pdo->query('SELECT * FROM serie');
11         }
12         $series = [];
13         while ($serie = $stmt->fetchObject('Serie')) {
14             $series[] = $serie;
15         }
16
17         return $series;
18     }
19 }
```