Adoptez une architecture MVC en PHP

I. Professionnalisez votre code

Contrairement à ce qu'on pourrait croire, ce n'est pas parce qu'un code "marche" qu'il est "professionnel".

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

- **Il est modulaire** : généralement découpé en de nombreux fichiers, où chaque fichier a un rôle et un seul 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** : et pour ça, il respecte très souvent les normes de formatage. En PHP, la plupart des développeurs recommandent de suivre la PSR-12. Je vous conseille d'y jeter un œil, si vous êtes curieux. Dans tous les cas, on commencera à se l'imposer dès nos premières lignes de code dans ce cours !

Les professionnels disent d'ailleurs qu'un code qui nécessite beaucoup de commentaires est un code trop complexe. Un code bien écrit contient des commentaires qui expliquent le pourquoi, pas le comment.

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, car vous avez même la possibilité de créer des *tests automatisés*. Vous savez que ça va marcher et votre code ne sera pas plus compliqué.

En résumé, on peut donc gagner du temps, travailler à plusieurs et ajouter des fonctionnalités facilement sans (presque) jamais causer des bugs.

II. Découvrez les limites d'un code de débutant

(Utilisez le zip fourni, créez une base blog et importez le fichier SQL)

Si on se concentre uniquement sur le côté visiteur, celui-ci affiche une liste de billets depuis la base de données :

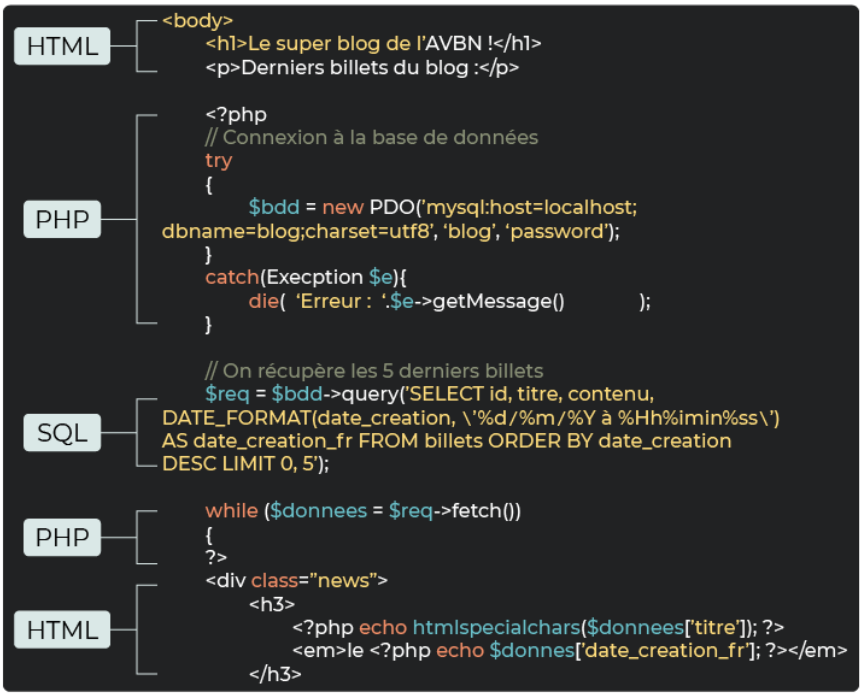
Le code de cette page tient dans un fichier index.php. Voici le code qui avait été écrit :

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

   <body>
      <h1>Le super blog de l'AVBN !</h1>
      <p>Derniers billets du blog :</p>

      <?php
      // Connexion à la base de données
      try
      {
          $bdd = new PDO('mysql:host=localhost;dbname=blog;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</a></em>
        </p>
      </div>
      <?php
      } // Fin de la boucle des billets
      $req->closeCursor();
      ?>
   </body>
</html>
```

Si vous avez bien lu le chapitre précédent, il y a des défauts qui devraient vous sauter aux yeux.

- **Le code est en français**. Pour le titre de la page ("Le blog de l'AVBN") et les autres informations affichées à l'utilisateur, c'est normal. Par contre, le code PHP lui-même ne devrait pas être en français (par exemple la variable $donnees). C'est mauvais, car le code ne pourra pas être lu par des développeurs qui ne parlent pas français.
- **Le formatage ne respecte aucune règle**. Les tabulations n'ont pas toujours le même nombre d'espaces – et encore, quand il y a des tabulations. Les accolades se baladent où bon leur semble. Et j'en passe. Il est beaucoup plus difficile de s'y retrouver et donc de comprendre la logique du code écrit.
- **Tout est dans un seul fichier**. On y mélange des opérations en PHP, des requêtes SQL et du code HTML. Il ne manquerait plus que du code CSS au milieu et ça serait le pompon ! Pas étonnant qu'on ait besoin de commentaires pour se repérer dans les différentes sections du fichier :



Il fonctionne, certes. Mais si vous commencez à ajouter de nouvelles fonctionnalités, vous allez avoir besoin d'ajouter de nouvelles requêtes SQL au milieu. Très vite, le fichier va grossir et faire plusieurs centaines de lignes. Ça marchera toujours. Ça ne sera même pas forcément plus lent.

Par contre, à chaque fois que vous allez ouvrir votre fichier, vous allez pousser un **grand soupir**.

Et bien sûr, n'oubliez pas que la plupart du temps les projets se font en équipe. Le webdesigner, qui fait l'intégration HTML et CSS du site, va devoir naviguer entre vos fonctions PHP. De votre côté, vous vous perdrez au milieu de ses imbrications de balises. Et au moment de rassembler votre travail, ce sera la fête des conflits Git !

Pour vous éviter ce genre de situation nous allons voir comment on **peut mieux organiser ce code en le découpant**.

III. Isolez l'affichage du traitement PHP

La première bonne pratique que nous devons prendre consiste à éviter de mélanger l'affichage du reste.

III.1. Faites une première séparation

La première bonne pratique que nous devons prendre consiste à éviter de mélanger l'affichage du reste.

Commençons par séparer le code en deux sections :

1. **On récupère les données**. C'est là qu'on fait notamment la requête SQL. Vous voyez aussi qu'on crée une variable structurée, appelée $posts, qui contient les "données brutes" utiles à l'affichage de chaque billet de blog. Je vous recommande d'utiliser **uniquement des types PHP simples** pour ces variables : tableaux, chaînes de caractères, nombres entiers...
2. **On affiche les données** en les formatant. Notez qu'on utilise uniquement ces fameuses "données brutes" en provenance de la première section. Étant donné qu'on dispose de variables de types simples, il est assez facile de les formater comme on le souhaite (avec les fonctions nl2br()et htmlspecialchars(), par exemple).

C'est relativement simple à faire :

```php
<?php
// We connect to the database.
try {
    $database = new PDO('mysql:host=localhost;dbname=blog;charset=utf8', 'root',
'');
} catch(Exception $e) {
    die('Erreur : '.$e->getMessage());
}

// We retrieve the 5 last blog posts.
$statement = $database->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");
$posts = [];
while (($row = $statement->fetch())) {
    $post = [
        'title' => $row['titre'],
        'french_creation_date' => $row['date_creation_fr'],
        'content' => $row['contenu'],
    ];
    $posts[] = $post;
}

?>

<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Le blog de l'AVBN</title>
        <link href="style.css" rel="stylesheet" />
    </head>

    <body>
        <h1>Le super blog de l'AVBN !</h1>
        <p>Derniers billets du blog :</p>

        <?php
        foreach ($posts as $post) {
        ?>
            <div class="news">
                <h3>
                    <?php echo htmlspecialchars($post['title']); ?>
                    <em>le <?php echo $post['french_creation_date']; ?></em>
                </h3>
                <p>
                    <?php
                    // We display the post content.
                    echo nl2br(htmlspecialchars($post['content']));
                    ?>
                    <br />
                    <em><a href="#">Commentaires</a></em>
                </p>
            </div>
        <?php
        } // The end of the posts loop.
        ?>
    </body>
</html>
```

Grâce à cette séparation, on sait plus facilement repérer la section qui s'occupe de récupérer les données de celle qui les affiche. Par ailleurs, on a mis en œuvre les conventions en nommant correctement nos variables et en écrivant le code et les commentaires en anglais.

C'est déjà bien mieux, mais ce n'est encore qu'un début. On peut aller plus loin et découper ça en 2 fichiers distincts.

III.2. Séparez le code en 2 fichiers

- index.php (il y a toujours un index.php, c'est le fichier de base de votre site lu en premier) : il s'occupera de récupérer les données et d'appeler l'affichage.
- templates/homepage.php : il affichera les données dans la page.

On crée un dossier qui contient seulement les gabarits d'affichage (les templates, en anglais) de chacune des pages du blog. Ils sont séparés du code responsable d'agir sur les données. Vous verrez plus tard dans votre apprentissage sur les frameworks (Symfony ou Laravel, par exemple), que c'est une pratique très souvent utilisée. Vous saurez que ça aide votre code à être plus maintenable et que ça favorise le travail en équipe !

index.php :

```php
<?php

// We connect to the database.
try {
    $database = new PDO('mysql:host=localhost;dbname=blog;charset=utf8', 'root',
'');
} catch(Exception $e) {
    die('Erreur : '.$e->getMessage());
}

// We retrieve the 5 last blog posts.
$statement = $database->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"
);
$posts = [];
while (($row = $statement->fetch())) {
    $post = [
        'title' => $row['titre'],
        'french_creation_date' => $row['date_creation_fr'],
        'content' => $row['contenu'],
    ];

    $posts[] = $post;
}

require('templates/homepage.php');
?>
```

Le code se contente d'appeler à la fin le fichier templates/homepage.php.

templates/homepage.php :

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

    <body>
        <h1>Le super blog de l'AVBN !</h1>
        <p>Derniers billets du blog :</p>

        <?php
        foreach ($posts as $post) {
        ?>
            <div class="news">
                <h3>
                    <?php echo htmlspecialchars($post['title']); ?>
                    <em>le <?php echo $post['french_creation_date']; ?></em>
                </h3>
                <p>
                    <?php
                    // We display the post content.
                    echo nl2br(htmlspecialchars($post['content']));
                    ?>
                    <br />
                    <em><a href="#">Commentaires</a></em>
                </p>
            </div>
        <?php
        } // The end of the posts loop.
        ?>
    </body>
</html>
```

Le fichier de l'affichage fonctionne comme avant. Quand il est chargé, la variable $posts lui est automatiquement transmise par index.php grâce au require. Il ne lui reste plus qu'à afficher les informations.

Le rôle de ce code est **uniquement d'afficher des informations**. On n'y fait pas des opérations ou des calculs complexes, on n'y fait pas des requêtes en base de données.

Nous faisons ici ce qu'on appelle de la **refactorisation**. On change l'organisation du code, sans ajouter de nouvelles fonctionnalités. Le but est d'avoir un code plus facile à modifier par la suite.

III.3. En résumé

- Une première étape pour rendre son code professionnel est d'isoler la partie du code responsable de générer l'affichage pour l'utilisateur.
- Les variables fournies aux templates doivent être le plus simple possible, pour rendre plus facile le travail sur l'interface utilisateur.
- Les gabarits d'affichage peuvent contenir du code – et c'est d'ailleurs ce qui fait toute leur puissance – mais celui-ci n'a pour rôle que d'afficher des informations.

IV.Isolez l'accès aux données

Notre code est déjà beaucoup mieux, mais nous pouvons aller un cran plus loin. Nous allons séparer complètement l'accès aux données (tout le traitement SQL) dans un fichier spécifique.

Nous allons avoir 3 fichiers :

- model/model.php : se connecte à la base de données et récupère les billets.
- templates/homepage.php : affiche la page. Ce fichier ne va pas changer du tout.
- index.php : fait le lien entre le modèle et l'affichage (oui, juste ça !).

Ces 3 fichiers forment la base d'une structure MVC (Modèle - Vue - Contrôleur) :

- Le modèle traite les données (model/model.php).
- La vue affiche les informations (templates/homepage.php).
- Le contrôleur fait le lien entre les deux (index.php).

src/model.php :

Notre nouveau fichier model/model.php contient une fonction getPosts() qui renvoie la liste des billets :

```php
<?php

function getPosts() {
    // We connect to the database.
    try {
        $database = new PDO('mysql:host=localhost;dbname=blog;charset=utf8',
'root', '');
    } catch(Exception $e) {
        die('Erreur : '.$e->getMessage());
    }

    // We retrieve the 5 last blog posts.
    $statement = $database->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"
    );
    $posts = [];
    while (($row = $statement->fetch())) {
        $post = [
            'title' => $row['titre'],
            'french_creation_date' => $row['date_creation_fr'],
            'content' => $row['contenu'],
        ];

        $posts[] = $post;
    }

    return $posts;
}

?>
```

templates/homepage.php :

pas de changement.

index.php :

La nouveauté, c'est désormais l'index (aussi appelé le contrôleur) qui sert d'intermédiaire entre le modèle et la vue :

```php
<?php

require('model/model.php');

$posts = getPosts();

require('templates/homepage.php');
?>
```

3 lignes, 3 étapes toutes simples :

1. On charge le fichier du modèle. Il ne se passe rien pour l'instant, parce qu'il ne contient qu'une fonction.
2. On appelle la fonction, ce qui exécute le code à l'intérieur de model/model.php. On y récupère la liste des billets dans la variable $posts.
3. On charge le fichier de la vue (l'affichage), qui va présenter les informations dans une page HTML.

Voilà, notre code commence à avoir les bases d'une structure plus solide !

En résumé :

- Le code qui accède aux données fait partie d'un second lot de code à isoler, à côté du code gérant l'affichage.
- Il y a des morceaux de code en PHP qui ne font rien par eux-mêmes et qui se mettent simplement à disposition des autres morceaux de code.
- Il y a un troisième lot de code à isoler. Il est responsable de gérer la requête HTTP de l'utilisateur, d'orchestrer la lecture des données et de renvoyer la réponse HTML.

V. Soignez la cosmétique du code

Notre code est maintenant découpé en 3 fichiers :

- Un pour le **traitement PHP** : il récupère les données de la base. On l'appelle le **modèle**.
- Un pour **l'affichage** : il affiche les informations dans une page HTML. On l'appelle la **vue**.
- Un pour **faire le lien entre les deux** : on l'appelle le **contrôleur**.

Cette structure est bonne, mais nous allons faire quelques améliorations cosmétiques. Ça peut paraître anodin, mais ça fera une vraie différence à la fin.

Voici ce que nous allons améliorer :

- l'anglais ;
- la balise de fermeture PHP ;
- l'utilisation de short open tags.

V.1. Découvrez le code amélioré

index.php :

```php
<?php

require('model/model.php');

$posts = getPosts();

require('templates/homepage.php');
```

Vous constaterez que j'ai enlevé la balise de fermeture `?>` à la fin du fichier et c'est une amélioration.

Les développeurs professionnels **enlèvent** en effet la balise de fermeture PHP dans les pages qui ne contiennent **que du PHP**. Sans rentrer dans les détails, ça permet d'éviter que ces fichiers n'envoient par erreur du code HTML sous forme d'espaces blancs alors qu'ils ne devraient pas. Retirez donc ce `?>` si votre fichier ne contient que du PHP, comme indiqué dans la recommandation officielle PSR-12 qui dit : *The closing `?>` tag MUST be omitted from files containing only PHP.*

model/model.php :

```php
<?php

function getPosts() {
    // We connect to the database.
    try {
        $database = new PDO('mysql:host=localhost;dbname=blog;charset=utf8',
'root', '');
    } catch(Exception $e) {
        die('Erreur : '.$e->getMessage());
    }

    // We retrieve the 5 last blog posts.
    $statement = $database->query(
        "SELECT id, title, content, DATE_FORMAT(creation_date, '%d/%m/%Y à
%Hh%imin%ss') AS french_creation_date FROM posts ORDER BY creation_date DESC LIMIT
0, 5"
    );
    $posts = [];
```

```php
    while (($row = $statement->fetch())) {
        $post = [
            'title' => $row['title'],
            'french_creation_date' => $row['french_creation_date'],
            'content' => $row['content'],
        ];

        $posts[] = $post;
    }

    return $posts;
}
```

Dans le modèle, les requêtes se font depuis une base de données en anglais (les noms des champs en base ont été renommés).

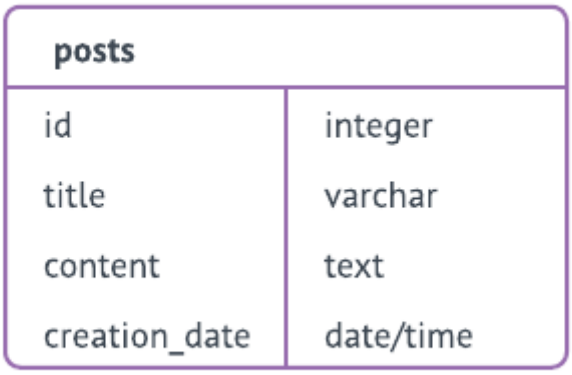
templates/homepage.php :

```php
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Le blog de l'AVBN</title>
        <link href="style.css" rel="stylesheet" />
    </head>

    <body>
        <h1>Le super blog de l'AVBN !</h1>
        <p>Derniers billets du blog :</p>

        <?php
        foreach ($posts as $post) {
        ?>
            <div class="news">
                <h3>
                    <?= htmlspecialchars($post['title']) ?>
                    <em>le <?= $post['french_creation_date']; ?></em>
                </h3>
                <p>
                    <?= nl2br(htmlspecialchars($post['content'])) ?>
                    <br />
                    <em><a href="#">Commentaires</a></em>
                </p>
            </div>
        <?php } ?>
    </body>
</html>
```

Ce n'est pas obligatoire, mais on en a profité pour remplacer les echo par les **short echo tags** < ?= htmlspecialchars($post['title']) ?>. C'est un raccourci en PHP pour faciliter la lisibilité du code.

V.1.1. La base de données

Pour avoir du code uniquement en anglais, la base de données a été renommée en anglais. Pour l'instant, nous avons une seule table posts qui doit représenter les posts de blog :



V.1.2. En résumé

- Il est recommandé de garder uniquement la balise ouvrante <?php dans les fichiers qui contiennent seulement du PHP. Ceux qui contiennent aussi du HTML ne changent pas.
- La syntaxe des "short open tags" peut être utilisée pour avoir des gabarits d'affichage plus concis.
- En tant que développeur PHP, vous êtes aussi responsable de la bonne gestion de la base de données. Vous devez la traiter comme votre code : des noms clairs, de l'anglais…

VI.Découvrez comment fonctionne une architecture MVC

Nous avons petit à petit construit les bases d'une architecture MVC.

Nous avons en fait reproduit le même raisonnement que de nombreux développeurs avant nous. 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*. Vous les retrouverez aussi, en français, sous le nom de patrons de conception.

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

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 :

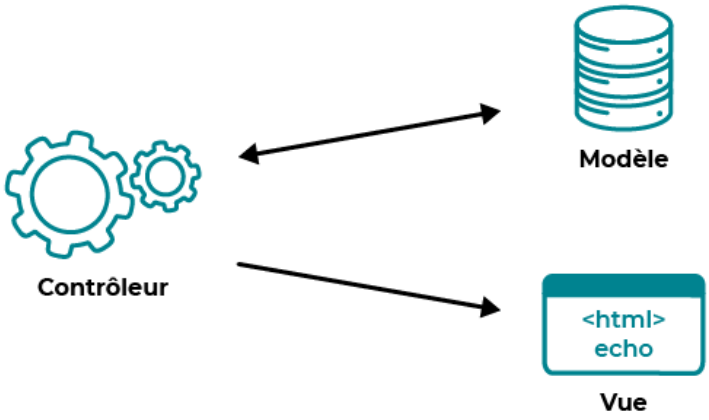
- **Modèle** : cette partie gère ce qu'on appelle la **logique métier** de votre site. Elle comprend notamment la gestion des données qui sont stockées, mais aussi tout le code qui prend des décisions autour de ces données. Son objectif est de fournir une interface d'action la plus simple possible au contrôleur. On y trouve donc entre autres des algorithmes complexes et des 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 les **échanges** avec l'utilisateur. C'est en quelque sorte l'intermédiaire entre l'utilisateur, le modèle et la vue. Le contrôleur va recevoir des requêtes de l'utilisateur. Pour chacune, il va demander au modèle d'effectuer certaines actions (lire des articles de blog depuis une base de données, supprimer un commentaire) et de lui renvoyer les résultats (la liste des articles, si la suppression est réussie). Puis il va adapter ce résultat et le donner à la vue. Enfin, il va renvoyer la nouvelle page HTML, générée par la vue, à l'utilisateur.

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

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



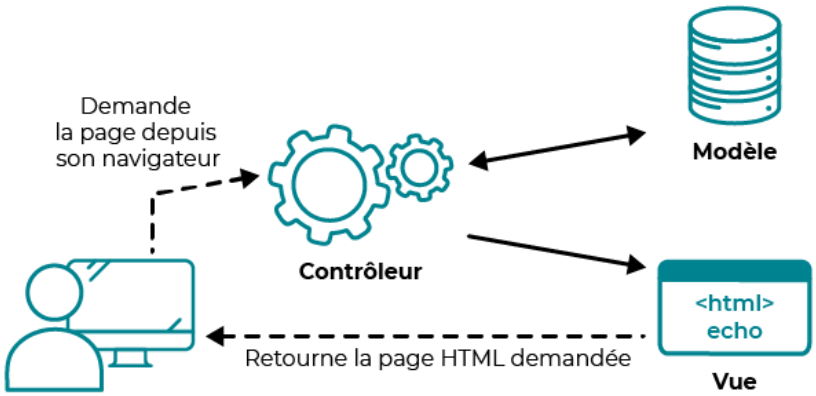
Il faut tout d'abord retenir que **le contrôleur est le chef d'orchestre** : c'est lui qui reçoit la requête du visiteur et qui contacte d'autres fichiers (le modèle et la vue) pour leur demander des services.

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 numéro 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.

Vous pouvez retenir que le contrôleur sert presque uniquement à faire la jonction entre le modèle et la vue.

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 grand nombre de sites professionnels !



Voilà la théorie. Vous avez pu expérimenter la pratique dans les chapitres précédents où, pour notre exemple très simple du blog, nous avions 3 fichiers.

Pour l'instant, vous avez quand même vu que ça vous permettait de travailler avec d'autres professionnels, et c'est déjà un énorme gain ! Cela dit, en travaillant tout seul ou avec d'autres développeurs, le gain ne semble pas très flagrant. Mais attendez que le projet se complexifie un peu et vous allez vite comprendre pourquoi nous avons besoin de cette structure.

VII. Affichez des commentaires

On vient de vous demander une nouvelle fonctionnalité ! On aimerait pouvoir afficher une page avec les commentaires de chaque billet.

Vous vous souvenez du lien "Commentaires" sous chaque billet ?

Lorsqu'on clique dessus, on va afficher une page avec le billet et sa liste de commentaires.

Comment s'y prendre avec une architecture MVC ?

Je vous propose le plan suivant pour arriver à vos fins :

- Vous commencez par écrire la **vue**. Après tout, votre objectif principal reste d'afficher la page des commentaires à l'utilisateur !
- Ensuite, vous allez écrire un **contrôleur**, mais en version très rapide, qui fera passer des fausses données à la vue. Ça vous permettra de vérifier que votre affichage correspond à vos attentes.

- Vous affinerez le **contrôleur** en le rendant dynamique et en commençant à imaginer les services que vous souhaiteriez demander à votre modèle.
- Vous finirez en implémentant votre **modèle**, pour qu'il réponde correctement aux demandes de votre contrôleur.

VII.1. Créez la vue

Qui dit nouvelle vue, dit nouveau fichier dans notre dossier templates/. On veut afficher un article de blog, alors nous pouvons l'appeler templates/post.php.

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

    <body>
        <h1>Le super blog de l'AVBN !</h1>
        <p><a href="index.php">Retour à la liste des billets</a></p>

        <div class="news">
            <h3>
                <?= htmlspecialchars($post['title']) ?>
                <em>le <?= $post['french_creation_date'] ?></em>
            </h3>

            <p>
                <?= nl2br(htmlspecialchars($post['content'])) ?>
            </p>
        </div>

        <h2>Commentaires</h2>

        <?php
        foreach ($comments as $comment) {
        ?>
        <p><strong><?= htmlspecialchars($comment['author']) ?></strong> le <?=
$comment['french_creation_date'] ?></p>
        <p><?= nl2br(htmlspecialchars($comment['comment'])) ?></p>
        <?php
        }
        ?>
    </body>
</html>
```

Dans cette vue, on affiche :

- le billet : on utilise une variable tableau $post, avec les index title, french_creation_date et content;
- les commentaires : un tableau de tableaux, avec les index author, french_creation_date et comment.

Pour l'instant, on ne peut pas encore tester notre code HTML. Il nous faut un contrôleur !

VII.2. Créez le contrôleur "simplet"

Nous avions déjà écrit un contrôleur index.php pour gérer la liste des derniers billets. Je vous propose d'écrire un autre contrôleur post.php qui affiche un post et ses commentaires.

```php
<?php

$post = [
    'title' => 'Un faux titre.',
    'french_creation_date' => '03/03/2022 à 12h14min42s',
    'content' => "Le faux contenu de mon billet.\nC'est fantastique !",
];
$comments = [
    [
        'author' => 'Un premier faux auteur',
        'french_creation_date' => '03/03/2022 à 12h15min42s',
        'comment' => 'Un faux commentaire.\n Le premier.',
    ],
    [
        'author' => 'Un second faux auteur',
        'french_creation_date' => '03/03/2022 à 12h16min42s',
        'comment' => 'Un faux commentaire.\n Le second.',
    ],
];

require('templates/post.php');
```

Ce faux contrôleur sert seulement à **tester que mon affichage correspond à mes attentes**. Vous n'avez pas encore besoin de données dynamiques à ce stade et ça ne vous coûte presque pas de temps de créer ces variables manuellement. Maintenant, si vous allez sur la page /post.php dans votre navigateur, vous pouvez vérifier : votre page de billet affiche bien l'article, suivi des commentaires :

VII.3. Dynamisez le contrôleur

Notre prochaine étape, c'est de faire en sorte que notre contrôleur soit encore plus puissant !

Déjà, on veut qu'il **prenne en paramètre un billet précis**. Il va falloir modifier les liens présents sur la page d'accueil templates/homepage.php, à la ligne 24, pour y renseigner notre nouvelle URL. On choisit d'utiliser le paramètre GET intitulé id pour faire passer l'information à notre nouveau contrôleur :

```
<em><a href="post.php?id=<?= urlencode($post['identifier'])
?>">Commentaires</a></em>
```

On a besoin d'une nouvelle propriété identifier au niveau de chaque $post de notre page d'accueil. Et on va directement demander à notre modèle de nous la donner :

```
<?php
// model/model.php:17

$post = [
    'title' => $row['title'],
    'french_creation_date' => $row['french_creation_date'],
    'content' => $row['content'],
    'identifier' => $row['id'],
];
```

Quand vous cliquez sur les liens "Commentaires" de la page d'accueil, vous accédez dorénavant à votre nouvelle page !

Modifions maintenant le contrôleur post.php pour prendre en compte ce paramètre GET :

```
<?php
require('model/model.php');

if (isset($_GET['id']) && $_GET['id'] > 0) {
    $identifier = $_GET['id'];
} else {
    echo 'Erreur : aucun identifiant de billet envoyé';

    die;
}

$post = getPost($identifier);
$comments = getComments($identifier);

require('templates/post.php');
```

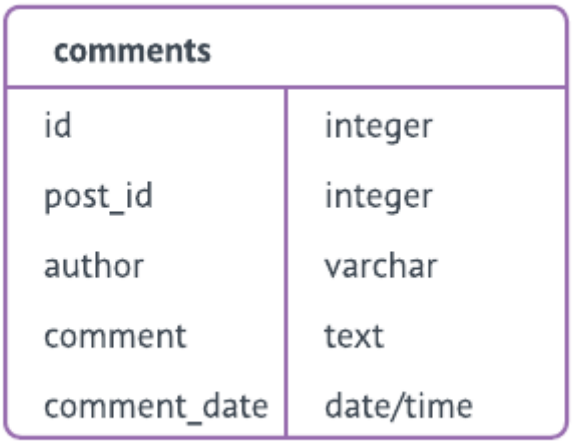
Il fait un test, un contrôle : il vérifie qu'on a bien reçu en paramètre un id dans l'URL ($_GET['id']).

Ensuite, il appelle les 2 fonctions du modèle dont on va avoir besoin :getPost et getComment. On récupère ça dans nos deux variables. Bien sûr, on attend du modèle qu'il nous renvoie les mêmes propriétés que celles qu'on avait définies :

- title, french_creation_date et content pour les billets ;
- author, french_creation_date et comment pour les commentaires.

VII.4. Mettez à jour le modèle

Tout d'abord, ajoutons une table pour gérer les commentaires dans la base. Elle aura cette structure :



Maintenant, concentrons-nous sur le code du modèle. Pour l'instant, notre fichier model/model.php ne contient qu'une seule fonction getPosts qui récupère tous les derniers posts de blog.

On va écrire 2 nouvelles fonctions :

- getPost(au singulier !), qui récupère un post précis en fonction de son ID ;
- getComments, qui récupère les commentaires associés à un ID de post.

Cela donne :

```php
<?php
// model/model.php

function getPosts()
{
    // ... (déjà écrite)
}

function getPost($identifier) {
    try {
        $database = new PDO('mysql:host=localhost;dbname=blog;charset=utf8',
'root', '');
    } catch(Exception $e) {
        die('Erreur : '.$e->getMessage());
    }

    $statement = $database->prepare(
        "SELECT id, title, content, DATE_FORMAT(creation_date, '%d/%m/%Y à
%Hh%imin%ss') AS french_creation_date FROM posts WHERE id = ?"
    );
    $statement->execute([$identifier]);

    $row = $statement->fetch();
    $post = [
        'title' => $row['title'],
        'french_creation_date' => $row['french_creation_date'],
```

```
        'content' => $row['content'],
    ];

    return $post;
}

function getComments($identifier)
{
    try {
        $database = new PDO('mysql:host=localhost;dbname=blog;charset=utf8',
'root', '');
    } catch(Exception $e) {
        die('Erreur : '.$e->getMessage());
    }

    $statement = $database->prepare(
        "SELECT id, author, comment, DATE_FORMAT(comment_date, '%d/%m/%Y à
%Hh%imin%ss') AS french_creation_date FROM comments WHERE post_id = ? ORDER BY
comment_date DESC"
    );
    $statement->execute([$identifier]);

    $comments = [];
    while (($row = $statement->fetch())) {
        $comment = [
            'author' => $row['author'],
            'french_creation_date' => $row['french_creation_date'],
            'comment' => $row['comment'],
        ];
        $comments[] = $comment;
    }

    return $comments;
}
```

Ces deux nouvelles fonctions prennent un paramètre : l'identifiant du billet qu'on recherche. Cela nous permet notamment de ne sélectionner que les commentaires liés au post concerné.

Par contre, on remarque du code qui se répète. C'est le cas en particulier de la connexion à la base de données avec les blocs try/catch. Factorisons ça.

Nous allons créer une fonction dbConnect() qui va renvoyer la connexion à la base de données. On l'ajoute à la fin du fichier :

```
<?php
// model/model.php

function getPosts() {
    $database = dbConnect();
    $statement = $database->query(
        "SELECT id, title, content, DATE_FORMAT(creation_date, '%d/%m/%Y à
%Hh%imin%ss') AS french_creation_date FROM posts ORDER BY creation_date DESC LIMIT
0, 5"
    );
    $posts = [];
```

```php
    while (($row = $statement->fetch())) {
        $post = [
            'title' => $row['title'],
            'french_creation_date' => $row['french_creation_date'],
            'content' => $row['content'],
            'identifier' => $row['id'],
        ];

        $posts[] = $post;
    }

    return $posts;
}

function getPost($identifier) {
    $database = dbConnect();
    $statement = $database->prepare(
        "SELECT id, title, content, DATE_FORMAT(creation_date, '%d/%m/%Y à
%Hh%imin%ss') AS french_creation_date FROM posts WHERE id = ?"
    );
    $statement->execute([$identifier]);

    $row = $statement->fetch();
    $post = [
        'title' => $row['title'],
        'french_creation_date' => $row['french_creation_date'],
        'content' => $row['content'],
    ];

    return $post;
}

function getComments($identifier)
{
    $database = dbConnect();
    $statement = $database->prepare(
        "SELECT id, author, comment, DATE_FORMAT(comment_date, '%d/%m/%Y à
%Hh%imin%ss') AS french_creation_date FROM comments WHERE post_id = ? ORDER BY
comment_date DESC"
    );
    $statement->execute([$identifier]);

    $comments = [];
    while (($row = $statement->fetch())) {
        $comment = [
            'author' => $row['author'],
            'french_creation_date' => $row['french_creation_date'],
            'comment' => $row['comment'],
        ];

        $comments[] = $comment;
    }

    return $comments;
}

// Nouvelle fonction qui nous permet d'éviter de répéter du code
function dbConnect()
{
    try {
```

```
        $database = new PDO('mysql:host=localhost;dbname=blog;charset=utf8',
'root', '');

        return $database;
    } catch(Exception $e) {
        die('Erreur : '.$e->getMessage());
    }
}
```

Vous pouvez maintenant tester que tout fonctionne.

VII.5. En résumé

Nous avons maintenant les fichiers suivants :

- model/model.php: le modèle, qui contient différentes fonctions pour récupérer des informations dans la base.
- index.php: le contrôleur de la page d'accueil. Il fait le lien entre le modèle et la vue.
- templates/homepage.php: la vue de la page d'accueil. Elle affiche la page.
- post.php: le contrôleur d'un billet et ses commentaires. Il fait le lien entre le modèle et la vue.
- templates/post.php: la vue d'un billet et ses commentaires. Elle affiche la page.

VIII. Créez un template de page

Essayons maintenant d'améliorer nos vues. Il y a du code qui se répète…

VIII.1. Créez un layout

On va créer un **layout** (une disposition, traduit littéralement) de page. On va y retrouver **toute la structure de la page**, avec des "trous" à remplir.

Attention à ne pas faire l'amalgame entre layout et template ! Un **layout** est une façon spécifique d'utiliser un template. Il sert à créer une disposition d'affichage. Dans un fichier layout, les "trous" à remplir seront très souvent comblés... par des **templates** !

Voici notre fichier templates/layout.php :

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

    <body>
        <?= $content ?>
    </body>
</html>
```

Il y a 2 "trous" à remplir dans ce layout : le <title> et le contenu de la page.

Évidemment, on pourrait faire plus compliqué si on voulait (par exemple, on pourrait réserver un espace pour personnaliser le menu). Mais vous voyez l'idée : vous **créez la structure** de votre page et vous **remplissez les trous par des variables**.

Il faut maintenant définir ce qu'on met dans ces variables. Voici comment on peut le faire dans la vue templates/homepage.php qui affiche la liste des derniers billets :

```
<?php $title = "Le blog de l'AVBN"; ?>

<?php ob_start(); ?>
<h1>Le super blog de l'AVBN !</h1>
<p>Derniers billets du blog :</p>

<?php
foreach ($posts as $post) {
?>
    <div class="news">
        <h3>
            <?= htmlspecialchars($post['title']); ?>
            <em>le <?= $post['french_creation_date']; ?></em>
        </h3>
        <p>
            <?= nl2br(htmlspecialchars($post['content'])); ?>
            <br />
            <em><a href="post.php?id=<?= urlencode($post['identifier'])
?>">Commentaires</a></em>
        </p>
    </div>
<?php
}
?>
<?php $content = ob_get_clean(); ?>

<?php require('layout.php') ?>
```

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.
3. 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 24) et on met le tout dans $content. 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.

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. On a un code bien plus flexible pour définir des "morceaux" de page dans des variables.

Ensuite mettez à jour templates/post.php de la même manière.

VIII.2. En résumé

- Un système de templates est un système où des morceaux de vue sont paramétrables.
- On peut utiliser un template de "mise en page", appelé layout, pour factoriser le code HTML redondant.
- Les fonctions ob_start() et ob_get_clean() ne sont pas nécessaires, mais aident à implémenter un système de templates.

IX. Créez un routeur

Pour l'instant, 2 fichiers permettent d'accéder aux pages de notre site. Ce sont les 2 contrôleurs :

- **index.php** : accueil du site, liste des derniers billets ;
- **post.php** : affichage d'un billet et de ses commentaires.

Si on continue comme ça, on va avoir une quantité faramineuse de fichiers à la racine de notre dépôt de code : contact.php, editComment.php…

Ça fonctionnerait. Mais ce n'est pas parce que ça fonctionnerait que ce serait forcément l'idéal.

Plus vous ajouterez de **fonctionnalités** sur le blog, plus le **nombre de fichiers va augmenter**. Ça deviendra très difficile pour l'équipe de développeurs de s'y retrouver quand une modification sera demandée.

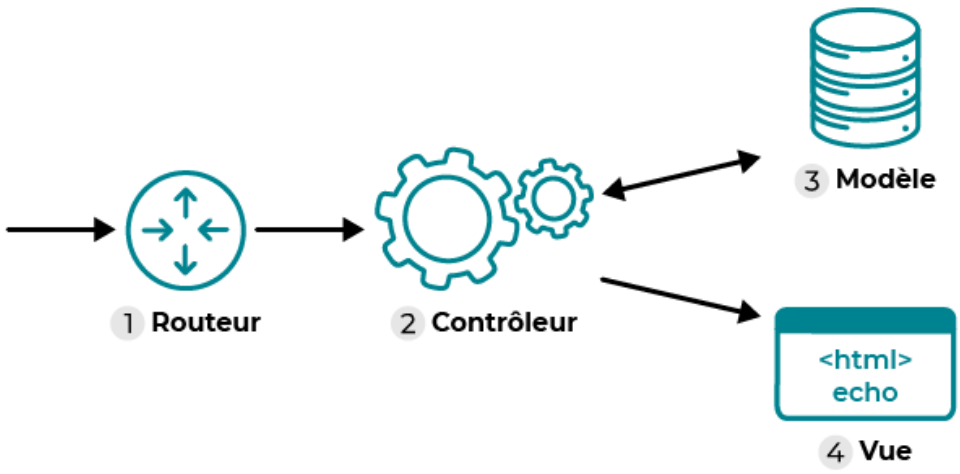
En plus de ça, on a souvent besoin d'exécuter du code en amont des contrôleurs :

- pour monitorer ce qui se passe sur votre site ;
- pour gérer les sessions utilisateurs ;
- ou encore du code "outillage", pour simplifier l'accès au contenu des requêtes HTTP.

Quand vous voudrez mettre en place ou modifier ces points, il faudra que vous passiez sur chacun des fichiers contrôleurs. Ce n'est pas optimal…

IX.1. Appréhendez la nouvelle structure des fichiers

Pour faciliter la maintenance, il est plus simple de passer par un contrôleur frontal, qui va jouer le rôle de **routeur**. Son objectif va être **d'appeler le bon contrôleur** (on dit qu'il route les requêtes).



On va travailler ici sur deux sections de code bien distinctes :

- index.php : ce sera le nom de notre **routeur**. Le routeur étant le premier fichier qu'on appelle en général sur un site, c'est normal de le faire dansindex.php. Il va se charger d'appeler le bon contrôleur.
- controllers/: ce dossier contiendra nos contrôleurs dans des fonctions. **On va y regrouper nos anciens index.php et post.php**.

Chaque contrôleur a le droit à son propre fichier. C'est une **unité de code** qui a une taille souvent suffisamment grande pour ne pas devoir la mélanger avec d'autres.

On va faire passer un paramètre action dans l'URL de notre routeur index.php pour savoir quelle page on veut appeler. Par exemple :

- index.php : va afficher la page d'accueil avec la liste des billets ;
- index.php?action=pos t: va afficher un billet et ses commentaires.

Certains trouvent que l'URL n'est plus très jolie sous cette forme. Peut-être préféreriez-vous voirmonsite.com/post plutôt que index.php?action=post.

Heureusement, cela peut se régler avec un mécanisme de réécriture d'URL (*URL rewriting*). On ne l'abordera pas ici, car ça se fait dans la configuration du serveur web (Apache), mais vous pouvez vous renseigner sur le sujet si vous voulez !

IX.2. Créez les contrôleurs

Commençons par notre dossier controllers/. On va y créer nos contrôleurs, un par fichier :

```php
<?php
// controllers/homepage.php

require_once(model/model.php');

function homepage() {
    $posts = getPosts();

    require('templates/homepage.php');
}
```

```php
<?php
// controllers/post.php

require_once('model/model.php');

function post(string $identifier)
{
    $post = getPost($identifier);
    $comments = getComments($identifier);

    require('templates/post.php');
}
```

On a apporté deux changements majeurs :

- **Nos contrôleurs sont placés dans des fonctions**. Chaque fichier devient du type "bibliothèque de code" et ne fait plus rien par lui-même. Il va simplement fournir à notre routeur un point d'accès pour lancer notre code.
- **Notre fichier model/model.php est inclus avec require_once**. C'est une fonction très semblable à require, mais qui vérifie d'abord si le fichier a déjà été inclus ! Étant donné que model/model.php est aussi un fichier de type "bibliothèque de code", on souhaite qu'il ne soit inclus qu'une seule fois. Sans ça, l'inclusion de nos deux contrôleurs va déclencher une double inclusion de notre modèle et donc un plantage de PHP.

Les plus attentifs auront aussi noté qu'on a simplifié le contrôleur post, en lui enlevant la responsabilité de chercher lui-même l'identifiant du billet dans la requête. On préfère déplacer ce travail sur notre routeur. Chaque contrôleur (et donc chaque nouvelle fonctionnalité métier) sera ainsi plus facile à développer.

IX.3. Créez le routeur index.php

Intéressons-nous maintenant à notre routeur index.php :

```php
<?php

require_once('controllers/homepage.php');
require_once('controllers/post.php');

if (isset($_GET['action']) && $_GET['action'] !== '') {
    if ($_GET['action'] === 'post') {
        if (isset($_GET['id']) && $_GET['id'] > 0) {
            $identifier = $_GET['id'];

            post($identifier);
        } else {
            echo 'Erreur : aucun identifiant de billet envoyé';

            die;
        }
    } else {
        echo "Erreur 404 : la page que vous recherchez n'existe pas.";
    }
} else {
    homepage();
}
```

Il a l'air un peu compliqué parce qu'on y fait pas mal de tests, mais le principe est tout simple : **appeler le bon contrôleur**. Ça donne :

1. On charge nos fichiers de contrôleurs controllers/homepage.php et controllers/post.php (pour que les fonctions soient en mémoire, quand même !).
2. On teste le paramètre action pour savoir quel contrôleur appeler. Si le paramètre n'est pas présent, on charge le contrôleur de la page d'accueil contenant la liste des derniers billets (ligne 21).
3. On teste les différentes valeurs possibles pour notre paramètre action et on redirige vers le bon contrôleur à chaque fois.

Il nous reste une petite modification à faire pour que notre blog soit de nouveau utilisable.

On a changé les URL auxquelles répondait notre application, mais on n'est pas encore repassé sur les liens de l'interface.

Pour cela, dans templates/homepage.php, modifiez la ligne 14 pour qu'elle prenne en compte notre paramètre action:

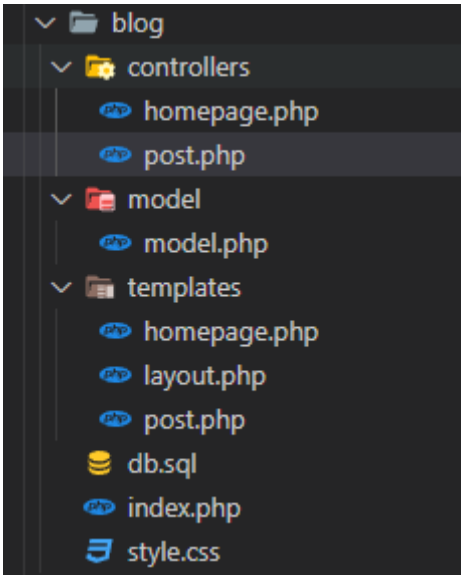
```
<em><a href="index.php?action=post&id=<?= urlencode($post['identifier'])
?>">Commentaires</a></em>
```

IX.4. En résumé

- Le routeur est un composant du code qui a pour rôle de recevoir toutes les requêtes de l'application et de router chacune vers le bon contrôleur.
- On préfère créer un fichier par contrôleur, tous rassemblés dans un même dossier. À l'intérieur, chaque fichier définit une fonction, qui sera appelée par le routeur.
- Quand on fonctionne avec des fichiers PHP de type "bibliothèque de code", il faut utiliser require_once pour éviter des plantages.

X. Organisez en dossiers

On commence à avoir un sacré nombre de fichiers entre tous nos dossiers !



Si on continue à ajouter des pages, on va se retrouver avec de nouveaux contrôleurs, de nouvelles vues... ça va vite devenir compliqué de s'y retrouver ! Il nous faut des moyens d'organiser notre code **à volonté.**

X.1. Identifiez le métier

En ajoutant des fonctionnalités, les fichiers vont se multiplier. Cela veut-il dire que l'on aura 200 fichiers dans le dossier controllers ?

Bien sûr que non, on ne va pas laisser ça se produire. Ce serait invivable – et je vous parle d'expérience... Il va falloir qu'on apprenne à diviser notre code selon le **métier**.

Le métier, c'est toute la partie de votre projet qui n'est pas la technique. Certes, ça laisse encore du flou, mais c'est déjà pas mal.

Prenons deux exemples :

- Une connexion à une base de données MySQL, c'est une **notion technique**. Il n'y a que le développeur qui la maîtrise. C'est d'ailleurs lui (ou l'architecte avant lui), qui fait le choix de la mettre en place. Et souvent, si vous en parlez à votre client, ça risque d'être du charabia pour lui.
- Afficher des commentaires sous un billet de blog, de l'autre côté, c'est une **notion métier**. La plupart du temps, c'est le client qui introduit ce type de notion dans le projet.

Chaque notion métier est bien souvent supportée par plusieurs notions techniques. Rien que pour mettre en ligne une page qui afficherait Hello, world! (une demande métier, donc), on a besoin d'Apache, de PHP, de HTML... Tout ça, ce sont des notions techniques.

X.2. Regroupez par sections du site

Pour pouvoir segmenter votre code à l'infini, sachez que les professionnels recommandent en général de le segmenter selon des **sections du métier**. Finis, les dossiers qui contiennent toutes vos requêtes SQL !

Et bonus pour vous, il vous suffira **d'écouter votre interlocuteur** pour définir votre segmentation ! Projetez-vous un instant dans cette demande du client :

Bonjour,

Pourriez-vous nous ajouter, s'il vous plaît, une page affichant un formulaire de contact sur le front ?

Nous nous tenons disponible pour en discuter plus en détail.

Où est-ce que vous placeriez votre contrôleur ?

Ma proposition, dans ce cas, serait de le mettre dans controllers/ (on garde bien sûr notre architecture MVC). Puis, à l'intérieur de ce dossier, je le placerais dans un fichier front/contact.php: dans la section front, un formulaire de contact. Au total : controllers/front/contact.php.

En fait, pour organiser ses demandes, le client a déjà conçu une segmentation métier. Parfois elle est un peu floue et il faut que vous la précisiez. Parfois elle est très claire et il ne reste qu'à traduire. Ce qui est sûr, c'est que quand surgira la prochaine demande de modification, le client utilisera les mêmes mots pour décrire ce qu'il veut. Vous n'aurez qu'à les suivre pour retrouver le code responsable de la fonctionnalité.

Pour les modèles et les vues, on va pouvoir utiliser exactement la même typologie de segmentation. Pas obligatoirement la même arborescence, parce que parfois une arborescence de contrôleurs est plus simple qu'une arborescence de vue ou de modèle. Ou l'inverse. Mais elles se ressembleront fortement.

Dans notre cas, on peut créer les fichiers suivants :

templates/front/contact.php: notre vue, qui est directement liée à notre contrôleur et qui a exactement la même arborescence.

model/contact.php: notre modèle qui sera responsable de sauvegarder les données de contact dans ma base de données, puis de les lire depuis celle-ci. La notion de contact est globale à notre blog (on aura sans doute une section "back-office" pour les traiter). La segmentation front n'ayant donc pas de sens ici, on ne met pas le dossier front/dans le chemin complet de ce fichier.

Il y a toujours des tas de possibilités pour segmenter son métier. Faites de votre mieux pour écouter le client, mais faites surtout de votre mieux. Je vous parle de mes propositions à chaque fois, parce qu'une solution différente pourrait être encore plus pertinente pour votre équipe.

On peut même parler de "sensibilité" pour ce genre de prises de décisions. Vous avez toute votre vie de développeur pour évoluer là-dessus !

XI. Ajoutez des commentaires

Maintenant, nous devons ajouter la possibilité pour les visiteurs d'ajouter des commentaires sur les billets.

Niveau organisation, nous allons faire les choses dans cet ordre :

1. Modifier la vue, pour afficher le formulaire. Il a deux champs : l'auteur et le commentaire.
2. Écrire le nouveau contrôleur, qui traite les données envoyées via le formulaire.
3. Mettre à jour le routeur, pour envoyer vers le bon contrôleur.
4. Écrire le modèle. À chaque commentaire ajouté, sa date de création est sauvegardée en base automatiquement. On va aussi en profiter pour refactoriser le modèle selon le métier.

XI.1. Mettez à jour la vue

Il faut commencer par modifier un peu la vue qui affiche un billet et ses commentaires (templates/post.php). En effet, nous devons ajouter le formulaire pour pouvoir envoyer des commentaires !

```php
<!-- templates/post.php:18 -->
<h2>Commentaires</h2>

<form action="index.php?action=addComment&id=<?= $post['identifier'] ?>"
method="post">
    <div>
        <label for="author">Auteur</label><br />
        <input type="text" id="author" name="author" />
    </div>
    <div>
        <label for="comment">Commentaire</label><br />
        <textarea id="comment" name="comment"></textarea>
    </div>
    <div>
        <input type="submit" />
    </div>
</form>
```

Il faut juste bien écrire l'URL vers laquelle le formulaire est censé envoyer. Ici, vous voyez que l'on envoie vers une action addComment. Il faudra bien penser à mettre à jour le routage.

Vous voyez aussi qu'on a besoin de $post['identifier'] que nous n'avions pas récupéré jusque-là. On va modifier notre modèle, à la ligne 32, pour aller la chercher depuis la base de données :

```php
    // model/model.php:32
    $post = [
        'title' => $row['title'],
        'french_creation_date' => $row['french_creation_date'],
        'content' => $row['content'],
        'identifier' => $row['id'],
    ];
    // ...
```

XI.2. Écrivez le contrôleur

L'écriture de notre nouveau contrôleur va être un peu plus conséquente. Le contrôleur va **contrôler** les données soumises par l'utilisateur via le formulaire. C'est un travail rigoureux : on peut par exemple vouloir vérifier le nombre de caractères dans le commentaire ou bien si le nom de l'auteur ne contient pas de caractères invalides. Ici on fera une version qui contient le strict minimum.

Le contrôleur va prendre en paramètres les deux entrées utilisateur :

- **L'identifiant du billet** auquel le commentaire doit être associé. Ce sera une chaîne de caractères.
- **Les données soumises par le formulaire**, sous la forme d'un tableau associatif de chaînes de caractères. Ça sera pratique, car c'est le format que PHP utilise dans $_POST.

Créons ce nouveau fichier controllers/add_comment.php :

```php
<?php
// controllers/add_comment.php

require_once('model/comment.php');

function addComment(string $post, array $input)
{
    $author = null;
    $comment = null;
    if (!empty($input['author']) && !empty($input['comment'])) {
        $author = $input['author'];
        $comment = $input['comment'];
    } else {
        die('Les données du formulaire sont invalides.');
    }

    $success = createComment($post, $author, $comment);
    if (!$success) {
        die('Impossible d\'ajouter le commentaire !');
    } else {
        header('Location: index.php?action=post&id=' . $post);
    }
}
```

Vous noterez qu'on teste le résultat renvoyé par notre modèle. Écrire en base de données est une opération qui peut échouer, alors on demandera à notre modèle de renvoyer true en cas de succès ou false en cas d'échec. Pour résumer : on teste s'il y a eu une erreur et on arrête tout (avec un die) si jamais il y a un souci.

Si tout va bien, il n'y a aucune page à afficher. Les données ont été insérées, on redirige donc le visiteur vers la page du billet pour qu'il puisse voir son beau commentaire qui vient d'être inséré !

XI.3. Mettez à jour le routeur

Il faut maintenant qu'on enregistre notre contrôleur au niveau de notre routeur. Ajoutons un elseif dans notre routeur (index.php) pour appeler le nouveau contrôleur addComment qu'on vient de créer et on devrait avoir tout bon !

```php
<?php
// index.php

require_once('controllers/add_comment.php');
require_once('controllers/homepage.php');
require_once('controllers/post.php');

if (isset($_GET['action']) && $_GET['action'] !== '') {
    if ($_GET['action'] === 'post') {
        if (isset($_GET['id']) && $_GET['id'] > 0) {
            $identifier = $_GET['id'];

            post($identifier);
        } else {
            echo 'Erreur : aucun identifiant de billet envoyé';
            die;
        }
```

```
    } elseif ($_GET['action'] === 'addComment') {
        if (isset($_GET['id']) && $_GET['id'] > 0) {
            $identifier = $_GET['id'];

            addComment($identifier, $_POST);
        } else {
            echo 'Erreur : aucun identifiant de billet envoyé';

            die;
        }
    } else {
        echo "Erreur 404 : la page que vous recherchez n'existe pas.";
    }
} else {
    homepage();
}
```

Il devient dur à lire ce routeur…

C'est vrai qu'avec tous ces if imbriqués, ça commence à faire beaucoup... Mais il n'y a pas trop le choix. Ceci dit, il y a une meilleure façon de gérer les erreurs, nous verrons cela par la suite.

Comme vous pouvez le voir, je teste si on a bien un ID de billet. Si c'est le cas, j'appelle le contrôleur addComment, qui appelle le modèle pour enregistrer les informations en base. Pour la validation des champs du formulaire, on a donné cette responsabilité au contrôleur, alors on lui passe la variable $_POST directement.

XI.4. Écrivez le modèle

On y est presque ! Il nous reste à créer notre nouveau fichier modèle. Je vous propose qu'on segmente autour de la notion de "commentaire" sur les billets. On va donc créer un nouveau fichier model/comment.php, où vous allez mettre la nouvelle fonction createComment(). On va aussi en profiter pour y déplacer la fonction getComments() :

```
<?php
// model/comment.php

function getComments(string $post)
{
    $database = commentDbConnect();
    $statement = $database->prepare(
        "SELECT id, author, comment, DATE_FORMAT(comment_date, '%d/%m/%Y à
%Hh%imin%ss') AS french_creation_date FROM comments WHERE post_id = ? ORDER BY
comment_date DESC"
    );
    $statement->execute([$post]);

    $comments = [];
    while (($row = $statement->fetch())) {
        $comment = [
            'author' => $row['author'],
            'french_creation_date' => $row['french_creation_date'],
            'comment' => $row['comment'],
        ];

        $comments[] = $comment;
```

```
    }

    return $comments;
}

function createComment(string $post, string $author, string $comment)
{
    $database = commentDbConnect();
    $statement = $database->prepare(
        'INSERT INTO comments(post_id, author, comment, comment_date) VALUES(?, ?,
?, NOW())'
    );
    $affectedLines = $statement->execute([$post, $author, $comment]);

    return ($affectedLines > 0);
}

function commentDbConnect()
{
    try {
        $database = new PDO('mysql:host=localhost;dbname=blog;charset=utf8',
'root', '');

        return $database;
    } catch(Exception $e) {
        die('Erreur : '.$e->getMessage());
    }
}
```

Rien de bien sorcier. Il faut juste penser à récupérer en paramètres les informations dont on a besoin :

- l'ID du billet auquel se rapporte le commentaire ;
- le nom de l'auteur ;
- le contenu du commentaire.

Le reste des informations (l'ID du commentaire, la date) sera généré automatiquement.

Au fait, nous avons refactorisé la manière de récupérer nos commentaires, mais nous n'avons pas encore mis à jour le contrôleur controllers/post.php. Il faut qu'on ajoute un require_once :

```
<?php
// controllers/post.php

require_once('model/model.php');
require_once('model/comment.php');

function post(string $identifier)
{
    $post = getPost($identifier);
    $comments = getComments($identifier);

    require('templates/post.php');
}
```

Remarque : les fichiers model/model.php et model/comment.php. Ensuite, on a dupliqué la fonction dbConnect(), ce n'est pas idéal…

Tout d'abord, le premier point. Ce n'est pas terrible. Cependant, quand on travaille sur un projet, on doit en permanence faire des concessions. Ici, on ne touche pas aux billets de blog. D'ailleurs, la plupart du temps, on ne se souviendra sans doute plus de comment ce code fonctionne.

On fait donc le choix de construire à côté de l'existant. Quand on souhaitera refactoriser notre gestion des billets de blog, on le fera dans un autre commit. C'est une des grandes forces de la segmentation métier du code !

Pour le second point, c'est un peu plus complexe. D'une part, l'unité de code "Comment" est différente de l'unité de code "Post". Ce qui affecte l'un ne doit pas forcément affecter l'autre. Ici, on peut considérer comme une coïncidence que la base de données utilisée soit la même entre ces deux bouts de code : par exemple, on pourrait très bien stocker nos commentaires via une API spécialisée.

Pour autant, ici toute la partie configuration (nom de la base, mot de passe) est ici dupliquée. Et ça, ce n'est pas une question de coïncidence. C'est une faiblesse dans notre code.

XI.5. En résumé

- Notre blog a maintenant une nouvelle fonctionnalité : permettre à l'utilisateur d'ajouter des commentaires aux billets.
- Une segmentation métier du code est maintenant mise en place. La notion de "commentaire" est bien isolée et plus facile à retrouver.

XII. Gérez les erreurs

La gestion des erreurs est un sujet important en programmation. Il y a souvent des erreurs et il faut savoir vivre avec. Mais comment faire ça bien ?

Si vous vous souvenez de notre routeur, il contient beaucoup de if. On fait des tests et on affiche des erreurs à chaque fois qu'il y a un problème :

```php
<?php

if (test) {
    // C'est bon, on fait des choses
    // ...

    if (encoreUnTest) {
     // C'est bon, on continue
    } else {
     echo 'Erreur';
    }
} else {
    echo 'Autre erreur';
}
```

Ça marche, mais comme toujours, ce n'est pas parce que ça marche que c'est pratique à la longue. Les développeurs ont en particulier du mal à gérer comme ça les erreurs qui ont lieu **à l'intérieur** des fonctions.

Que se passe-t-il s'il y a une erreur dans le contrôleur ou dans le modèle ? Va-t-on les laisser se charger d'afficher des erreurs ? Ça ne devrait normalement pas être à eux de le faire. Ils devraient **remonter** qu'il y a une erreur et laisser une partie spécialisée du code **traiter l'erreur**.

XII.1. Tirez parti des exceptions

Les exceptions sont un moyen en programmation de gérer les erreurs. Vous en avez déjà vu dans du code PHP :

```php
<?php

try {
   // Essayer de faire quelque chose
} catch (Exception $e) {
   // Si une erreur se produit, on arrive ici
}
```

En premier lieu, l'ordinateur essaie d'exécuter les instructions qui se trouvent dans le bloc try. Deux possibilités :

- soit il ne se passe aucune erreur dans le bloc try : dans ce cas, on saute le bloc catch et on passe à la suite du code ;
- soit une erreur se produit dans le bloc try : on arrête ce qu'on faisait et on va directement dans le catch (pour "attraper" l'erreur).

C'est par exemple ce qu'on fait pour se connecter à la base de données.

Et on peut afficher l'erreur qui nous a été envoyée avec $e->getMessage().

Pour générer une erreur, il faut "jeter une exception", ou "lancer une exception". Dès qu'il y a une erreur quelque part dans votre code, dans une fonction par exemple, on utilisera cette ligne :

```php
<?php
throw new Exception('Message d\'erreur à transmettre');
```

XII.2. Ajoutez la gestion des exceptions dans le routeur

Nous allons entourer tout notre routeur par un bloc try/catch comme ceci :

```php
<?php
// index.php

require_once('controllers/add_comment.php');
require_once('controllers/homepage.php');
require_once('controllers/post.php');

try {
    if (isset($_GET['action']) && $_GET['action'] !== '') {
        if ($_GET['action'] === 'post') {
            if (isset($_GET['id']) && $_GET['id'] > 0) {
                $identifier = $_GET['id'];

                post($identifier);
            } else {
```

```
                throw new Exception('Aucun identifiant de billet envoyé');
            }
        } elseif ($_GET['action'] === 'addComment') {
            if (isset($_GET['id']) && $_GET['id'] > 0) {
                $identifier = $_GET['id'];

                addComment($identifier, $_POST);
            } else {
                throw new Exception('Aucun identifiant de billet envoyé');
            }
        } else {
            throw new Exception("La page que vous recherchez n'existe pas.");
        }
    } else {
        homepage();
    }
} catch (Exception $e) { // S'il y a eu une erreur, alors...
    echo 'Erreur : '.$e->getMessage();
}
```

Comme vous pouvez le voir, à l'endroit où les erreurs se produisent se trouvent des throw new Exception. Cela arrête le bloc try et amène directement l'ordinateur au bloc catch.

Ici, notre bloc catch se contente de récupérer le message d'erreur qu'on a transmis et de l'afficher.

XII.3. Remontez les exceptions

Pour l'instant, vous vous dites sûrement que ça n'est pas fou fou. Ok, les exceptions sont faites pour gérer les erreurs, mais on a surtout compliqué le code du routeur avec un nouveau bloc.

C'est parce que vous n'avez pas encore vu **à quel point les exceptions peuvent être pratiques !** Quand il se passe une erreur à l'intérieur d'une fonction située dans le bloc try, **celle-ci est "remontée" jusqu'au bloc catch**.

Par exemple, notre routeur appelle la fonction du contrôleur addComment. Que se passe-t-il quand il y a une erreur dans le contrôleur ? Pour l'instant, on fait ça :

```
require_once('model/comment.php');

function addComment(string $post, array $input)
{
    $author = null;
    $comment = null;
    if (!empty($input['author']) && !empty($input['comment'])) {
        $author = $input['author'];
        $comment = $input['comment'];
    } else {
        die('Les données du formulaire sont invalides.');
    }

    $success = createComment($post, $author, $comment);
    if (!$success) {
        die('Impossible d\'ajouter le commentaire !');
    } else {
        header('Location: index.php?action=post&id=' . $post);
    }
}
```

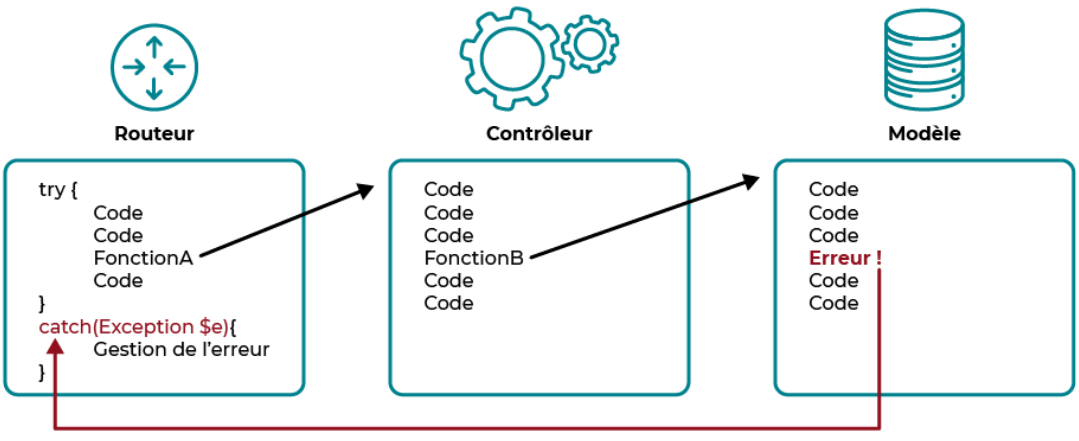
Notre contrôleur arrête tout et affiche ses erreurs avec des die. Il y a moyen de faire plus propre : jetons ici des exceptions, le code s'y arrêtera, et **les erreurs seront remontées jusque dans le routeur qui contenait le bloc try** !

Voilà comment on peut mieux gérer les erreurs, en ajoutant des throw :

```php
<?php
// controllers/add_comment.php

require_once('model/comment.php');

function addComment(string $post, array $input)
{
    $author = null;
    $comment = null;
    if (!empty($input['author']) && !empty($input['comment'])) {
        $author = $input['author'];
        $comment = $input['comment'];
    } else {
        throw new Exception('Les données du formulaire sont invalides.');
    }

    $success = createComment($post, $author, $comment);
    if (!$success) {
        throw new Exception('Impossible d\'ajouter le commentaire !');
    } else {
        header('Location: index.php?action=post&id=' . $post);
    }
}
```

Ce principe de "remontée" de l'erreur jusqu'à l'endroit du code qui contenait le bloc try est un gros avantage des exceptions

Comme ce n'est pas forcément évident à voir comme ça, voici un schéma qui résume le concept :



Du coup, dans la fonction dbConnect() de notre modèle, il n'est plus forcément nécessaire de garder un bloc try/catch. L'erreur de connexion à la base, s'il y en a une, sera remontée jusqu'au routeur :

```
function dbConnect()
{
    $database = new PDO('mysql:host=localhost;dbname=blog;charset=utf8', 'root',
'');

    return $database;
}
```

XII.4. Exercez-vous

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 templates/error.php qui affiche joliment le message d'erreur.

Il faudrait faire quelque chose dans ce goût-là :

```
// index.php

require_once('controllers/add_comment.php');
require_once('controllers/homepage.php');
require_once('controllers/post.php');

try {
    // ...
} catch (Exception $e) {
    $errorMessage = $e->getMessage();

    require('templates/error.php');
}
```

Essayez de créer la vue vous-même, vous avez compris le concept.

Sinon, voici une proposition simple :

```
<?php $title = "Le blog de l'AVBN"; ?>

<?php ob_start(); ?>
<h1>Le super blog de l'AVBN !</h1>
<p>Une erreur est survenue : <?= $errorMessage ?></p>
<?php $content = ob_get_clean(); ?>

<?php require('layout.php') ?>
```

XII.5. En résumé

- Les exceptions sont un mécanisme qui permet de gérer les remontées d'erreurs en PHP. Elles s'utilisent avec les trois mots clés :try, catch et throw.
- try et catch permettent de créer des zones de contrôles sur les exceptions.
- throw permet de lancer une exception. Elle interrompra le flux classique d'exécution du code, jusqu'à être gérée par une zone de contrôle d'exception.
- Les zones de contrôle d'exception peuvent s'imbriquer à l'infini.