

# Evolution d'un projet PHP simple vers le modèle MVC

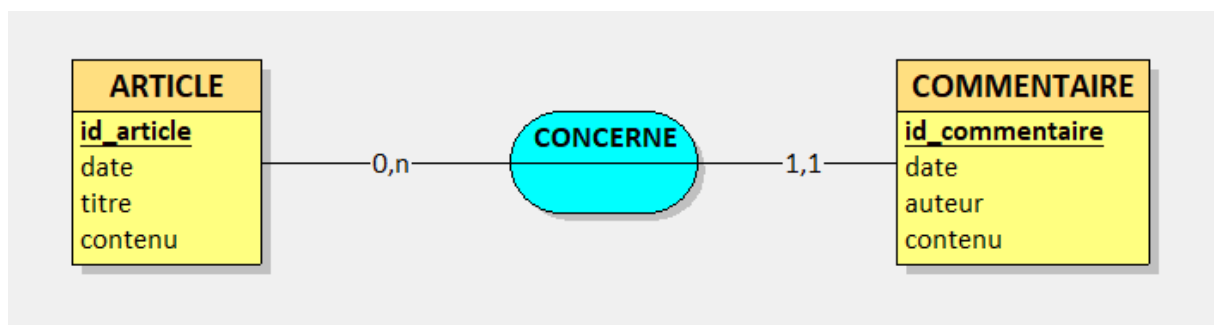
Le projet que nous allons faire évoluer vers un modèle MVC est une petite application de blog en PHP avec une base de données MySQL, qui permet aux utilisateurs de créer des articles, puis de les commenter.

## 1. Première version de l'application

### 1.1. La base de données

La base de données du projet est très simple. Elle comporte 2 tables, articles et commentaires :

*Voici le MCD de cette base de données :*



*le MLD :*

**ARTICLE** = (id\_article INT, date\_ DATETIME, titre VARCHAR(50), contenu TEXT);  
**COMMENTAIRE** = (id\_commentaire INT, date\_ DATETIME, auteur VARCHAR(50), contenu TEXT, #id\_article);

*et enfin le SQL de création, y compris l'insertion de quelques données de test :*

```

-- Création de la base de données
DROP DATABASE IF EXISTS blog;
CREATE DATABASE blog CHARACTER SET utf8 COLLATE utf8_general_ci;

-- Connexion à la base de données
USE blog;

-- Création de la table "article"
DROP TABLE IF EXISTS article;
CREATE TABLE ARTICLE(
    id INT NOT NULL AUTO_INCREMENT,
    titre VARCHAR(50) NOT NULL,
    date DATETIME NOT NULL,
    contenu TEXT NOT NULL,
    PRIMARY KEY(id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

-- Création de la table "commentaire"
DROP TABLE IF EXISTS commentaire;
CREATE TABLE commentaire(
    id INT NOT NULL AUTO_INCREMENT,
    date DATETIME NOT NULL,
  
```

```

    auteur VARCHAR(50) NOT NULL,
    contenu TEXT NOT NULL,
    id_article INT NOT NULL,
    PRIMARY KEY(id),
    FOREIGN KEY(id_article) REFERENCES ARTICLE(id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

-- Insertion de données dans la table "article"
INSERT INTO article(date, titre, contenu) VALUES
(NOW(), 'Premier article, 'Bonjour monde ! Ceci est le premier article sur
mon blog.');
```

```

INSERT INTO article(date, titre, contenu) VALUES
(NOW(), 'Au travail', 'Il faut enrichir ce blog dès maintenant.');
```

```

-- Insertion de données dans la table "commentaire"
INSERT INTO commentaire(date, auteur, contenu, id_article) values
(NOW(), 'A. Nonyme', 'Bravo pour ce début', 1);
INSERT INTO commentaire(date, auteur, contenu, id_article) values
(NOW(), 'Moi', 'Merci ! Je vais continuer sur ma lancée', 1);
```

----- V1 -----

## 1.2. Page principale du blog (v1)

La page principale du projet (index.php) est écrite en HTML5 et PHP. On utilise PDO pour accéder à la base de données et une feuille de style (style.css) suffira à améliorer le design. On utilise l'affichage abrégé PHP `< ?= ... ?>` plutôt que `< ?php echo ... ?>`, ainsi que la syntaxe alternative dans la boucle foreach

### 1.2.1 Fichier index.php

```

<!doctype html>
<html lang="fr">
<head>
    <meta charset="utf-8" />
    <link rel="stylesheet" href="style.css" />
    <title>Mon Blog</title>
</head>
<body>
    <div id="global">
        <header>
            <a href="index.php"><h1 id="titreBlog">Mon Blog</h1></a>
            <p>Je vous souhaite la bienvenue sur ce modeste blog.</p>
        </header>
        <div id="contenu">
            <?php
                $bdd = new PDO('mysql:host=localhost;dbname=blog;charset=utf8',
                'root', '');
                $articles = $bdd->query('select id, date, titre, contenu from
                article order by id desc');
                foreach ($articles as $article): ?>
                    <article>
                        <header>
```

```

        <h1 class="titreArticle">
            <?= $article['titre'] ?>
        </h1>
        <time><?= $article['date'] ?></time>
    </header>
    <p><?= $article['contenu'] ?></p>
</article>
<hr />
<?php endforeach; ?>
</div> <!-- #contenu -->
<footer id="piedBlog">
    Blog réalisé avec PHP, HTML5 et CSS.
</footer>
</div> <!-- #global -->
</body>

```

### 1.2.2 Fichier style.css

```

/* Pour pouvoir utiliser une hauteur (height) ou une hauteur minimale
(min-height) sur un bloc, il faut que son parent direct ait lui-même une
hauteur déterminée (donc toute valeur de height sauf "auto": hauteur en
pixels, em, autres unités...).
Si la hauteur du parent est en pourcentage, elle se réfère alors à la
hauteur du «grand-père», et ainsi de suite.
Pour pouvoir utiliser un "min-height: 100%" sur div#global, il nous faut:
- un parent (body) en "height: 100%";
- le parent de body également en "height: 100%". */
html, body {
    height: 100%;
}
body {
    color: #bfbfbf;
    background: black;
    font-family: 'Futura-Medium', 'Futura', 'Trebuchet MS', sans-serif;
}
h1 {
    color: white;
}
a {
    color: white;
}
.titreArticle {
    margin-bottom : 0px;
}
#global {
    min-height: 100%; /* Voir commentaire sur html et body plus haut */
    background: #333534;
    width: 70%;
    margin: auto; /* Permet de centrer la div */
}

```

```

    text-align: justify;
    padding: 5px 20px;
}
#contenu {
    margin-bottom : 30px;
}
#titreBlog, #piedBlog {
    text-align: center;
}

```

Le résultat dans un navigateur :



### 1.3. La problématique de cette première version

La page index.php mélange HTML et PHP, ce qui nuit à la compréhension et à la maintenance. De plus, sa conception monobloc rend difficile la réutilisation des parties du code qui pourraient être employées dans d'autres fichiers

Une application doit le plus souvent gérer les 3 problématiques suivantes :

- *la présentation* : c'est l'ensemble des interactions avec l'extérieur du programme, affichage, saisie de l'utilisateur et contrôle ;
- *les traitements* : opérations sur les données, en rapport avec la logique métier ;
- *les données* : accès et stockages des informations manipulées par le programme.

Cette première version mélange *présentation* (balises HTML) et *accès aux données* (requêtes SQL), ce qui est contraire au principe de *responsabilité unique*. Selon ce principe, l'application doit être décomposées en sous-parties, chacune responsable d'une problématique unique. L'application de ce principe permet de maintenir la complexité et le volume du code dans des valeurs raisonnables et favorise la compréhension ainsi que la maintenabilité du programme.

## 2. Evolution vers une architecture MVC simple

----- V2 -----

### 2.1 Isolation de l'affichage (v2)

```

<!-- Accès aux données-->
<?php
    $bdd = new PDO('mysql:host=localhost;dbname=blog;charset=utf8', 'root',
    '');
    $articles = $bdd->query('select id, date, titre, contenu from article
    order by id desc');
?>

<!-- Affichage -->
<!doctype html>
<html lang="fr">
<head>
    ...
</head>
    ...
    <div id="contenu">
        <?php foreach ($articles as $article): ?>
            <article>
                <header>
                    <h1 class="titreArticle">
                        <?= $article['titre'] ?>
                    </h1>
                    <time>
                        <?= $article['date'] ?>
                    </time>
                </header>
                <p>
                    <?= $article['contenu'] ?>
                </p>
            </article>
            <hr />
        <?php endforeach; ?>
    </div> <!-- #contenu -->

```

Le code devient plus lisible, mais les problématiques de données et d'affichage sont toujours gérées au sein du même fichier PHP.

On peut aller plus loin dans le découplage en déplaçant le code d'affichage dans un fichier dédié : vueAccueil.php

----- V3 -----

### Fichier vueAccueil.php (v3)

```

<!doctype html>
<html lang="fr">

```

```

<head>
...
</head>
<body>
...
<div id="contenu">
    <?php foreach ($articles as $article): ?>
        <article>
            ...
        </article>
        <hr />
    <?php endforeach; ?>
</div> <!-- #contenu -->
...
</body>
</html>

```

Fichier index.php

```

<?php
// Accès aux données
$bdd = new PDO('mysql:host=localhost;dbname=blog;charset=utf8', 'root',
'');
$articles = $bdd->query('select id, date, titre, contenu from article order
by id desc');

// Affichage
require './vueAccueil.php';

```

Rappel : la fonction PHP require fonctionne de manière similaire à include : elle inclut et exécute le fichier spécifié. En cas d'échec, include ne produit qu'un avertissement alors que require stoppe le script.

*La balise de fin de code PHP ?> est volontairement omise à la fin du fichier index.php. C'est une bonne pratique pour les fichiers qui ne contiennent que du PHP. Elle permet d'éviter des problèmes lors d'inclusions de fichiers.*

----- V4 -----

## 2.1 Isolation de l'accès aux données (v4)

On peut avantageusement augmenter la modularité du code en isolant le code d'accès aux données dans un fichier séparé

Fichier modele.php

```

<?php
// Renvoie la liste de tous les articles, triés par identifiant décroissant
function getArticles()
{
    $bdd = new PDO('mysql:host=localhost;dbname=blog;charset=utf8', 'root',
'');

```

```

    $articles = $bdd->query('select id, date, titre, contenu from article
    order by id desc');
    return $articles;
}

```

Dans ce fichier, c'est la fonction `getArticles()` qui se charge de récupérer la liste des articles

Le fichier `index.php` est maintenant simplifié au maximum et s'appuie sur les services fournis par les fichiers `vueAccueil.php` et `modele.php`

Fichier `index.php` (v4)

```

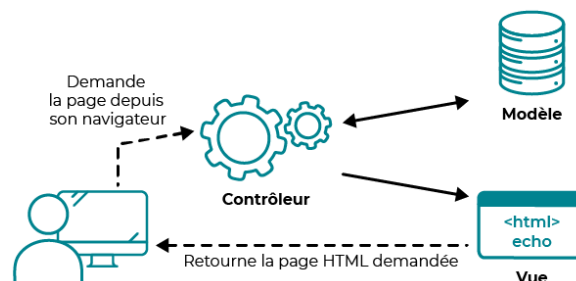
<?php
// Accès aux données
require './modele.php';
$articles = getArticles();

// Affichage
require './vueAccueil.php';

```

## 2.3 Le modèle MVC

Dans cette dernière version, le code est séparé en 3 fichiers (`index.php`, `vueAccueil.php` et `modele.php`) qui ont chacun une responsabilité précise et unique. On appelle ce modèle de conception « *MVC* » pour *Modèle-Vue-Contrôleur*



1. La demande de l'utilisateur (exemple : requête HTTP) est reçue et interprétée par le Contrôleur.
2. Celui-ci utilise les services du Modèle afin de préparer les données à afficher.
3. Ensuite, le Contrôleur fournit ces données à la Vue, qui les présente à l'utilisateur (par exemple sous la forme d'une page HTML).

Une application construite sur le principe du MVC se compose toujours de trois parties distinctes. Cependant, il est fréquent que chaque partie soit elle-même décomposée en plusieurs éléments. On peut ainsi trouver plusieurs modèles, plusieurs vues ou plusieurs contrôleurs à l'intérieur d'une application MVC.



## 2.4 Améliorations supplémentaires (v5)

Un site Web se réduit rarement à une seule page. Il serait donc souhaitable de définir à un seul endroit les éléments communs des pages HTML affichées à l'utilisateur (les vues).

Pour cela, nous allons utiliser un modèle de page (« gabarit », en anglais « template »). Ce modèle contiendra tous les éléments communs et permettra d'ajouter les éléments spécifiques à chaque vue.

### 2.4.1 Fichier gabarit.php

```
<!doctype html>
<html lang="fr">

<head>
    <meta charset="UTF-8" />
    <link rel="stylesheet" href="style.css" />
    <title>
        <?= $titre ?>
    </title> <!-- Élément spécifique -->
</head>

<body>
    <div id="global">
        <header>
            <a href="index.php">
                <h1 id="titreBlog">Mon Blog</h1>
            </a>
            <p>Je vous souhaite la bienvenue sur ce modeste blog.</p>
        </header>
        <div id="contenu">
            <?= $contenu ?> <!-- Élément spécifique -->
        </div>
        <footer id="piedBlog">
            Blog réalisé avec PHP, HTML5 et CSS.
        </footer>
    </div> <!-- #global -->
</body>
```

Au moment de l'affichage d'une vue HTML, il suffit de définir les valeurs des éléments spécifiques, puis de déclencher le rendu de notre gabarit. Pour cela, on utilise des fonctions PHP qui manipulent le flux de sortie de la page. Voici notre page `vueAccueil.php` réécrite :

### Fichier `vueAccueil.php` (v5)

```
<?php $titre = 'Mon Blog'; ?>
<?php ob_start(); ?>
<?php foreach ($articles as $article): ?>
    <article>
        <header>
            <h1 class="titreArticle"><?= $article['titre'] ?></h1>
```

```

        <time><?= $article['date'] ?></time>
    </header>
    <p><?= $article['contenu'] ?></p>
</article>
<hr />
<?php endforeach; ?>
<?php $contenu = ob_get_clean(); ?>
<?php require 'gabarit.php'; ?>

```

1. La première ligne définit la valeur de l'élément spécifique \$titre ;
2. La deuxième ligne utilise la fonction PHP ob\_start. Son rôle est de déclencher la mise en tampon du flux HTML de sortie : au lieu d'être envoyé au navigateur, ce flux est stocké en mémoire ;
3. La suite du code (boucle foreach) génère les balises HTML article associées aux articles du blog. Le flux HTML créé est mis en tampon ;
4. Une fois la boucle terminée, la fonction PHP ob\_get\_clean permet de récupérer dans une variable le flux de sortie mis en tampon depuis l'appel à ob\_start. La variable se nomme ici \$contenu, ce qui permet de définir l'élément spécifique associé ;
5. Enfin, on déclenche le rendu du gabarit. Lors du rendu, les valeurs des éléments spécifiques \$titre et \$contenu seront insérés dans le résultat HTML envoyé au navigateur.

L'affichage utilisateur est strictement le même qu'avant l'utilisation d'un gabarit. Cependant, nous disposons maintenant d'une solution souple pour créer plusieurs vues tout en centralisant la définition de leurs éléments communs.

#### 2.4.2 Factorisation de la connexion à la base de données

On peut améliorer l'architecture de la partie Modèle en isolant le code qui établit la connexion à la base de données sous la forme d'une fonction getBdd ajoutée dans le fichier modele.php. Cela évitera de dupliquer le code de connexion lorsque nous ajouterons d'autres fonctions au Modèle.

Fichier modele.php (v5)

```

<?php
// Renvoie la liste de tous les articles, triés par identifiant décroissant
function getArticles()
{
    $bdd = getBdd();
    $articles = $bdd->query('select id, date, titre, contenu from article
    order by id desc');
    return $articles;
}

// Effectue la connexion à la BDD
// Instancie et renvoie l'objet PDO associé
function getBdd()
{
    $bdd = new PDO('mysql:host=localhost;dbname=blog;charset=utf8', 'root',
    '');
    return $bdd;
}

```

## V6

### 2.4.3 Gestion des erreurs (v6)

Par souci de simplification, nous avons mis de côté la problématique de la gestion des erreurs. Il est temps de s'y intéresser. Pour commencer, il faut décider quelle partie de l'application aura la responsabilité de traiter les erreurs qui pourraient apparaître lors de l'exécution. Ce pourrait être le Modèle, mais il ne pourra pas les gérer correctement à lui seul ni informer l'utilisateur. La Vue, dédiée à la présentation, n'a pas à s'occuper de ce genre de problématique.

Le meilleur choix est donc d'implémenter la gestion des erreurs au niveau du Contrôleur. Gérer la dynamique de l'application, y compris dans les cas dégradés, fait partie de ses responsabilités.

Nous allons tout d'abord modifier la connexion à la base de données afin que les éventuelles erreurs soient signalées sous la forme d'exceptions.

Fichier `modele.php` (v6)

```
...
    $bdd = new PDO('mysql:host=localhost;dbname=blog;charset=utf8', 'root',
        '', array(PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION));
    return $bdd;
...
```

Ajout de l'attribut `ATTR_ERRMODE` afin de forcer MySQL à retourner les erreurs rencontrées lors des opérations de base de données sous forme d'exceptions.

On peut ensuite ajouter à notre page une gestion minimaliste des erreurs de la manière suivante :

Fichier `index.php` (v6)

```
<?php
// Accès aux données
require './modele.php';
try {
    $articles = getArticles();
    // Affichage
    require './vueAccueil.php';
}
catch (Exception $e) {
    echo '<html><body>Erreur ! ' . $e->getMessage() . '</body></html>';
}
```

Le premier `require` inclut uniquement la définition d'une fonction et est placé en dehors du bloc `try`. Le reste du code est placé à l'intérieur de ce bloc. Si une exception est levée lors de son exécution, une page HTML minimale contenant le message d'erreur est affichée.

On peut souhaiter conserver l'affichage du gabarit des vues même en cas d'erreur. Il suffit de définir une vue `vueErreur.php` dédiée à leur affichage.

Fichier `vueErreur` (v6)

```
<?php $titre = 'Mon Blog'; ?>
<?php ob_start() ?>
```

```
<p>Une erreur est survenue :  
    <?= $msgErreur ?>  
</p>  
<?php $contenu = ob_get_clean(); ?>  
<?php require 'gabarit.php'; ?>
```







On modifie ensuite le contrôleur pour déclencher le rendu de cette vue en cas d'erreur.

Fichier index.php (V6)

```
<?php  
// Accès aux données  
require './modele.php';  
try {  
    $articles = getArticles();  
    // Affichage  
    require './vueAccueil.php';  
}  
catch (Exception $e) {  
    $msgErreur = $e->getMessage();  
    require 'vueErreur.php';  
}
```

#### 2.4.4 Bilan

Nous avons accompli sur notre page d'exemple un important travail de refactorisation qui a modifié son architecture en profondeur. Notre page respecte à présent un modèle MVC simple.

-  gabarit.php
-  index.php
-  modele.php
-  style.css
-  vueAccueil.php
-  vueErreur.php

### 3. Ajout de nouvelles fonctions

V7

#### 3.1. Affichage des détails d'un article

Afin de rendre notre contexte d'exemple plus réaliste, nous allons ajouter un nouveau besoin : le clic sur le titre d'un article du blog doit afficher sur une nouvelle page le contenu et les commentaires associés à cet article.

Commençons par ajouter dans notre modèle (fichier modele.php) les fonctions d'accès aux données dont nous avons besoin.

Fichier modele.php (V7)

```
...
// Renvoie les informations sur un article
function getArticle($idArticle)
{
    $bdd = getBdd();
    $article = $bdd->prepare('select id, date, titre, contenu from article
    where id=?');
    $article->execute(array($idArticle));
    if ($article->rowCount() == 1)
        return $article->fetch(); // Accès à la première ligne de résultat
    else
        throw new Exception("Aucun article ne correspond à l'identifiant
        '$idArticle'");
}

// Renvoie la liste des commentaires associés à un article
function getCommentaires($idArticle)
{
    $bdd = getBdd();
    $commentaires = $bdd->prepare('select id, date, auteur, contenu from
    commentaire where id_article=?');
    $commentaires->execute(array($idArticle));
    return $commentaires;
}
```

Nous créons ensuite une nouvelle vue vueArticle.php dont le rôle est d'afficher les informations demandées.

Fichier vueArticle.php (V7)

```
<?php $titre = "Mon Blog - " . $article['titre']; ?>
<?php ob_start(); ?>
<article>
    <header>
        <h1 class="titreArticle"><?= $article['titre'] ?></h1>
        <time><?= $article['date'] ?></time>
```

```

    </header>
    <p><?= $article['contenu'] ?></p>
</article>
<hr />
<header>
    <h1 id="titreReponses">Réponses à <?= $article['titre'] ?></h1>
</header>
<?php foreach ($commentaires as $commentaire): ?>
    <p><?= $commentaire['auteur'] ?> dit :</p>
    <p><?= $commentaire['contenu'] ?></p>
<?php endforeach; ?>
<?php $contenu = ob_get_clean(); ?>
<?php require 'gabarit.php'; ?>

```

Bien entendu, cette vue définit les éléments dynamiques \$titre et \$contenu, puis inclut le gabarit commun.

Enfin, on crée un nouveau fichier contrôleur, article.php, qui fait le lien entre modèle et vue pour répondre au nouveau besoin. Elle a besoin de recevoir en paramètre l'identifiant de l'article. Elle s'utilise donc sous la forme article.php?id=<id de l'article>.

Fichier article.php (V7)

```

<?php
require 'modele.php';
try {
    if (isset($_GET['id'])) {
        // intval renvoie la valeur numérique du paramètre ou 0 en cas
        // d'échec
        $id = intval($_GET['id']);
        if ($id != 0) {
            $article = getArticle($id);
            $commentaires = getCommentaires($id);
            require 'vueArticle.php';
        } else {
            throw new Exception("Identifiant d'article incorrect");
        }
    } else {
        throw new Exception("Aucun identifiant d'article spécifié");
    }
} catch (Exception $e) {
    $msgErreur = $e->getMessage();
    require 'vueErreur.php';
}

```

Il faut également modifier la vue vueAccueil.php afin d'ajouter un lien vers la page article.php sur le titre de l'article :

```

...
<header>
    <a href="<?= "article.php?id=" . $article['id'] ?>">

```

```

        <h1 class="titreArticle"><?= $article['titre'] ?></h1>
    </a>
    <time><?= $article['date'] ?></time>
</header>
...

```

Pour finir, on enrichit la feuille de style CSS afin de conserver une présentation harmonieuse :

```

...
#titreReponses {
    font-size : 100%;
}

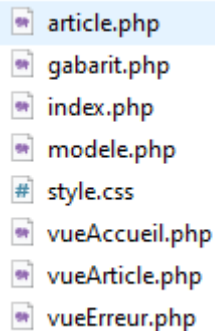
```

Page de détail



## 4. Amélioration de l'architecture MVC

Pour l'heure, notre blog d'exemple possède la structure suivante :



- modele.php représente la partie Modèle (accès aux données) ;
- vueAccueil.php, vueArticle.php et vueErreur.php constituent la partie Vue (affichage à l'utilisateur). Ces pages utilisent la page gabarit.php (template de mise en forme commune) ;
- index.php et article.php correspondent à la partie Contrôleur (gestion des requêtes entrantes).

### 4.1 Mise en œuvre d'un contrôleur frontal (Front Controller)

L'architecture actuelle, basée sur n contrôleurs indépendants, souffre de certaines limitations :

- elle expose la structure interne du site (noms des fichiers PHP) ;
- elle rend délicate l'application de politiques communes à tous les contrôleurs (authentification, sécurité, etc.).

Pour remédier à ces défauts, il est fréquent d'ajouter au site un *contrôleur frontal*.

La mise en œuvre d'un contrôleur frontal implique que index.php recevra à la fois les demandes d'affichage de la liste des articles et les demandes d'affichage d'un article précis. Il faut donc lui fournir de quoi lui permettre d'identifier l'action à réaliser. Une solution courante est d'ajouter à l'URL un paramètre action. Dans notre exemple, voici comment ce paramètre sera interprété :

- si action vaut « article », le contrôleur principal déclenchera l'affichage d'un article ;
- si action n'est pas valorisé, le contrôleur déclenchera l'affichage de la liste des articles (action par défaut).

Toutes les actions réalisables sont rassemblées sous la forme de fonctions dans le fichier controleur.php.

----- V8 -----

#### Fichier controleur.php (V8)

```
<?php
require 'modele.php';
// Affiche la liste de tous les articles du blog
function accueil()
{
    $articles = getArticles();
    require 'vueAccueil.php';
}
```



```
// Affiche les détails sur un article
function article($idArticle)
{
    $article = getArticle($idArticle);
    $commentaires = getCommentaires($idArticle);
    require 'vueArticle.php';
}
// Affiche une erreur
function erreur($msgErreur)
{
    require 'vueErreur.php';
}
```

L'action à réaliser est déterminée par le fichier index.php de notre blog, réécrit sous la forme d'un contrôleur frontal.

```
<?php
require('controleur.php');
try {
    if (isset($_GET['action'])) {
        if ($_GET['action'] == 'article') {
            if (isset($_GET['id'])) {
                $idArticle = intval($_GET['id']);
                if ($idArticle != 0)
                    article($idArticle);
                else
                    throw new Exception("Identifiant d'article non valide");
            } else
                throw new Exception("Identifiant d'article non défini");
        } else
            throw new Exception("Action non valide");
    } else {
        accueil(); // action par défaut
    }
} catch (Exception $e) {
    erreur($e->getMessage());
}
```

Remarque : l'ancien fichier contrôleur article.php est désormais inutile et peut être supprimé.

Enfin, le lien vers un article doit être modifié afin de refléter la nouvelle architecture.

Fichier vueAccueil.php (V8)

```
...
<a href="<?= "index.php?action=article&id=" . $article['id'] ?>">
    <h1 class="titreArticle"><?= $article['titre'] ?></h1>
</a>
```

...

La mise en oeuvre d'un contrôleur frontal a permis de préciser les responsabilités et de clarifier la dynamique de la partie Contrôleur de notre site :

1. Le contrôleur frontal analyse la requête entrante et vérifie les paramètres fournis ;
2. Il sélectionne et appelle l'action à réaliser en lui passant les paramètres nécessaires ;
3. Si la requête est incohérente, il signale l'erreur à l'utilisateur.

Autre bénéfice : l'organisation interne du site est totalement masquée à l'utilisateur, puisque seul le fichier `index.php` est visible dans les URL. Cette encapsulation facilite les réorganisations internes, comme celle que nous allons entreprendre maintenant.

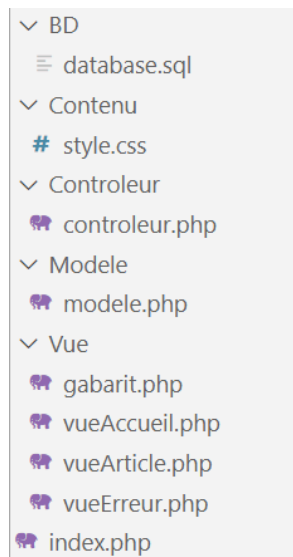
#### 4.2 Réorganisation des fichiers

Par souci de simplicité, nous avons jusqu'à présent stocké tous nos fichiers source dans le même répertoire. À mesure que le site gagne en complexité, cette organisation montre ses limites. Il est maintenant difficile de deviner le rôle de certains fichiers sans les ouvrir pour examiner leur code.

Nous allons donc restructurer notre site. La solution la plus évidente consiste à créer des sous-répertoires en suivant le découpage MVC :

- le répertoire *Modele* contiendra le fichier `modele.php` ;
- le répertoire *Vue* contiendra les fichiers `vueAccueil.php`, `vueArticle.php` et `vueErreur.php`, ainsi que la page commune `gabarit.php` ;
- le répertoire *Contrôleur* contiendra le fichier des actions `controleur.php`.

On peut également prévoir un répertoire *Contenu* pour les contenus statiques (fichier CSS, images, etc.) et un répertoire *DB* pour le script de création de la base de données. On aboutit à l'organisation suivante :



Il est évidemment nécessaire de mettre à jour les inclusions et les liens pour prendre en compte la nouvelle organisation des fichiers source. On remarque au passage que les mises à jour sont localisées et internes : grâce au contrôleur frontal, les URL permettant d'utiliser notre site ne changent pas.