Evoluer vers une architecture MVC en PHP

Rapport du Devoir : comment améliorer l’architecture d’un site web, à travers l’utilisation de l’architecture MVC.

Toutes les fichiers source sont disponibles sur le dépôt GitHub associé à l’article.

**Par : MAZTAOUI ILIAS**

**Lien GitHub :**

**https://github.com/ilias-coder/MVC-architecture**

**Sommaire :**

1. Présentation d’exemple :
2. Base de données :
3. Page principal
4. Affichage obtenu
5. Critique de l’exemple
6. Mise en place d’une architecture MVC simple
7. Amélioration de l’exemple
8. Isolation de l’affichage
9. Isolation de l’accès aux données
10. Bilan provisoire
11. Le modèle MVC
12. Présentation
13. Rôles des composants
14. Interactions entre les composants
15. Améliorations supplémentaires
16. Factorisation des éléments d’affichage communs
17. Factorisation de la connexion à la base
18. Gestion des erreurs
19. Bilan Provisoire
20. Application : affichage des détails d’un élève
21. Prise en compte du nouveau besoin
22. Affichage obtenu
23. Amélioration de l’architecture MVC
24. Rappels sur l’architecture actuelle
25. Mise en œuvre d’un contrôleur frontal
26. Réorganisation des fichiers source
27. Bilan provisoire
28. Passage à une architecture MVC orientée objet
29. Mise en œuvre du modèle objet de PHP
30. Rappels
31. Passage à un modèle orienté objet
32. Passage à une vue orienté objet
33. Passage à une vue orienté objet
34. Passage à un contrôleur orienté objet
35. Bilan Provisoire
36. Application : Ajout d’un élève
37. Application : Suppression d’un élève
38. Application :Modifier un élève
39. Construction d’un framework MVC
40. Limites de l’architecture actuelle
41. Etapes de construction du framework
42. **Présentation du context d’exemple:**

Nous mettrons en oeuvre les principes de ce qu’on a étudié et du document d’aide du framework MVC pour pouvoir faire un site web permettant d’accéder directement à la liste d’absence des étudiants.

1. **Base de données:**

La base de données utilisée est constitué d’une table stockant la liste des professeurs et une autre table stockant la liste des étudiants.

Puis, on commence à remplir notre base de données d’après phpmyadmin. (Soit en faisant des insert into par le code sql… ou en le faisant d’après le bouton insert qui nous donne un tableau à remplir.)

index.php

<!doctype html>

<html lang="fr">

    <head>

        <meta charset="UTF-8" />

        <link rel="stylesheet" href="style.css" />

        <title>Gestion des élèves</title>

    </head>

<body>

    <?php

       $bdd = new PDO('mysql:host=localhost;dbname=ensa;charset=utf8', 'root', '');

       $eleves = $bdd->query('select nom,prenom,cne,etat from eleves');

    ?>

<div id="global">

<center><h1>Liste des élèves</h1></center>

<table>

    <tr>

        <td>CNE</td>

        <td>Nom</td>

        <td>Prénom</td>

        <td>Etat</td>

    </tr>

<?php

foreach ($eleves as $eleve) {

    $et="";

    $lien="";

    if($eleve["etat"]=="true")

        {

            $et="active";

            $lien="activer.php?cne=".$eleve["cne"]."&etat=false";

        }

    else

        {

            $et="desactive";

            $lien="activer.php?cne=".$eleve["cne"]."&etat=true";

        }

?>

<tr>

    <td><?php echo $eleve["cne"]; ?></td>

    <td><?php echo $eleve["nom"]; ?></td>

    <td><?php echo $eleve["prenom"]; ?></td>

    <td><a href="<?php echo $lien; ?>"><?php echo $et; ?></a></td>

</tr>

<?php

}

?>

</table>

</div>

</body>

</html>

**Affichage obtenu :**

On utilise une feuille de style CSS pour améliorer le rendu HTML :

Style.css

#global {

    overflow-x:auto;

}

table {

    border-collapse: collapse;

    width: 100%;

}

th, td {

    text-align: left;

    border: 2px solid black;

    padding: 8px;

}

tr:nth-child(even){background-color: #f2f2f2}

Résultat obtenu de notre code jusqu’à maintenant :



1. **D- Critique de l’exemple :**

Les principaux défauts de cette page Web sont les suivants :

* elle mélange balises HTML et code PHP ;

• sa structure est monobloc, ce qui rend sa réutilisation difficile.

De manière générale, tout logiciel doit gérer plusieurs problématiques :

• interactions avec l'extérieur, en particulier l'utilisateur : saisie et contrôle de données, affichage.

C'est la problématique de **présentation** ;

• opérations sur les données (calculs) en rapport avec les règles métier (« business logic »).

C'est la problématique des traitements ;

• accès et stockage des informations qu'il manipule, notamment entre deux utilisations. C'est la problématique des **données**.

La page Web actuelle mélange code de présentation (les balises HTML) et accès aux données (requêtes SQL). Ceci est contraire au principe de responsabilité unique. Ce principe de **conception logicielle** est le suivant : afin de clarifier l'architecture et de faciliter les évolutions, une application bien conçue doit être décomposée en sous-parties, chacune ayant un rôle et une responsabilité particuliers. L'architecture actuelle montre ses limites dès que le contexte se complexifie. Le volume de code des pages PHP explose et la maintenabilité devient délicate. Il faut faire mieux.

1. **Mise en place d’une architecture MVC simple :**
2. **Amélioration de l’exemple :**
3. **Isolation de l’affichage :**

Commençons tout d’abord par la séparation du code d’accès aux données du code de présentation au sein du fichier index.php.

<?php

// accès aux donées d'après la base de donées!

    $bdd = new PDO('mysql:host=localhost;dbname=ensa;charset=utf8', 'root', '');

    $eleves = $bdd->query('select nom,prenom,cne,etat from eleves');

?>

<!doctype html>

<html lang="fr">

    <head>

        <meta charset="UTF-8" />

        <link rel="stylesheet" href="style.css" />

        <title>Gestion des élèves</title>

    </head>

<body>

<!-- Affichage de données: -->

<div id="global">

<center><h1>Liste des élèves</h1></center>

<table>

    <tr>

        <td>CNE</td>

        <td>Nom</td>

        <td>Prénom</td>

        <td>Etat</td>

    </tr>

<?php

foreach ($eleves as $eleve) {

    $et="";

    $lien="";

    if($eleve["etat"]=="true")

        {

            $et="active";

            $lien="activer.php?cne=".$eleve["cne"]."&etat=false";

        }

    else

        {

            $et="desactive";

            $lien="activer.php?cne=".$eleve["cne"]."&etat=true";

        }

?>

<tr>

    <td><?php echo $eleve["cne"]; ?></td>

    <td><?php echo $eleve["nom"]; ?></td>

    <td><?php echo $eleve["prenom"]; ?></td>

    <td><a href="<?php echo $lien; ?>"><?php echo $et; ?></a></td>

</tr>

<?php

}

?>

</table>

</div>

</body>

</html>

Mais, ce n’est pas tout, car on créera un autre fichier où on regroupera le code d’affichage :

On crée le fichier se nommant : vueaccueil.php et son contenu est comme suit :

<!doctype html>

<html lang="fr">

    <head>

        <meta charset="UTF-8" />

        <link rel="stylesheet" href="style.css" />

        <title>Gestion des élèves</title>

    </head>

<body>

<!-- Affichage de données: -->

<div id="global">

<center><h1>Liste des élèves</h1></center>

<table>

    <tr>

        <td>CNE</td>

        <td>Nom</td>

        <td>Prénom</td>

        <td>Etat</td>

    </tr>

<?php

foreach ($eleves as $eleve) {

    $et="";

    $lien="";

    if($eleve["etat"]=="true")

        {

            $et="active";

            $lien="activer.php?cne=".$eleve["cne"]."&etat=false";

        }

    else

        {

            $et="desactive";

            $lien="activer.php?cne=".$eleve["cne"]."&etat=true";

        }

?>

<tr>

    <td><?php echo $eleve["cne"]; ?></td>

    <td><?php echo $eleve["nom"]; ?></td>

    <td><?php echo $eleve["prenom"]; ?></td>

    <td><a href="<?php echo $lien; ?>"><?php echo $et; ?></a></td>

</tr>

<?php

}

?>

</table>

</div>

</body>

</html>

Et dans notre premier fichier (index.php), il nous reste :

<?php

// accès aux donées d'après la base de donées!

    $bdd = new PDO('mysql:host=localhost;dbname=ensa;charset=utf8', 'root', '');

    $eleves = $bdd->query('select nom,prenom,cne,etat from eleves');

// Affichage des données d'après:

require 'vueaccueil.php';

**Isolation de l’accès aux données :**

On ajoute un nouveau fichier nommé (modele.php) dont son contenu est comme suit :

<?php

function getEleves(){

        $bdd = new PDO('mysql:host=localhost;dbname=ensa;charset=utf8', 'root', '');

    $eleves = $bdd->query('select nom,prenom,cne,etat from eleves');

        return $eleves;

}

Et index.php devient finalement comme ça :

<?php

// accès aux donées d'après la base de donées!

require 'modele.php';

$eleves = getEleves();

// Affichage des données d'après:

require 'vueaccueil.php';

**Bilan provisoire :**

Outre la feuille de style CSS, notre page Web est maintenant constituée de trois fichiers :

• **Modele.php** (PHP uniquement) pour l'accès aux données ;

• **vueAccueil.php** (PHP et HTML) pour l'affichage des élèves ;

• **index.php** (PHP uniquement) pour faire le lien entre les deux pages précédentes.

Cette nouvelle structure est plus complexe, mais les responsabilités de chaque partie sont maintenant claires. En faisant ce travail de refactoring, nous avons rendu notre exemple conforme à un modèle d'architecture très employé sur le Web : le modèle MVC.

1. **Le modèle MVC :**
2. **Présentation :**

Le modèle MVC décrit une manière d'architecturer une application informatique en la décomposant en trois sous-parties :

• **la partie Modèle** ;

• **la partie Vue** ;

• **la partie Contrôleur**.

Ce modèle de conception (« design pattern ») a été imaginé à la fin des années 1970 pour le langage Smalltalk afin de bien séparer le code de l'interface graphique de la logique applicative. Il est utilisé dans de très nombreux langages : bibliothèques Swing et Model 2 (JSP) de Java, frameworks PHP, ASP.NET MVC, etc.

1. **Rôles des composants :**

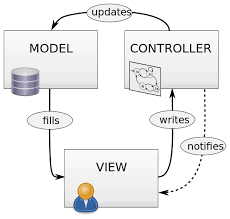
La partie **Modèle** d'une architecture MVC encapsule la logique métier (« business logic ») ainsi que l'accès aux données. Il peut s'agir d'un ensemble de fonctions (Modèle procédural) ou de classes (Modèle orienté objet).

La partie **Vue** s'occupe des interactions avec l'utilisateur : présentation, saisie et validation des données.

La partie **Contrôleur** gère la dynamique de l'application. Elle fait le lien entre l'utilisateur et le reste de l'application.

1. **Interaction entre les composants :**

Le diagramme ci-dessous résume les relations entre les composants d'une architecture MVC.



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.

1. **Améliorations supplémentaires :**

Même si notre architecture a déjà été nettement améliorée, il est possible d'aller encore plus loin.

1. **Factorisation des éléments d’affichage communs :**

**U**n 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).

Une première solution consiste à inclure les éléments communs avec des fonctions PHP **include**. Il existe une autre technique, plus souple, que nous allons mettre en œuvre : l'utilisation d'un modèle de page (gabarit), appelé **template** en anglais. Ce modèle contiendra tous les éléments communs et permettra d'ajouter les éléments spécifiques à chaque vue. On peut écrire ce template de la manière suivante (fichier **gabarit.php** dont son contenu est :).

<!doctype html>

<html lang="fr">

    <head>

        <meta charset="UTF-8" />

        <link rel="stylesheet" href="style.css" />

        <title><?= title ?></title>

    </head>

<body>

<!-- Affichage de données: -->

<div id="global">

    <?= content ?>

</div>

</body>

</html>

**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:**

<?php

$title = "Gestion des eleves";

ob\_start();

?>

<center><h1>Liste des élèves</h1></center>

<table>

    <tr>

        <td>CNE</td>

        <td>Nom</td>

        <td>Prénom</td>

        <td>Etat</td>

    </tr>

<?php

foreach ($eleves as $eleve) {

    $et="";

    $lien="";

    if($eleve["etat"]=="true")

        {

            $et="active";

            $lien="activer.php?cne=".$eleve["cne"]."&etat=false";

        }

    else

        {

            $et="desactive";

            $lien="activer.php?cne=".$eleve["cne"]."&etat=true";

        }

?>

<tr>

    <td><?php echo $eleve["cne"]; ?></td>

    <td><?php echo $eleve["nom"]; ?></td>

    <td><?php echo $eleve["prenom"]; ?></td>

    <td><a href="<?php echo $lien; ?>"><?php echo $et; ?></a></td>

</tr>

<?php

}

?>

</table>

<?php $content=ob\_get\_clean();

require 'gabarit.php';

?>

</div>

</body>

</html>

Pour expliquer ce qui était fait :

* Tout d’abord, on a ajouté la première ligne qui définit le titre ($title) qu’on a écrit en **gabarit.php**
* Puis on a ajouté la deuxième ligne qui 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.
* Puis, on a notre code HTML et la boucle foreach.
* 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 $content, ce qui permet de définir l'élément spécifique associé .
* Enfin, on déclenche le rendu de gabarit. Lors du rendu, les valeurs des éléments spécifiques $title & $content 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.

1. **Factorisation de la connexion à la base :**

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.

<?php

function getEleves(){

        $bdd = getbdd();

        $eleves = $bdd->query('select nom,prenom,cne,etat from eleves');

        return $eleves;

}

function getbdd()

{

    $bdd = new PDO('mysql:host=localhost;dbname=ensa;charset=utf8', 'root', '');

    return $bdd;

}

1. **Gestion des erreurs :**

Par souci de simplification, nous avions 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.

Notre **modele.php** devient :

<?php

function getEleves(){

        $bdd = getbdd();

        $eleves = $bdd->query('select nom,prenom,cne,etat from eleves');

        return $eleves;

}

function getbdd()

{

    $bdd = new PDO('mysql:host=localhost;dbname=ensa;charset=utf8', 'root', '', array(PDO::ATTR\_ERRMODE => PDO::ERRMODE\_EXCEPTION));

    return $bdd;

}

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

**index.php :**

<?php

// accès aux donées d'après la base de donées!

require 'modele.php';

try {

    $eleves = getEleves();

    // Affichage des données d'après:

    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.

**VueErreur.php** :

<?php $title = 'Gestion des élèves'; ?>

<?php ob\_start();?>

<p>Une erreur est survenue :<?= $mdgErreur ?> </p>

<?php $content = ob\_get\_clean(); ?>

<?php require 'gabarit.php'; ?>

Donc, notre index.php devient :

<?php

// accès aux donées d'après la base de donées!

require 'modele.php';

try {

    $eleves = getEleves();

    // Affichage des données d'après:

    require 'vueaccueil.php';

}

catch (Exception $e)

{

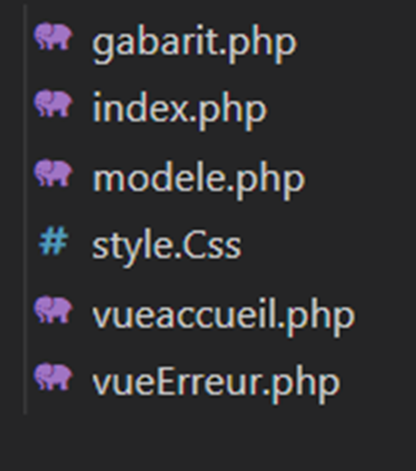
    $msgErruer = $e->getMessage();

    require 'VueErreur.php';

}

1. **Bilan provisoire :**

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



L'ajout de nouvelles fonctionnalités se fait à présent en trois étapes :

• écriture des fonctions d'accès aux données dans le modèle ;

• création d'une nouvelle vue utilisant le gabarit pour afficher les données.

• ajout d'une page contrôleur pour lier le modèle et la vue.

1. **Affichage des détails d’un élève :**

Jusqu’à maintenant, notre application est basique. Alors on essaiera de la rendre plus réaliste en ajoutant la possibilité de voir les détails concernant chaque étudiant ! Ces détails doivent contenir bien sûr les absences et plus d’informations sur l’élèves sélectionné.

1. **Prise en compte du nouveau besoin** :

On doit ajouter tout d’abord des fonctions permettant d’accéder aux données dont on a besoin !

Modele.php devient :

<?php

    function getbdd()

    {

        $bdd = new PDO('mysql:host=localhost;dbname=ensa;charset=utf8', 'root', '', array(PDO::ATTR\_ERRMODE => PDO::ERRMODE\_EXCEPTION));

        return $bdd;

    }

    function getEleves(){

        $bdd = getbdd();

        $eleves = $bdd->query('select nom,prenom,cne,etat from eleves');

        return $eleves;

    }

    function getEleve($cne){

        $bdd = getbdd();

        $eleve = $bdd->prepare('select nom,prenom,cne,etat from eleves where cne=?');

        $eleve->execute(array($cne));

        if ($eleve->rowcount()==1)

        return $eleve->fetch(); // Pour accéder à la première ligne de résultat !

        else

        throw new exception("Aucun élève ne correspond à l'identifiant");

    }

    function getAbsences($cne)

    {

        $bdd = getbdd();

        $absences = $bdd->prepare('select matiere, date, nbre\_heure from absences where cne=?');

        $absences->execute(array($cne));

        return $absences;

    }

    function getAbsencesParmatiere($cne)

    {

        $bdd = getbdd();

        $absences = $bdd->prepare('select matiere, sum(nbre\_heure) as Nbre\_heures from absences where cne=? group by matiere' );

        $absences->execute(array($cne));

        return $absences;

    }

    function getCommentaire($cne)

    {

        $bdd = getbdd();

        $commentaires = $bdd->prepare('select commentaire from eleves where cne=?');

        $commentaires->execute(array($cne));

        if ($commentaires->rowcount()==1)

        return $commentaires->fetch(); // Pour accéder à la première ligne de résultat !

        else

        throw new exception("Aucun élève ne correspond à l'identifiant");

    }

Et on crée une vue qui s’appelle vueEleve.php dont son role est d’afficher les informations demandés en modele.php

<?php $title="Gestionnaire des absences " .$eleve['nom'];?>

<?php ob\_start(); ?>

<h5><?= $eleve['cne'] ?></h5>

<h5><?= $eleve['nom'] ?></h5>

<h5><?= $eleve['prenom'] ?></h5>

<p><a href="index.php"> retour à la liste des élèves</a></p>

<hr>

<h5> Etat:

     <?php if($eleve['etat']==1) {echo "<strong style='color=green;'>Actif"</strong>;}

    else {echo "<strong style='color=red;'> Inactif" </strong>;}?></h5>

<hr>

<h3>Les absences de Mr <?= $eleve['nom'] ?> <?= $eleve['prenom'] ?> </h3>

<table>

    <tr>

        <th>Matière :</th>

        <th>Jour</th>

        <th>Nombre d'heures</th>

    </tr>

    <?php foreach ($absences1 as $absence): ?>

    <tr>

        <td><?= $absence['matiere'] ?></td>

        <td><time><?= $absence['date']?></time></td>

        <td><?= $absence['nbre\_heure'] ?></td>

    </tr>

   <?php endforeach; ?>

</table>

<br>

<hr>

<h2>Heures absentés par matière: </h2>

<table>

    <tr>

        <th>Matière</th>

        <th>Nombre d'heures</th>

    </tr>

    <?php foreach ($absences2 as $absence): ?>

    <tr>

        <td><?= $absence['matiere'] ?></td>

        <td><?= $absence['Nbre\_heures'] ?></td>

    </tr>

    <?php endforeach; ?>

</table>

<br>

<hr>

<p><?= $commentaires['commentaire']?></p>

<hr>

<?php $content = 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, **eleve.php**, qui fait le lien entre modèle et vue pour répondre au nouveau besoin. Elle a besoin de recevoir en paramètre l'identifiant du billet. Elle s'utilise donc sous la forme eleve.php?cne=<cneeleve>.

<?php

require 'modele.php';

try {

    if(isset($\_GET['cne']))

    {

        $cne=strval($\_GET['cne']);

        if ($cne!='null')

        {

            $eleve=getEleve($cne);

            $absences1=getAbsences($cne);

            $absences2=getAbsencesParmatiere($cne);

            $commentaires=getCommentaire($cne);

            require 'vueEleve.php';

        }

        else

        throw new exception("Identifiant d'élève incorrect");

    }

    else

    throw new exception("Aucun CNE d'élève");

    }

    catch (exception $e)

    {

        $msgErreur=$e->getMessage();

        require 'vueErreur.php';

    }

Maintenant, il est temps de modifier la vue vueAccueil.php pour faire une liaison vers eleve.php

<td><a href="<?="eleve.php?cne=".$eleve['cne']?>"><button>Détails</button></a></td>

Maintenant, il nous reste de faire une feuille de style pour ajouter du style à notre texte HTML !

#global {

    overflow-x:auto;

}

table {

    border-collapse: collapse;

    width: 100%;

}

th, td {

    text-align: left;

    border: 2px solid black;

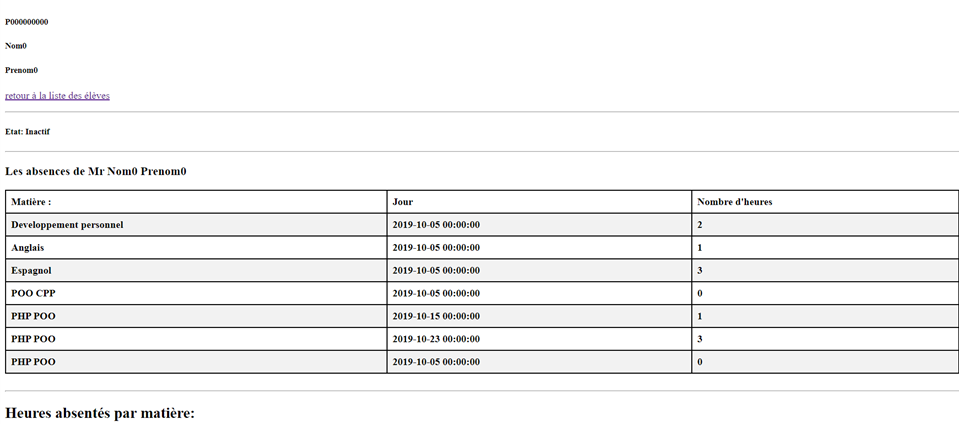
    padding: 8px;

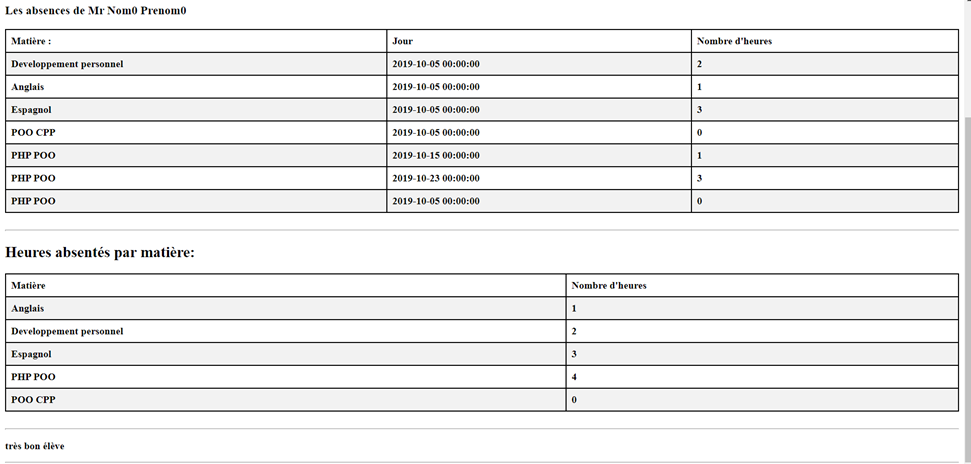
}

tr:nth-child(even){background-color: #f2f2f2}

1. **Affichage du contenu:**

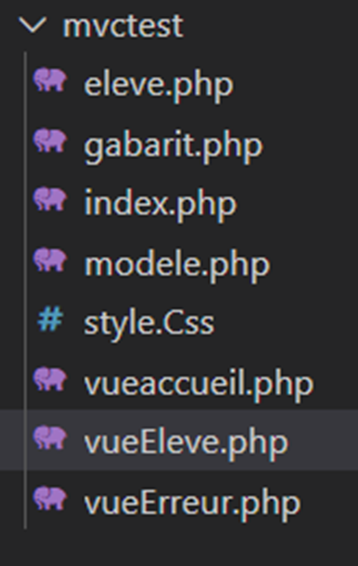
****

****

****

1. **Amélioration de l’architecture MVC:**
2. **Rappels sur l’architecture actuelle** :

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



Rappelons les rôles de chaque élément :

• **Modele.php** représente la partie Modèle (accès aux données) ;

• **vueAccueil.php**, **vueBillet.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 **billet.php** correspondent à la partie Contrôleur (gestion des requêtes entrantes).

1. **Mise en œuvre d’un contrôleur frontal :**

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.

Le contrôleur frontal constitue le point d'entrée unique du site. Son rôle est de centraliser la gestion des requêtes entrantes. Il utilise le service d'un autre contrôleur pour réaliser l'action demandée et renvoyer son résultat sous la forme d'une vue.

Un choix fréquent consiste à transformer le fichier principal index.php en contrôleur frontal. Nous allons mettre en œuvre cette solution.

Ce changement d'architecture implique un changement d'utilisation du site. Voici comment fonctionne actuellement notre gestionnaire d’absence:

• l'exécution de **index.php** permet d'afficher la liste des élèves ;

• l'exécution de **eleve.php?id=<cne>** affiche les détails du billet identifié dans l'URL.

La mise en œuvre d'un contrôleur frontal implique que **index.php** recevra à la fois les demandes d'affichage de la liste des élèves et les demandes d'affichage d'un billet 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 « eleve»**, le contrôleur principal déclenchera l'affichage d'un élève.

• si **action n'est pas valorisé**, le contrôleur déclenchera l'affichage de la liste des élèves (action par défaut).

Toutes les actions réalisables sont rassemblées sous la forme de fonctions dans le fichier **Controleur.php. dont son contennu est :**

<?php

require 'modele.php';

function accueil()

{

    $eleves=geteleves();

    require 'vue/vueAccueil.php';

}

function eleve($cne)

{

    $eleve=getEleve($cne);

    $absences1=getAbsences($cne);

    $absences2=getAbsencesParmatiere($cne);

    $commentaires=getCommentaire($cne);

    require 'vue/vueEleve.php';

}

function erreur($msgErreur)

{

    require 'vue/vueErreur.php';

}

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

**index.php**

<?php

// accès aux donées d'après la base de donées!

require('controlleur/controlleur.php');

try{

    if (isset($\_GET['action']))

    {

        if ($\_GET['action'=='eleve'])

        {

            if(isset($\_GET['cne']))

            {

                $cne=strval($\_GET['cne']);

                if ($cne!='null')

                eleve($cne);

                else throw new exception("identifiant d'élève non valide");

            }

            else throw new exception("identifiant d'élève non défini");

        }

        else throw new exception("Action non valide");

    }

    else accueil();

}

catch (Exception $e)

{

    erreur($e->getMessage());

}

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

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

**vueaccueil.php**

<td><a href="<?="index.php?action=eleve&cne=".$eleve['cne']?>"><button>Détails</button></a></td>

La mise en œuvre d'un contrôleur frontal a permis de préciser les responsabilités et de clarifier la dynamique de la

partie Contrôleur de notre site :

* Le contrôleur frontal analyse la requête entrante et vérifie les paramètres fournis ;
* Il sélectionne et appelle l'action à réaliser en lui passant les paramètres nécessaires ;
* 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.

1. **Réorganisation des fichiers source :**

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.

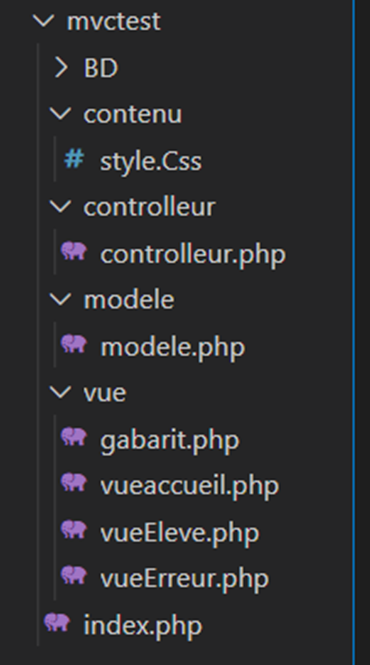
Nous allons donc restructurer notre site. La solution la plus évidente consiste à créer des sous-répertoires en suivant le découpage MVC :

• le répertoire **Modele** contiendra le fichier Modele.php ;

• le répertoire **Vue** contiendra les fichiers **vueAccueil.php**, **vueEleve.php** et **vueErreur.php**, ainsi que la page commune **gabarit.php** ;

• le répertoire **Controleur** 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 **BD** 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.

1. **Bilan provisoire :**

Notre gestionnaire d’élève est maintenant structuré selon les principes du modèle MVC, avec une séparation nette des responsabilités entre composants qui se reflète dans l'organisation des sources. Notre solution est avant tout **procédurale** : les actions du contrôleur et les services du modèle sont implémentés sous la forme de fonctions.

L'amélioration de l'architecture passe maintenant par la mise en œuvre des concepts de la programmation orientée objet, que PHP supporte pleinement depuis plusieurs années.

1. **Passage à une architecture MVC orientée objet :**
2. **Passage à un modèle orienté objet :**

Dans le cadre d'un passage à la POO, il serait envisageable de créer des classes métier modélisant les entités du domaine, en l'occurrence **Eleve et Absence**.

Plus modestement, nous allons nous contenter de définir les services d'accès aux données en tant que méthodes et non comme simples fonctions. Voici une première version de la classe **Modele**.

<?php

class Modele

{

    private function getbdd()

    {

        $bdd = new PDO('mysql:host=localhost;dbname=ensa;charset=utf8', 'root', '', array(PDO::ATTR\_ERRMODE => PDO::ERRMODE\_EXCEPTION));

        return $bdd;

    }

    public function getEleves(){

        $bdd = $this->getbdd();

        $eleves = $bdd->query('select nom,prenom,cne,etat from eleves');

        return $eleves;

    }

    public function getEleve($cne){

        $bdd = $this->getbdd();

        $eleve = $bdd->prepare('select nom,prenom,cne,etat from eleves where cne=?');

        $eleve->execute(array($cne));

        if ($eleve->rowcount()==1)

        return $eleve->fetch(); // Pour accéder à la première ligne de résultat !

        else

        throw new exception("Aucun élève ne correspond à l'identifiant");

    }

    public function getAbsences($cne)

    {

        $bdd = $this->getbdd();

        $absences = $bdd->prepare('select matiere, date, nbre\_heure from absences where cne=?');

        $absences->execute(array($cne));

        return $absences;

    }

    public function getAbsencesParmatiere($cne)

    {

        $bdd = $this->getbdd();

        $absences = $bdd->prepare('select matiere, sum(nbre\_heure) as Nbre\_heures from absences where cne=? group by matiere' );

        $absences->execute(array($cne));

        return $absences;

    }

    public function getCommentaire($cne)

    {

        $bdd = $this->getbdd();

        $commentaires = $bdd->prepare('select commentaire from eleves where cne=?');

        $commentaires->execute(array($cne));

        if ($commentaires->rowcount()==1)

        return $commentaires->fetch(); // Pour accéder à la première ligne de résultat !

        else

        throw new exception("Aucun élève ne correspond à l'identifiant");

    }

}

Par rapport à notre ancien modèle procédural, la seule réelle avancée offerte par cette classe est l'encapsulation (mot-clé private) de la méthode de connexion à la base. De plus, elle regroupe des services liés à des entités distinctes (billets et commentaires), ce qui est contraire au principe de cohésion forte, qui recommande de regrouper des

éléments (par exemple des méthodes) en fonction de leur problématique.

Une meilleure solution consiste à créer un modèle par entité du domaine, tout en regroupant les services communs dans une superclasse commune. On peut écrire la classe Billet, en charge de l'accès aux données des billets, comme ceci.

**eleve.php**

<?php

require\_once 'modele/modele.php';

class Eleve extends Modele

{

    public function getEleves()

    {

        $sql = 'select nom,prenom,cne,etat from eleves';

        $eleves=$this->executerRequete($sql);

        return $eleves;

    }

    public function getEleve($cne)

    {

        $sql='select nom,prenom,cne,etat from eleves where cne=?';

        $eleve=$this->executerRequete($sql,array($cne));

        if ($eleve->rowcount()==1)

        return $eleve->fetch(); // Pour accéder à la première ligne de résultat !

        else

        throw new exception("Aucun élève ne correspond à l'identifiant");

    }

}

?>

La classe **Modele** est désormais **abstraite** (mot-clé abstract) et fournit à ses classes dérivées un service d'exécution d'une requête SQL :

**modele.php**

<?php

abstract class Modele

{

    private $bdd;

    protected function executerRequete($sql, $params=null)

    {

        if($params==null)

        {

            $resultat=$this->getbdd()->query($sql);

        }

        else

        {

            $resultat = $this->getbdd()->prepare($sql);

            $resultat->execute($params);

        }

return $resultat;

    }

    private function getbdd()

    {

        if ($this->bdd == null)

        {

        $this->bdd = new PDO('mysql:host=localhost;dbname=ensa;charset=utf8', 'root', '', array(PDO::ATTR\_ERRMODE => PDO::ERRMODE\_EXCEPTION));

        return $this->bdd;

        }

    }

}

Et on ajoute **absence.php** tel que :

<?php

require\_once 'modele/modele.php';

class Absence extends Modele

{

    public function getAbsences($cne)

    {

        $sql='select matiere, date, nbre\_heure from absences where cne=?';

        $absences=$this->executerRequete($sql,array($cne));

        return $absences;

    }

    public function getAbsencesParmatiere($cne)

    {

        $sql='select matiere, sum(nbre\_heure) as Nbre\_heures from absences where cne=? group by matiere';

        $absences=$this->executerRequete($sql,array($cne));

        return $absences;

    }

}

?>

Et **commentaire.php** tel que :

<?php

require\_once 'modele/modele.php';

class Commentaire extends Modele

{

    public function getCommentaire($cne)

    {

        $sql='select commentaire from eleves where cne=?';

        $commentaires=$this->executerRequete($sql,array($cne));

        if ($commentaires->rowcount()==1)

        return $commentaires->fetch(); // Pour accéder à la première ligne de résultat !

        else

        throw new exception("Aucun élève ne correspond à l'identifiant");

    }

}

?>

1. **Passage à une vue orienté objet :**

**Vue.php**

Nous allons créer une classe Vue dont le rôle sera de gérer la génération des vues.

<?php

class Vue

{

    private $fichier; // nom du fichier associé à la vue

    private $title; //

    public function \_\_construct($action)

    {

        $this->fichier = "vue/vue" .$action .".php";

    }

    // Générer et afficher la vue

    public function generer($donnees)

    {

        $content = $this->genererFichier($this->fichier,$donnees);

        $vue=$this->genererFichier('vue/gabarit.php',array('title'=>$this->title,'content'=>$content));

        echo $vue;

    }

    private function genererFichier($fichier,$donnees)

    {

        if(file\_exists($fichier))

        {

            // Rend les éléments du tableau $donnees accessible sur la vue

            extract($donnees);

            // Démarrer la temporisation de la sortie

            ob\_start();

            // inclut le fichier vue

            // Son résultat est placé dans le tampon de sortie

            require $fichier;

            // arret de la temporisation et renvoi du tampon de sortie

            return ob\_get\_clean();

        }

else throw new exception("Fichier '$fichier' introuvable");

   }

}

?>

Le constructeur de Vue prend en paramètre une action, qui détermine le fichier vue utilisé. Sa méthode generer() génère d'abord la partie spécifique de la vue afin de définir son titre (attribut $title) et son contenu (variable locale $content). Ensuite, le gabarit est généré en y incluant les éléments spécifiques de la vue. Sa méthode interne

genererFichier() encapsule l'utilisation de require et permet en outre de vérifier l'existence du fichier vue à afficher. Elle utilise la fonction http://php.net/manual/fr/function.extract.php pour que la vue puisse accéder aux variables PHP requises, rassemblées dans le tableau associatif $donnees.

Il n'est pas nécessaire de modifier le fichier gabarit. Par contre, les fichiers de chaque vue doivent être modifiés pour définir $this->title et supprimer les appels aux fonctions PHP de temporisation. ob\_start();

<?php $content=ob\_get\_clean();

require 'gabarit.php';

?>

L'affichage d'une vue se fera désormais en instanciant un objet de la classe Vue, puis en appelant sa méthode

generer().

1. **Passage à un contrôleur orienté objet**

Notre partie Contrôleur actuelle se compose d'une série d'actions écrites sous la forme de fonctions et du contrôleur frontal index.php.

Toute évolution du site Web va faire augmenter le nombre d'actions possibles, jusqu'à rendre les fichiers **Controleur.php** et **index.php** difficiles à lire et à maintenir.

Une solution plus modulaire consiste à répartir les actions dans plusieurs classes contrôleur, en fonction du contexte associé aux actions. Ici, nous pourrions créer une classe **ControleurAccueil** pour gérer l'accueil et une classe **ControleurEleve** pour gérer l'affichage d'un billet.

Bien entendu, les nouveaux contrôleurs utilisent les services des classes des parties Modèle et Vue définies précédemment.

**ControleurAccueil.php**

<?php

require\_once 'modele/eleve.php';

require\_once 'vue/vue.php';

class ControlleurAccueil

{

    private $eleve;

    public function \_\_construct()

    {

        $this->eleve = new Eleve();

    }

    public function accueil()

    {

        $eleves=$this->eleve->getEleves();

        $vue = new Vue("accueil");

        $vue->generer(array('eleves' => $eleves));

    }

}

?>

**ControleurELeve.php**

<?php

require\_once 'modele/eleve.php';

require\_once 'modele/absence.php';

require\_once 'vue/vue.php';

class ControlleurEleve

{

    private $eleve;

    private $absence;

    private $commentaire;

    public function \_\_construct()

    {

        $this->eleve = new Eleve();

        $this->absence = new Absence();

        $this->commentaire = new Commentaire();

    }

    public function eleve($cne)

    {

        $eleve=$this->eleve->getEleve($cne);

        $absences1 = $this->absence->getAbsences($cne);

        $absences2 = $this->absence->getAbsencesparmatiere($cne);

        $commentaires= $this->commentaire->getCommentaire($cne);

        $vue = new Vue("Eleve");

        $vue->generer(array('eleves' => $eleves,'absences1' => $absences1,'absences2' => $absences2,'commentaires'=>$commentaires));

    }

}

?>

Chaque classe contrôleur instancie les classes modèle requises, puis utilise leurs méthodes pour récupérer les données nécessaires aux vues. La méthode generer de la classe Vue définie plus haut est utilisée en lui passant en paramètre un tableau associatif contenant l'ensemble des données nécessaires à la génération de la vue. Chaque élément de ce tableau est constitué d'une clé (entre apostrophes) et de la valeur associée à cette clé.

Quant au contrôleur frontal, on peut le modéliser à l'aide d'une classe **Routeur** dont la méthode principale analyse la requête entrante pour déterminer l'action à entreprendre. On parle souvent de **routage de la requête.**

**Routeur.php**

<?php

require\_once 'controlleur/controlleurAccueil.php';

require\_once 'controlleur/controlleurEleve.php';

require\_once 'modele/commentaire.php';

require\_once 'vue/vue.php';

class Routeur

{

    private $ctrlAccueil;

    private $ctrlEleve;

    public function \_\_construct()

    {

        $this->ctrlAccueil = new ControlleurAccueil();

        $this->ctrlEleve = new ControlleurEleve();

    }

    public function routerRequete()

{

    try{

        if (isset($\_GET['action']))

        {

            if ($\_GET['action'=='eleve'])

            {

                if(isset($\_GET['cne']))

                {

                    $cne=strval($\_GET['cne']);

                    if ($cne!='null')

                    eleve($cne);

                    else throw new exception("identifiant d'élève non valide");

                }

                else throw new exception("identifiant d'élève non défini");

            }

            else throw new exception("Action non valide");

        }

        else $this->ctrlAccueil->accueil();

    }

        catch (Exception $e)

        {

            $this->erreur($e->getMessage());

        }

    }

    private function erreur($msgErreur)

    {

        $vue=new Vue("Erreur");

        $vue->generer(array('msgErreur' => $msgErreur));

    }

}

Le fichier principal index.php est maintenant simplifié à l'extrême. Il se contente d'instancier le routeur puis de lui

faire router la requête.

<?php

// accès aux donées d'après la base de donées!

require 'controlleur/routeur.php';

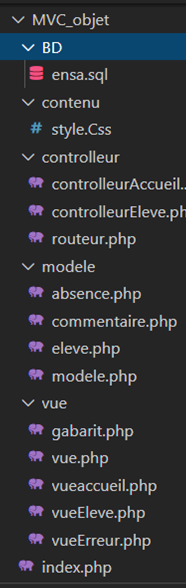
$routeur = new Routeur();

$routeur->routerRequete();

1. **Bilan provisoire :**

La structure actuelle du site est présentée ci-dessous. Elle est évidemment beaucoup plus complexe qu'au départ.

Cette complexité est le prix à payer pour disposer de bases robustes qui faciliteront la maintenance et les évolutions futures.



L'ajout de nouvelles fonctionnalités se fait à présent en trois étapes :

* Ajout ou enrichissement de la classe **modèle** associée ;
* Ajout ou enrichissement d'une **vue** utilisant le gabarit pour afficher les données ;
* Ajout ou enrichissement d'une classe **contrôleur** pour lier le modèle et la vue.

1. **Ajout d’une absence**

On souhaite maintenant mettre à jour d’élève par l’activer ou le désactiver à partir de la page d’accueil.

**A-Prise en compte du nouveau besoin**

On commence par ajouter à la classe **Eleve** une méthode permettant de faire la mise à jour l’état d’un élève.

public function update\_etat($cne,$etat)

    {

        $sql='update eleves set etat=' .$etat .'where cne='.$cne.'';

        $maj=$this->executerRequete($sql);

    }

Ensuite, on ajoute le lien de redirection vers l’action voulu ;

En **Vueaccueil.php**

    $et="";

    $lien="";

    if($eleve["etat"]==1)

        {

            $et="active";

            $lien="index.php?action=update&cne=".$eleve["cne"]."&etat=0";

        }

    else

        {

            $et="desactive";

            $lien="index.php?action=update&cne=".$eleve["cne"]."&etat=1";

        }

?>

Maintenant, on ajoutera au controleur (**ControlleurEleve.php**) une méthode associé à l’action de la soumission du formulaire.

public function update\_etat($cne,$etat)

    {

        $this->eleve->update\_etat($cne,$etat);

    }

Finalement il nous reste de mettre à jour la méthode de routage :

            else if ($\_GET['action']=='update')

            {

                $cne=$this->getPArametre($\_GET,'cne');

                $etat=$this->getParametre($\_GET,'etat');

                $this->ctrlEleve->update\_etat($cne,$etat);

                $this->ctrlAccueil->accueil();

            }

            else throw new exception("Action non valide");

Finalement, le résultat qu’on trouve :



