

DECOUVRIR LE PATTERN MVC

Modèle – Vue - Contrôleur

PARTIE 1

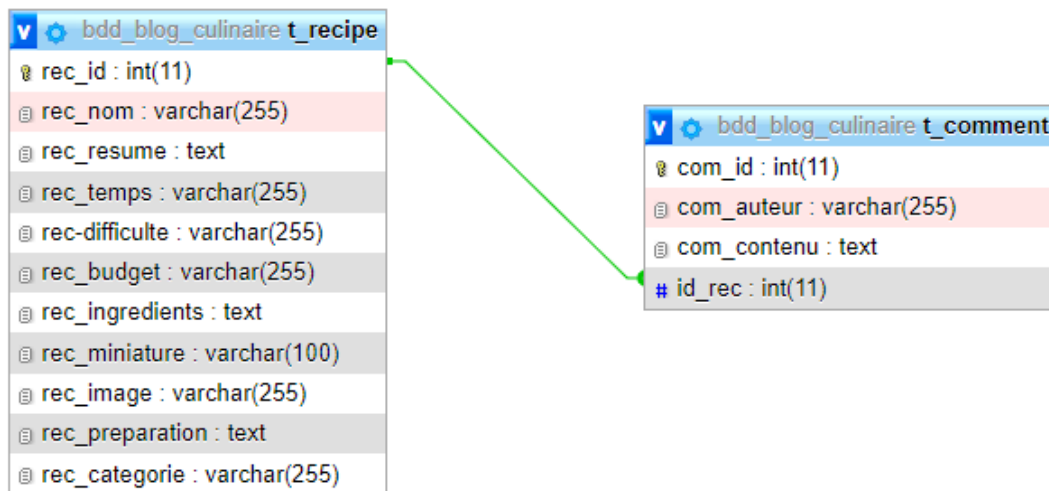
Réalisation Stéphane Pontonnier

1 - Présentation de l'exemple

Nous mettrons en œuvre les principes présentés dans ce TP sur un exemple simple : une page Web PHP de type « blog » puisant ses informations dans une base de données relationnelle.

Vous trouverez les fichiers source en annexe.

La base de données



Cette base de données est à importer via ***bdd_blog_culinaire.sql***

La page principale

La page principale est ***index.php***

Affichage obtenu

Il s'agit d'un exemple assez classique d'utilisation de PHP pour construire une page dynamique affichée par le navigateur client.

Elle liste 1 recette mise à la une, les 3 dernières recettes ainsi que les 3 derniers commentaires.

Critique de l'exemple

Quelles sont les remarques que vous pouvez faire en découvrant le code source de cette page ?

Quels sont les défauts de cette page ? Comment y remédier ?

2 - Mise en place d'une architecture MVC simple

Amélioration de l'exemple

Isolation de l'affichage

Dans le fichier ***index.php***, séparez le code d'accès aux données du code de présentation.

Que pouvez-vous dire de ce nouveau fichier ? Correspond-il aux bonnes pratiques de développement PHP (PSR-1) ?

Vous pouvez aller plus loin, en regroupant le code d'affichage dans un fichier dédié nommé ***vueAccueil.php***

```
1 <!doctype html>
2 <html lang="fr">
3 <head>
4 ...
5 </head>
6 <body>
7 ...
8     <div id="contenu">
9         <?php foreach ($recipes as $recipe): ?>
10             <article>
11                 ...
12             </article>
13             <hr />
14         <?php endforeach; ?>
15     </div> <!-- #contenu -->
16 ...
17 </body>
18 </html>
```

Votre page principale doit maintenant contenir l'accès aux données et l'affichage

```
1 <?php
2 // index.php
3
4 // Accès aux données
5 // votre code .....
6
7 // Affichage
8 // votre code .....
9
10 ?>
```



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.

Isolation de l'accès aux données

Vous pouvez gagner en modularité en isolant le code d'accès aux données dans un fichier PHP nommé **Modele.php**

```
1 <?php
2 // Modele.php
3
4 // Constantes de connexion à la BDD
5
6 // Affiche les 3 dernières recettes
7 function threeLastRecipes(){
8     // code....
9 }
10
11 // Affiche les 3 derniers commentaires
12 function threeLastComments(){
13     // code....
14 }
15
16 ?>
```

Dans ces fonctions vous récupérez les données stockées en base, puis vous les retournez via les fonctions dédiées

Le code d'affichage de **vueAccueil.php** ne change pas. Le lien entre accès aux données et présentation est effectué par le fichier principal **index.php**. Ce fichier est maintenant très simple.

```

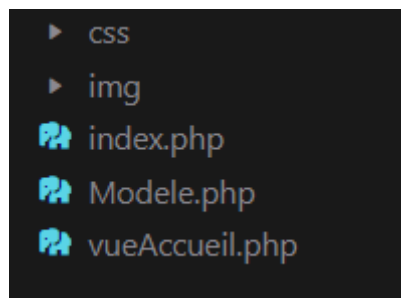
1 <?php
2 // index.php
3
4 require 'Modele.php';
5
6 $recipes = threeLastRecipes();
7 $comments = threeLastComments();
8
9 require 'vueAccueil.php';
10
11
12 ?>

```

Bilan provisoire

Outre la feuille de style CSS et les images, votre 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 billets du blog ;
- **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*, vous avez rendu votre exemple conforme à un modèle d'architecture très employé sur le Web : le modèle **MVC**.

Le modèle MVC

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.

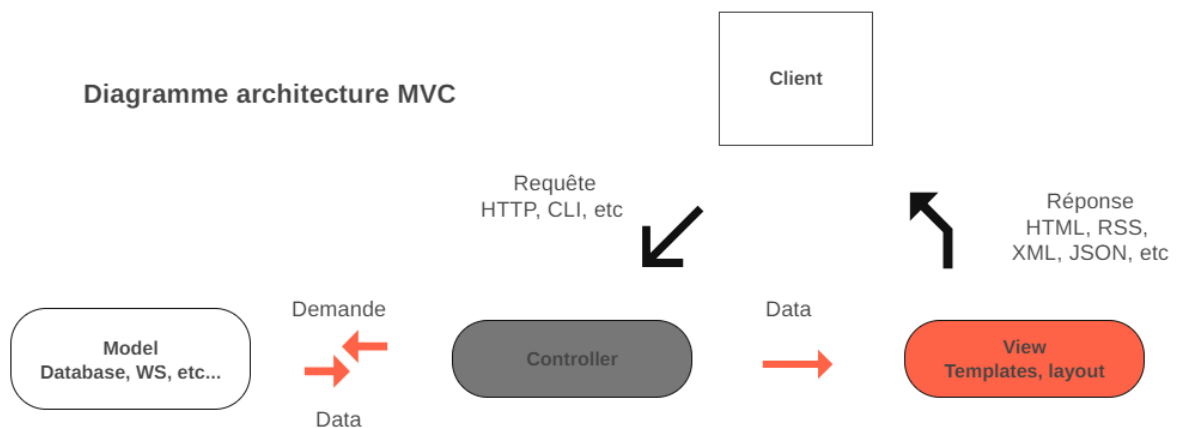
Rôles des composants

Décrivez le rôle de chacun des composants dans une architecture MVC.

Qu'est-ce que la logique métier (« business logic ») ?

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.

Avantages et inconvénients

Le modèle MVC offre une séparation claire des responsabilités au sein d'une application, en conformité avec les principes de conception déjà étudiés : responsabilité unique, couplage **faible** et cohésion **forte**. Le prix à payer est une augmentation de la complexité de l'architecture.

Dans le cas d'une application Web, l'application du modèle MVC permet aux pages HTML (qui constituent la partie Vue) de contenir le moins possible de code serveur,

étant donné que le scripting est regroupé dans les deux autres parties de l'application.

Différences avec le modèle en couches

Recherchez ce qu'est un modèle en couche ? Quelle différence avec le modèle MVC ?

Améliorations supplémentaires

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

Factorisation des éléments d'affichage communs

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).

Une première solution consiste à inclure les éléments communs avec des fonctions PHP **include**. Il existe une autre technique, plus souple, que vous allez 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**).

```
1 <!doctype html>
2 <html lang="fr">
3 <head>
4 <meta charset="UTF-8" />
5 <link rel="stylesheet" href="style.css" />
6 <title><?=$titre ?></title> <!-- Élément spécifique -->
7 </head>
8 <body>
9 <div id="global">
10 <header>
11 <a href="index.php"><h1 id="titreBlog">Mon Blog</h1></a>
12 <p>Je vous souhaite la bienvenue sur ce modeste blog culinaire.</p>
13 </header>
14 <div id="contenu">
15 <?=$contenu ?> <!-- Élément spécifique -->
16 </div>
17 <footer id="piedBlog">
18 Blog réalisé avec PHP, HTML5 et CSS.
19 </footer>
20 </div> <!-- #global -->
21 </body>
22 </html>
```

Exemple de gabarit.php

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

```
1 <?=$titre = "Le Blog Culinaire"; ?>
2
3 <?php ob_start(); ?>
4 <div class="container col-xxl-8 px-4 py-5">
5   <h1 class="display-5 fw-bold text-center line"><span>Les recettes de saison... </span></h1>
6   <div class="row">
7     <?php foreach ($recettes as $recette): ?>
8       <div class="col-12 col-md-6 col-xl-4 my-3">
9         <div class="card mx-auto" style="width: 18rem;">
10          
11          <div class="card-body">
12            <p class="card-text"><?=$recette['rec_resume'];?></p>
13          </div>
14        </div>
15      </div>
16    <?php endforeach; ?>
17  </div>
18 </div>
19 <div class="container col-xxl-8 px-4 py-5">
20   <h1 class="display-5 fw-bold text-center line"><span>Les derniers commentaires... </span></h1>
21
22   <div id="carouselExampleSlidesOnly" class="carousel slide" data-bs-ride="carousel">
23     <div class="carousel-inner">
24       <?php
25         $flag = 1;
26         foreach ($commentaires as $comment):
27           if($flag == 1){
28             echo '<div class="carousel-item active">';
29           } else{
30             echo '<div class="carousel-item">';
31           }
32           <div class="carousel-caption d-md-block">
33             <h5 class="text-dark"><?=$comment['com_contenu'];?></h5>
34             <p class="text-dark"><?=$comment['com_auteur'];?></p>
35             <p class="text-recipe-carousel fst-italic">Recette : <?=$comment['rec_nom'];?></p>
36           </div>
37         </div>
38       <?php
39         $flag += 1;
40       endforeach;?>
41     </div>
42   </div> <!-- fin de carousel -->
43 </div>
44 <?php $contenu = ob_get_clean(); ?>
45
46 <?php require 'gabarit.php'; ?>
```

Ce code mérite quelques explications :

- 1 La première ligne définit la valeur de l'élément spécifique **\$titre** ;
- 2 Le 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 associées aux recettes 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.

Factorisation de la connexion à la base

Il est possible d'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 vous ajouterez d'autres fonctions au Modèle.

```
1 <?php
2 // Modele.php
3
4 // Constantes de connexion à la BDD
5
6 // Affiche les 3 dernières recettes
7 function threeLastRecipes(){
8     // code....
9 }
10
11 // Affiche les 3 derniers commentaires
12 function threeLastComments(){
13     // code....
14 }
15
16 /*
17 Effectue la connexion à la BDD
18 Instancie et renvoie l'objet PDO associé
19 */
20 function getBdd(){
21     // code.....
22 }
23
24 ?>
```

A vous de jouer 😊 pour écrire le code de ce fichier.

Gestion des erreurs

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.

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

Dans le fichier *modele.php* :

```
1 <?php
2 function getBdd() {
3     $dsn = 'mysql:host='.DBHOST.';dbname='.DBNAME;
4
5     $db = new PDO($dsn, DBUSER, DBPASS, array(PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION));
6     $db->exec("SET NAMES utf8");
7     $db->setAttribute(PDO::ATTR_DEFAULT_FETCH_MODE, PDO::FETCH_ASSOC);
8
9     return $db;
10 }
11 ?>
```

Vous pouvez ensuite ajouter à votre page une gestion minimaliste des erreurs de la manière suivante :

```
1 <?php
2 // index.php
3
4 require 'Modele.php';
5
6 try{
7     $recipes = threeLastRecipes();
8     $comments = threeLastComments();
9     require 'vueAccueil.php';
10 }
11 catch(Exception $e) {
12     echo '<html><body>Erreur ! ' . $e->getMessage() . '</body></html>';
13 }
14
15 ?>
```


Le premier **require** inclut uniquement la définition des fonctions de récupération 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.

Pour tester :

Modifiez le nom de votre BDD dans la constante de connexion et visualisez le résultat sur le navigateur

Il est possible d'utiliser l'affichage du gabarit des vues même en cas d'erreur. Il suffit de définir une vue **vueErreur.php** dédiée à leur affichage.

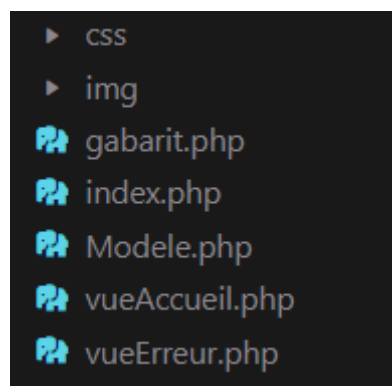
	<p>Sur le modèle de <i>vueAccueil.php</i>, créez la page <i>vueErreur.php</i>. N'oubliez pas d'utiliser <code>ob_start()</code> et <code>ob_get_clean()</code>.</p>
---	---

Modifiez ensuite le contrôleur pour déclencher le rendu de cette vue en cas d'erreur :

```
1 <?php
2 // index.php
3
4 require 'Modele.php';
5
6 try{
7     $recipes = threeLastRecipes();
8     $comments = threeLastComments();
9     require 'vueAccueil.php';
10 }
11 catch(Exception $e) {
12     $msgErreur = $e->getMessage();
13     require 'vueErreur.php';
14 }
15
16 ?>
```

Bilan provisoire

Vous avez accompli sur votre page d'exemple un important travail de *refactoring* qui a modifié son architecture en profondeur. Votre page respecte à présent un modèle MVC simple.



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


- Ecriture 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.

Application : affichage des détails d'une recettes

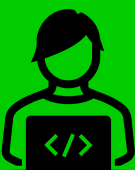
Afin de rendre votre exemple plus réaliste, vous allez ajouter un nouveau besoin : le clic sur le l'image d'une recette du blog doit afficher sur une nouvelle page le contenu et les commentaires associés à cette recette.

Prise en compte du besoin

	<p>Ajoutez dans votre modèle (fichier Modele.php) les fonctions d'accès aux données dont vous avez besoin. Utilisez les requêtes préparées pour récupérer les données.</p>
---	---

```

1 <?php
2 // Modele.php
3
4 // code .....
5
6 // Renvoie les informations sur une recette
7 function getRecipe($idRecipe) {
8     // Votre code
9 }
10
11 // Renvoie la liste des commentaires associés à une recette
12 function getComments($idRecipe) {
13     // Votre code
14 }
15
16
17 // code .....
18
19 ?>
```

	<p>Vous créez ensuite une nouvelle vue vueRecette.php dont le rôle est d'afficher les informations demandées. Bien entendu, cette vue définit les éléments dynamiques \$titre et \$contenu, puis inclut le gabarit commun (prenez vueAccueil.php comme exemple).</p>
---	--



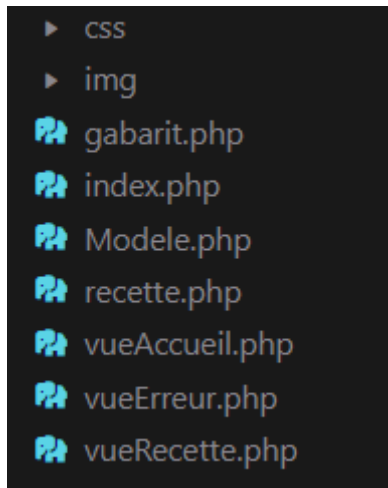
Enfin, créez un nouveau fichier contrôleur, **recette.php**, qui fait le lien entre modèle et vue pour répondre au nouveau besoin. Il reçoit en paramètre l'identifiant de la recette. Elle s'utilise donc sous la forme **recette.php?id=<id du de la recettet>**.

```
1 <?php
2 // recette.php
3
4 require 'Modele.php';
5 try {
6     if (isset($_GET['id'])) {
7         // intval renvoie la valeur numérique du paramètre ou 0 en cas d'échec
8         $id = intval($_GET['id']);
9         if ($id != 0) {
10             $recette = getRecipe($id);
11             $commentaires = getComments($id);
12             require 'vueRecette.php';
13         }
14         else
15             throw new Exception("Identifiant de recette incorrect");
16     }
17     else
18         throw new Exception("Aucun identifiant de recette");
19 }
20 catch (Exception $e) {
21     $msgErreur = $e->getMessage();
22     require 'vueErreur.php';
23 }
24
25 ?>
```

Il faut également modifier la vue **vueAccueil.php** afin d'ajouter un lien vers la page **recette.php** sur l'image de la recette.

```
1 //vueAccueil.php
2
3 <a href="recette.php?id=?= $recipe['rec_id'] ?>">
4     
5 </a>
```

Votre blog d'exemple possède la structure suivante :



Les rôles de chaque élément :

- **Modele.php** représente la partie Modèle (accès aux données) ;
- **vueAccueil.php**, **vueRecette.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 **recette.php** correspondent à la partie Contrôleur (gestion des requêtes entrantes).