Évoluer vers une architecture MVC en PHP

Par Baptiste Pesquet

Date de publication : 27 mars 2013

Dernière mise à jour : 15 octobre 2013

TOUT PUBLIC

Découvrez comment améliorer l'architecture d'un site Web, depuis une organisation classique vers une architecture MVC utilisant un *framework*.

Cet article est une adaptation d'un cours donné aux étudiants de seconde année de BTS SIO (Services Informatiques aux Organisations) au lycée **La Martinière Duchère** de Lyon.

Tous les fichiers source sont disponibles sur le **dépôt GitHub associé à l'article**.

I - Présentation du contexte d'exemple....3
I-A - Base de données....3
I-B - Page principale....3
I-C - Affichage obtenu....4
I-D - Critique de l'exemple....6
II - Mise en place d'une architecture MVC simple....6
II-A - Amélioration de l'exemple....6
II-A-1 - Isolation de l'affichage....6
II-A-2 - Isolation de l'accès aux données....7
II-A-3 - Bilan provisoire....8
II-B - Le modèle MVC....8
II-B-1 - Présentation....8
II-B-2 - Rôles des composants....8
II-B-3 - Interactions entre les composants....8
II-B-4 - Avantages et inconvénients....9
II-B-5 - Différences avec un modèle en couches....9
II-C - Améliorations supplémentaires....10
II-C-1 - Factorisation des éléments d'affichage communs....10
II-C-2 - Factorisation de la connexion à la base....11
II-C-3 - Gestion des erreurs....12
II-D - Bilan provisoire....13
II-E - Application : affichage des détails d'un billet....13
II-E-1 - Prise en compte du nouveau besoin....13
II-E-2 - Affichage obtenu....15
III - Amélioration de l'architecture MVC....16
III-A - Rappels sur l'architecture actuelle....16
III-B - Mise en œuvre d'un contrôleur frontal (front controller)....16
III-C - Réorganisation des fichiers source....18
III-D - Bilan provisoire....19
IV - Passage à une architecture MVC orientée objet....19
IV-A - Aperçu du modèle objet de PHP....19
IV-A-1 - Caractéristiques du modèle objet de PHP....21
IV-A-2 - Spécificités du modèle objet de PHP....21
IV-B - Mise en œuvre du modèle objet de PHP....21
IV-B-1 - Rappels sur l'architecture actuelle....21
IV-B-2 - Passage à un Modèle orienté objet....22
IV-B-3 - Passage à une Vue orientée objet....25
IV-B-4 - Passage à un Contrôleur orienté objet....27
IV-C - Bilan provisoire....30
IV-D - Application : ajout d'un commentaire....31
IV-D-1 - Description du nouveau besoin....31
IV-D-2 - Prise en compte du nouveau besoin....32
V - Construction d'un framework MVC....34
V-A - Où aller maintenant ?....34
V-A-1 - Intérêt d'un framework....34
V-A-2 - Limites de l'architecture actuelle....34
V-B - Étapes de construction du framework....35
V-B-1 - Accès générique aux données....35
V-B-2 - Automatisation du routage de la requête....38
V-B-3 - Mise en place d'URL génériques....42
V-B-4 - Sécurisation des données reçues et affichées....43
V-B-5 - Contraintes sur l'architecture du site....44
V-C - Application : utilisation du framework sur le contexte d'exemple....44
VI - Conclusion et perspectives....46
VI-A - Bilan final....46
VI-B - Pour aller encore plus loin....47
VI-C - Liens utiles....48
VII - Remerciements....48

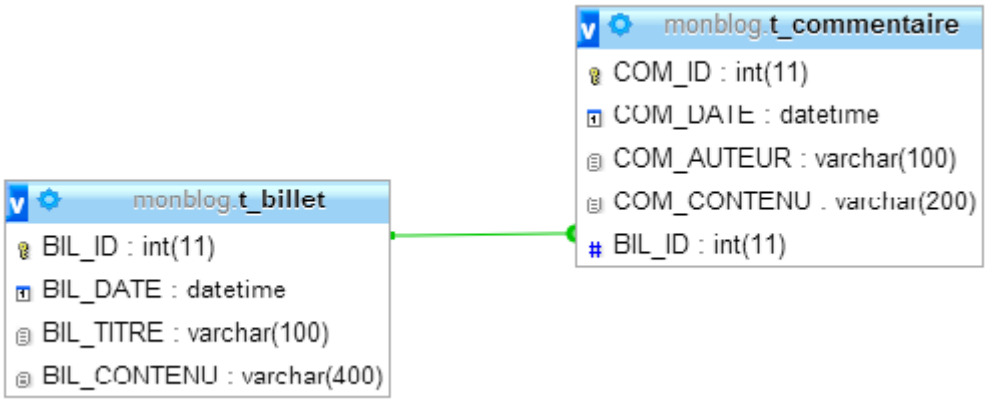
I - Présentation du contexte d'exemple

Nous mettrons en œuvre les principes présentés dans cet article 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 du contexte initial à l'adresse **https://github.com/bpesquet/MonBlog/tree/sans-mvc**

I-A - Base de données

La base de données utilisée est très simple. Elle se compose de deux tables, l'une stockant les billets (articles) du blog et l'autre les commentaires associés aux articles.



Cette base de données contient quelques données de test, insérées par le script SQL ci-dessous.

```
insert into T_BILLET(BIL_DATE, BIL_TITRE, BIL_CONTENU) values
(NOW(), 'Premier billet', 'Bonjour monde ! Ceci est le premier billet sur mon blog.');
insert into T_BILLET(BIL_DATE, BIL_TITRE, BIL_CONTENU) values
(NOW(), 'Au travail', 'Il faut enrichir ce blog dès maintenant.');

insert into T_COMMENTAIRE(COM_DATE, COM_AUTEUR, COM_CONTENU, BIL_ID) values
(NOW(), 'A. Nonyme', 'Bravo pour ce début', 1);
insert into T_COMMENTAIRE(COM_DATE, COM_AUTEUR, COM_CONTENU, BIL_ID) values
(NOW(), 'Moi', 'Merci ! Je vais continuer sur ma lancée', 1);
```

I-B - Page principale

Voici le code source PHP de la page principale **index.php** de notre blog.

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">
```

index.php
```
        <?php
        $bdd = new PDO('mysql:host=localhost;dbname=monblog;charset=utf8',
          'root', '');
        $billets = $bdd->query('select BIL_ID as id, BIL_DATE as date,'
          . ' BIL_TITRE as titre, BIL_CONTENU as contenu from T_BILLET'
          . ' order by BIL_ID desc');
        foreach ($billets as $billet): ?>
          <article>
            <header>
              <h1 class="titreBillet"><?= $billet['titre'] ?></h1>
              <time><?= $billet['date'] ?></time>
            </header>
            <p><?= $billet['contenu'] ?></p>
          </article>
          <hr />
        <?php endforeach; ?>
      </div> <!-- #contenu -->
      <footer id="piedBlog">
        Blog réalisé avec PHP, HTML5 et CSS.
      </footer>
    </div> <!-- #global -->
  </body>
</html>
```

On peut faire les remarques suivantes :

- cette page est écrite en HTML5 et utilise certaines nouvelles balises, comme **<article>** ;
- elle emploie l'affichage abrégé **<?= … ?>** plutôt que **<?php echo … ?>**, ainsi que la **syntaxe alternative** pour la boucle **foreach** ;
- elle utilise l'extension **PDO** de PHP afin d'interagir avec la base de données.

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

I-C - Affichage obtenu

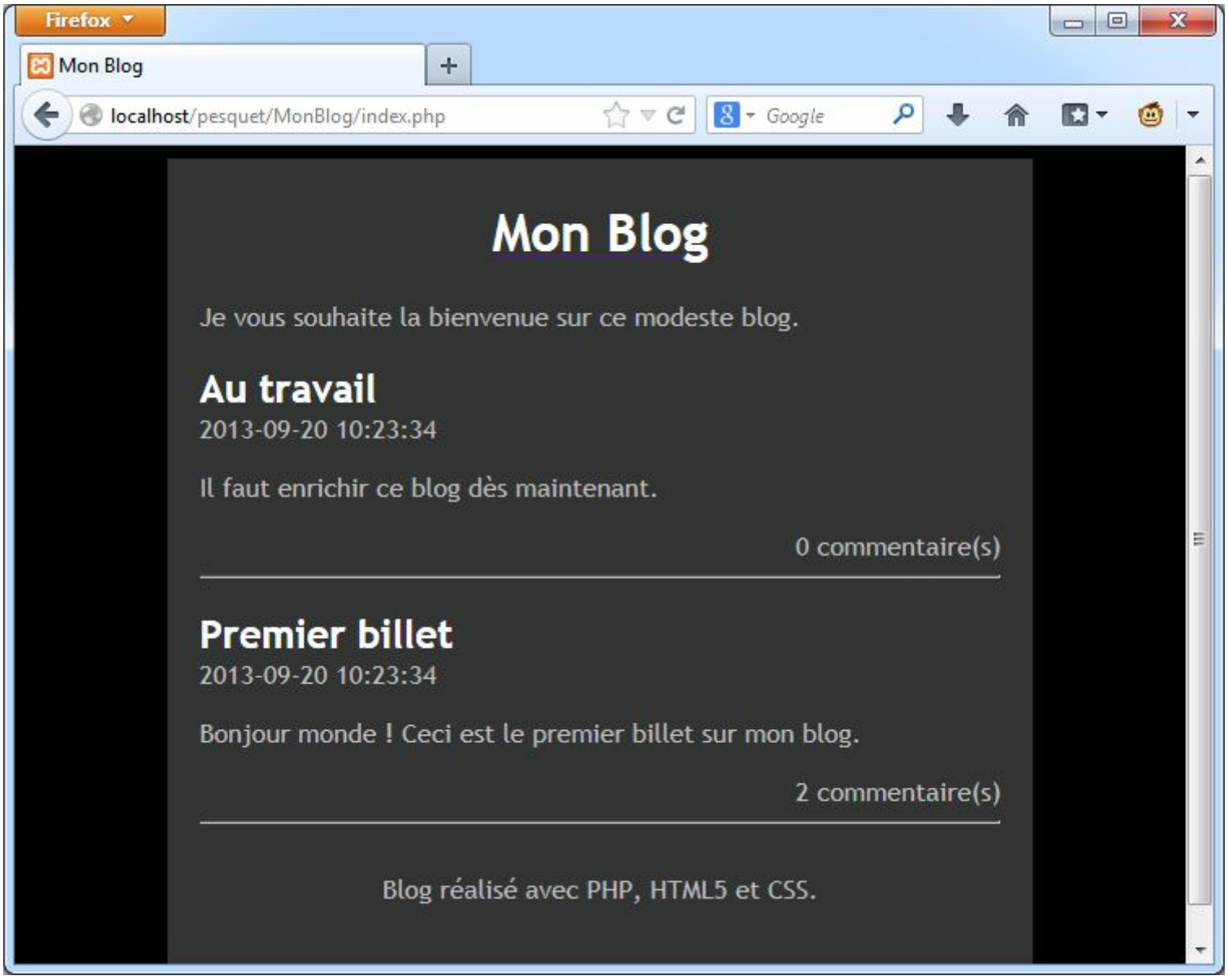
Une feuille de style CSS est utilisée afin d'améliorer le rendu HTML. Voici le code source associé.

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;
}

.titreBillet {
    margin-bottom : 0px;
}
```

style.css

```
#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 obtenu depuis un navigateur client est le suivant.



I-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.

II - Mise en place d'une architecture MVC simple

II-A - Amélioration de l'exemple

II-A-1 - Isolation de l'affichage

Une première amélioration consiste à séparer le code d'accès aux données du code de présentation au sein du fichier **index.php**.

index.php

```
<?php
// Accès aux données
$bdd = new PDO('mysql:host=localhost;dbname=monblog;charset=utf8', 'root', '');
$billets = $bdd->query('select BIL_ID as id, BIL_DATE as date,'
  . ' BIL_TITRE as titre, BIL_CONTENU as contenu from T_BILLET'
  . ' order by BIL_ID desc');
?>

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

Le code est devenu plus lisible, mais les problématiques de présentation et d'accès aux données sont toujours gérées au sein d'un même fichier PHP. En plus de limiter la modularité, ceci est contraire aux bonnes pratiques de développement PHP (norme **PSR-1**).

On peut aller plus loin dans le découplage en regroupant le code d'affichage précédent dans un fichier dédié nommé **vueAccueil.php**.

vueAccueil.php
```
<!doctype html>
<html lang="fr">
<head>
  ...
</head>
<body>
  ...
    <div id="contenu">
      <?php foreach ($billets as $billet): ?>
        <article>
          ...
        </article>
        <hr />
      <?php endforeach; ?>
    </div> <!-- #contenu -->
  ...
</body>
</html>
```

La page principale **index.php** devient alors :

index.php
```
<?php
// Accès aux données
$bdd = new PDO('mysql:host=localhost;dbname=monblog;charset=utf8', 'root', '');
$billets = $bdd->query('select BIL_ID as id, BIL_DATE as date,'
  . ' BIL_TITRE as titre, BIL_CONTENU as contenu from T_BILLET'
  . ' order by BIL_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.*

II-A-2 - Isolation de l'accès aux données

Nous avons amélioré l'architecture de notre page, mais nous pourrions gagner en modularité en isolant le code d'accès aux données dans un fichier PHP dédié. Appelons ce fichier **Modele.php**.

Modele.php
```
<?php

// Renvoie la liste de tous les billets, triés par identifiant décroissant
function getBillets() {
  $bdd = new PDO('mysql:host=localhost;dbname=monblog;charset=utf8', 'root', '');
  $billets = $bdd->query('select BIL_ID as id, BIL_DATE as date,'
    . ' BIL_TITRE as titre, BIL_CONTENU as contenu from T_BILLET'
    . ' order by BIL_ID desc');
  return $billets;
}
```

Dans ce fichier, nous avons déplacé la récupération des billets du blog à l'intérieur d'une fonction nommée **getBillets**.

Le code d'affichage (fichier **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.

index.php
```php
<?php

require 'Modele.php';

$billets = getBillets();

require 'vueAccueil.php';
```

II-A-3 - 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 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*, nous avons rendu notre exemple conforme à un modèle d'architecture très employé sur le Web : le modèle **MVC**.

II-B - Le modèle MVC

II-B-1 - 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.

II-B-2 - Rôles des composants

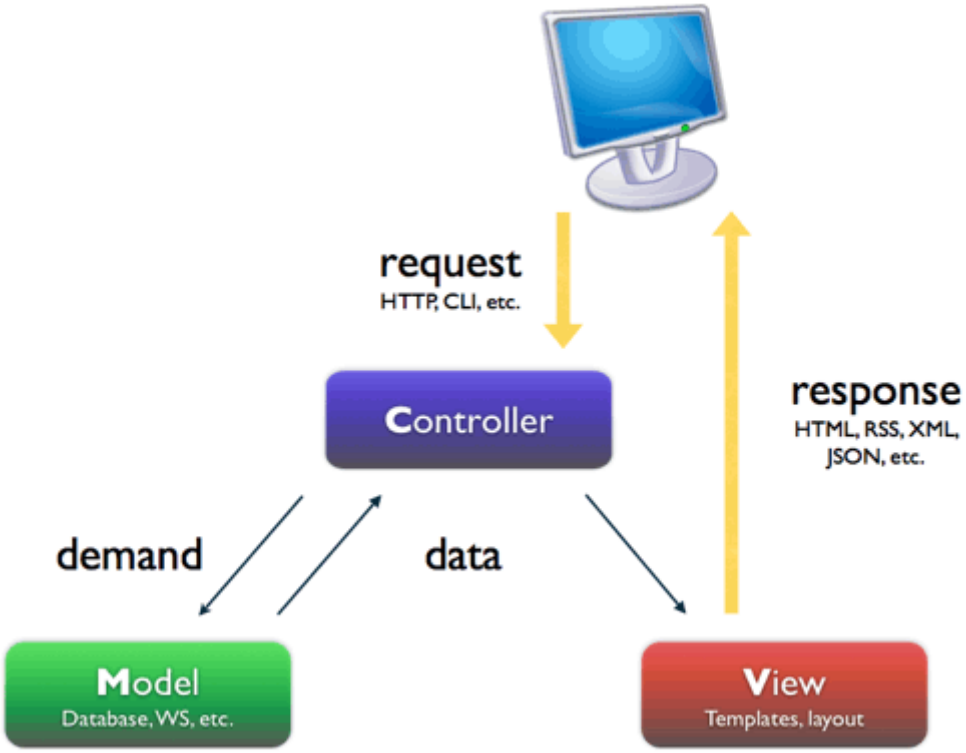
La partie **Modèle** d'une architecture MVC encapsule la logique métier (« *business logic* ») ainsi que l'accès aux données. Il peut s'agir d'un ensemble de fonctions (Modèle procédural) ou de classes (Modèle orienté objet).

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.

II-B-3 - Interactions entre les composants

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



Extrait de la documentation du framework Symfony

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.

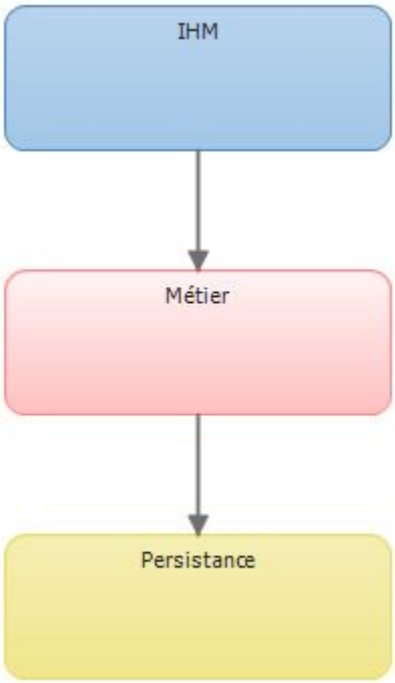
II-B-4 - Avantages et inconvénients

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.

II-B-5 - Différences avec un modèle en couches

Attention à ne pas employer le terme de « couche » à propos du modèle MVC. Un modèle en couches se caractérise par l'interdiction pour une couche non transversale de communiquer au-delà des couches adjacentes. De plus, le nombre de couches n'est pas imposé : il est fonction de la complexité du contexte.



II-C - Améliorations supplémentaires

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

II-C-1 - 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 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**).

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>
</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 :

vueAccueil.php
```
<?php $titre = 'Mon Blog'; ?>

<?php ob_start(); ?>
<?php foreach ($billets as $billet): ?>
  <article>
    <header>
      <h1 class="titreBillet"><?= $billet['titre'] ?></h1>
      <time><?= $billet['date'] ?></time>
    </header>
    <p><?= $billet['contenu'] ?></p>
  </article>
  <hr />
<?php endforeach; ?>
<?php $contenu = ob_get_clean(); ?>

<?php require 'gabarit.php'; ?>
```

Ce code mérite quelques explications :

1 La première ligne définit la valeur de l'élément spécifique **$titre** ;
2 Le deuxième ligne utilise la fonction PHP **ob_start**. Son rôle est de déclencher la mise en tampon du flux HTML de sortie : au lieu d'être envoyé au navigateur, ce flux est stocké en mémoire ;
3 La suite du code (boucle *foreach*) génère les balises HTML **article** associées aux billets du blog. Le flux HTML créé est mis en tampon ;
4 Une fois la boucle terminée, la fonction PHP **ob_get_clean** permet de récupérer dans une variable le flux de sortie mis en tampon depuis l'appel à **ob_start**. La variable se nomme ici **$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.

II-C-2 - 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.

Modele.php
```
<?php

// Renvoie la liste de tous les billets, triés par identifiant décroissant
function getBillets() {
  $bdd = getBdd();
  $billets = $bdd->query('select BIL_ID as id, BIL_DATE as date,'
    . ' BIL_TITRE as titre, BIL_CONTENU as contenu from T_BILLET'
    . ' order by BIL_ID desc');
  return $billets;
}

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

Modele.php
```
}
```

II-C-3 - 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.

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

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

index.php
```
<?php

require 'Modele.php';

try {
  $billets = getBillets();
  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 $titre = 'Mon Blog'; ?>

<?php ob_start() ?>
<p>Une erreur est survenue : <?= $msgErreur ?></p>
<?php $contenu = ob_get_clean(); ?>

<?php require 'gabarit.php'; ?>
```

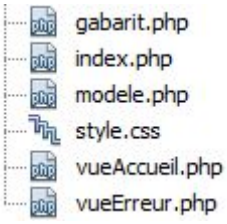
On modifie ensuite le contrôleur pour déclencher le rendu de cette vue en cas d'erreur.

index.php
```
<?php

require 'Modele.php';

try {
  $billets = getBillets();
```

index.php
```
  require 'vueAccueil.php';
}
catch (Exception $e) {
  $msgErreur = $e->getMessage();
  require 'vueErreur.php';
}
```

II-D - 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.

II-E - Application : affichage des détails d'un billet

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

II-E-1 - Prise en compte du nouveau besoin

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

Modele.php
```
...
// Renvoie les informations sur un billet
function getBillet($idBillet) {
  $bdd = getBdd();
  $billet = $bdd->prepare('select BIL_ID as id, BIL_DATE as date,'
    . ' BIL_TITRE as titre, BIL_CONTENU as contenu from T_BILLET'
    . ' where BIL_ID=?');
  $billet->execute(array($idBillet));
  if ($billet->rowCount() == 1)
    return $billet->fetch();  // Accès à la première ligne de résultat
  else
   throw new Exception("Aucun billet ne correspond à l'identifiant '$idBillet'");
}

// Renvoie la liste des commentaires associés à un billet
function getCommentaires($idBillet) {
  $bdd = getBdd();
  $commentaires = $bdd->prepare('select COM_ID as id, COM_DATE as date,'
    . ' COM_AUTEUR as auteur, COM_CONTENU as contenu from T_COMMENTAIRE'
    . ' where BIL_ID=?');
  $commentaires->execute(array($idBillet));
  return $commentaires;
```

Modele.php
```
}
```

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

vueBillet.php
```
<?php $titre = "Mon Blog - " . $billet['titre']; ?>

<?php ob_start(); ?>
<article>
  <header>
    <h1 class="titreBillet"><?= $billet['titre'] ?></h1>
    <time><?= $billet['date'] ?></time>
  </header>
  <p><?= $billet['contenu'] ?></p>
</article>
<hr />
<header>
  <h1 id="titreReponses">Réponses à <?= $billet['titre'] ?></h1>
</header>
<?php foreach ($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, **billet.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 **billet.php?id=<*id du billet*>**.

billet.php
```
<?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) {
      $billet = getBillet($id);
      $commentaires = getCommentaires($id);
      require 'vueBillet.php';
    }
    else
      throw new Exception("Identifiant de billet incorrect");
  }
  else
    throw new Exception("Aucun identifiant de billet");
}
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 **billet.php** sur le titre du billet.

vueAccueil.php
```
...
    <header>
      <a href="<?= "billet.php?id=" . $billet['id'] ?>">
        <h1 class="titreBillet"><?= $billet['titre'] ?></h1>
```

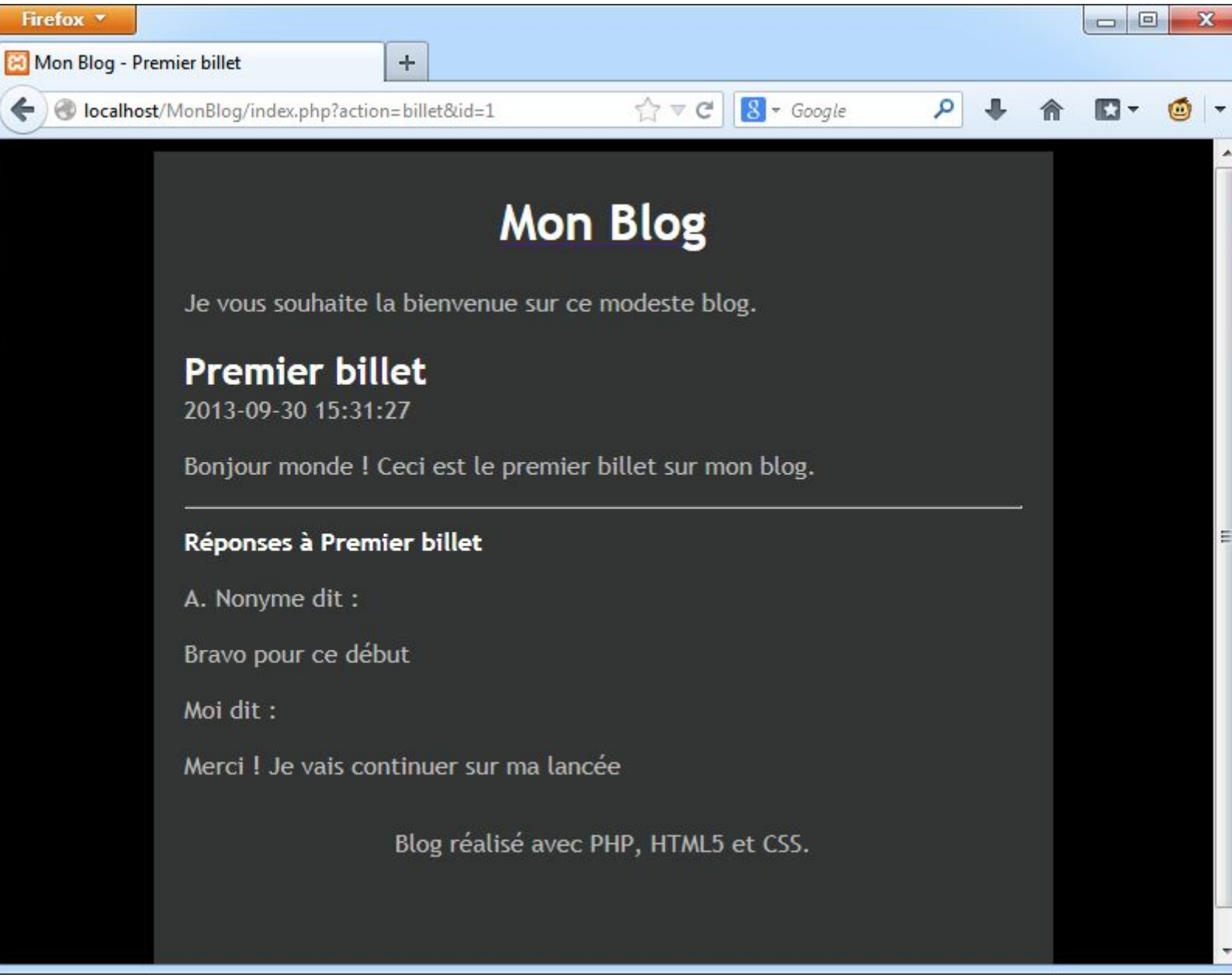
vueAccueil.php
```
        </a>
        <time><?= $billet['date'] ?></time>
      </header>
...
```

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

style.css
```
...

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

II-E-2 - Affichage obtenu

Le résultat obtenu est le suivant :

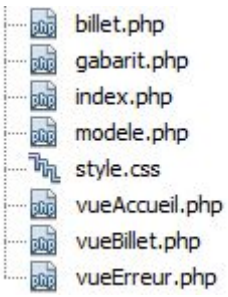


Vous trouverez les fichiers source associés à l'adresse **https://github.com/bpesquet/MonBlog/tree/mvc-simple**.

III - Amélioration de l'architecture MVC

III-A - 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).

III-B - 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**.

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 blog :

- l'exécution de **index.php** permet d'afficher la liste des billets ;
- l'exécution de **billet.php?id=<*id du billet*>** 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 billets 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 « billet », le contrôleur principal déclenchera l'affichage d'un billet ;

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

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

Controleur.php
```php
<?php

require 'Modele.php';

// Affiche la liste de tous les billets du blog
function accueil() {
  $billets = getBillets();
  require 'vueAccueil.php';
}

// Affiche les détails sur un billet
function billet($idBillet) {
  $billet = getBillet($idBillet);
  $commentaires = getCommentaires($idBillet);
  require 'vueBillet.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.

index.php
```php
<?php

require('Controleur.php');

try {
  if (isset($_GET['action'])) {
    if ($_GET['action'] == 'billet') {
      if (isset($_GET['id'])) {
        $idBillet = intval($_GET['id']);
        if ($idBillet != 0)
          billet($idBillet);
        else
          throw new Exception("Identifiant de billet non valide");
      }
      else
        throw new Exception("Identifiant de billet 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 **billet.php** est désormais inutile et peut être supprimé.

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

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

vueAccueil.php
```
    </a>
...
```

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 :

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.

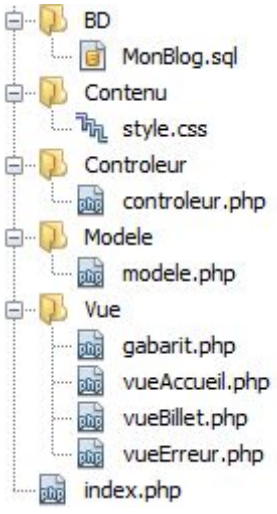
III-C - Réorganisation des fichiers source

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**, **vueBillet.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.

Vous trouverez les fichiers source associés à l'adresse **https://github.com/bpesquet/MonBlog/tree/mvc-procedural**.

III-D - Bilan provisoire

Notre blog d'exemple 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.

IV - Passage à une architecture MVC orientée objet

IV-A - Aperçu du modèle objet de PHP

Nous allons étudier le modèle objet de PHP, et notamment ses spécificités par rapport à d'autres langages comme Java ou C#, à travers l'exemple classique des comptes bancaires.

Voici la définition PHP d'une classe **CompteBancaire**.

CompteBancaire.php
```php
<?php

class CompteBancaire
{
    private $devise;
    private $solde;
    private $titulaire;

    public function __construct($devise, $solde, $titulaire)
    {
        $this->devise = $devise;
        $this->solde = $solde;
        $this->titulaire = $titulaire;
    }

    public function getDevise()
    {
        return $this->devise;
    }

    public function getSolde()
    {
        return $this->solde;
    }

    protected function setSolde($solde)
    {
        $this->solde = $solde;
    }

    public function getTitulaire()
    {
        return $this->titulaire;
    }

    public function crediter($montant) {
        $this->solde += $montant;
    }

    public function __toString()
    {
        return "Le solde du compte de $this->titulaire est de " .
            $this->solde . " " . $this->devise;
    }
}
```

En complément, voici la définition d'une classe **CompteEpargne**.

CompteEpargne.php
```php
<?php

require_once 'CompteBancaire.php';

class CompteEpargne extends CompteBancaire
{
    private $tauxInteret;

    public function __construct($devise, $solde, $titulaire, $tauxInteret)
    {
        parent::__construct($devise, $solde, $titulaire);
        $this->tauxInteret = $tauxInteret;
    }

    public function getTauxInteret()
    {
        return $this->tauxInteret;
    }

    public function calculerInterets($ajouterAuSolde = false)
    {
        $interets = $this->getSolde() * $this->tauxInteret;
        if ($ajouterAuSolde == true)
            $this->setSolde($this->getSolde() + $interets);
        return $interets;
    }

    public function __toString()
    {
        return parent::__toString() .
            '. Son taux d\'interet est de ' . $this->tauxInteret * 100 . '%.';
    }
}
```

Voici un exemple d'utilisation de ces deux classes.

poo.php
```php
<?php

require 'CompteBancaire.php';
require 'CompteEpargne.php';

$compteJean = new CompteBancaire("euros", 150, "Jean");
echo $compteJean . '<br />';
$compteJean->crediter(100);
echo $compteJean . '<br />';

$comptePaul = new CompteEpargne("dollars", 200, "Paul", 0.05);
echo $comptePaul . '<br />';
echo 'Interets pour ce compte : ' . $comptePaul->calculerInterets() .
    ' ' . $comptePaul->getDevise() . '<br />';
$comptePaul->calculerInterets(true);
echo $comptePaul . '<br />';
```

Enfin, voici le résultat de son exécution.

Le solde du compte de Jean est de 150 euros
Le solde du compte de Jean est de 250 euros
Le solde du compte de Paul est de 200 dollars. Son taux d'interet est de 5%.
Interets pour ce compte : 10 dollars
Le solde du compte de Paul est de 210 dollars. Son taux d'interet est de 5%.

IV-A-1 - Caractéristiques du modèle objet de PHP

L'observation des exemples précédents nous permet de retrouver certains concepts bien connus de la POO, repris par PHP :

- une classe se compose d'attributs et de méthodes ;
- le mot-clé **class** permet de définir une classe ;
- les différents niveaux d'accessibilité sont **public**, **protected** et **private** ;
- le mot-clé **extends** permet de définir une classe dérivée (comme en Java) ;
- le mot-clé **$this** permet d'accéder aux membres de l'objet courant.

IV-A-2 - Spécificités du modèle objet de PHP

Même s'il est similaire à ceux de C#, Java ou C++, le modèle objet de PHP possède certaines particularités :

- PHP étant un langage à typage dynamique, on ne précise pas les types des attributs et des méthodes, mais seulement leur niveau d'accessibilité ;
- le mot-clé **function** permet de déclarer une méthode, quelle que soit sa valeur de retour ;
- le mot-clé **parent** permet d'accéder au parent de l'objet courant. Il joue en PHP le même rôle que **base** en C# et **super** en Java ;
- le constructeur d'une classe s'écrit **__construct** ;
- la méthode **__toString** détermine comment l'objet est affiché en tant que chaîne de caractères ;
- on peut redéfinir (*override*) une méthode, comme ici **__toString**, sans mot-clé particulier ;
- le mot-clé **$this** est obligatoire pour accéder aux membres de l'objet courant. Son utilisation est optionnelle en C# et en Java, et souvent limitée à la levée des ambiguïtés entre attributs et paramètres ;
- il est possible de définir une valeur par défaut pour les paramètres d'une méthode. Elle est utilisée lorsque l'argument (paramètre effectif) n'est pas précisé au moment de l'appel.

Remarques :

- Les méthodes **__construct** et **__toString** font partie de ce qu'on appelle les **méthodes magiques**.
- L'instruction **require_once** est similaire à **require** mais n'inclut le fichier demandé qu'une seule fois. Elle est utile pour éviter, comme ici, les définitions multiples de classes.

IV-B - Mise en œuvre du modèle objet de PHP

Munis de cette connaissance minimale du modèle objet de PHP, nous pouvons à présent améliorer l'architecture de notre site d'exemple en tirant parti des possibilités de la POO.

IV-B-1 - Rappels sur l'architecture actuelle

Pour mémoire, voici la définition actuelle de notre partie **Modèle** puis de notre partie **Contrôleur**.

Modele.php
```php
<?php

// Renvoie la liste des billets du blog
function getBillets() {
  $bdd = getBdd();
  $billets = $bdd->query('select BIL_ID as id, BIL_DATE as date,'
    . ' BIL_TITRE as titre, BIL_CONTENU as contenu from T_BILLET'
    . ' order by BIL_ID desc');
  return $billets;
}

// Renvoie les informations sur un billet
```

Modele.php
```php
function getBillet($idBillet) {
  $bdd = getBdd();
  $billet = $bdd->prepare('select BIL_ID as id, BIL_DATE as date,'
    . ' BIL_TITRE as titre, BIL_CONTENU as contenu from T_BILLET'
    . ' where BIL_ID=?');
  $billet->execute(array($idBillet));
  if ($billet->rowCount() == 1)
    return $billet->fetch();  // Accès à la première ligne de résultat
  else
   throw new Exception("Aucun billet ne correspond à l'identifiant '$idBillet'");
}

// Renvoie la liste des commentaires associés à un billet
function getCommentaires($idBillet) {
  $bdd = getBdd();
  $commentaires = $bdd->prepare('select COM_ID as id, COM_DATE as date,'
    . ' COM_AUTEUR as auteur, COM_CONTENU as contenu from T_COMMENTAIRE'
    . ' where BIL_ID=?');
  $commentaires->execute(array($idBillet));
  return $commentaires;
}

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

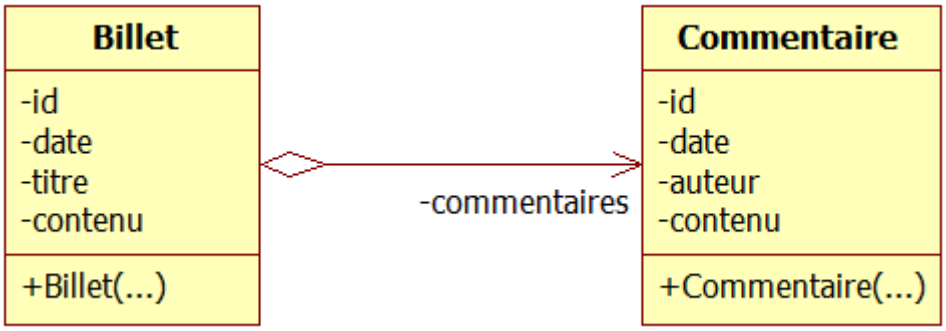
On rappelle que le rôle de la partie Modèle est de rassembler la logique métier et l'accès aux données.

Controleur.php
```php
<?php

require 'Modele/Modele.php';

// Affiche la liste de tous les billets du blog
function accueil() {
  $billets = getBillets();
  require 'Vue/vueAccueil.php';
}

// Affiche les détails sur un billet
function billet($idBillet) {
  $billet = getBillet($idBillet);
  $commentaires = getCommentaires($idBillet);
  require 'Vue/vueBillet.php';
}

// Affiche une erreur
function erreur($msgErreur) {
  require 'Vue/vueErreur.php';
}
```

La partie Contrôleur, quant à elle, gère la dynamique d'une application MVC.

IV-B-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 **Billet** et **Commentaire**.



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**.

Modele.php

```php
<?php

class Modele {

  // Renvoie la liste des billets du blog
  public function getBillets() {
    $bdd = $this->getBdd();
    $billets = $bdd->query('select BIL_ID as id, BIL_DATE as date,'
      . ' BIL_TITRE as titre, BIL_CONTENU as contenu from T_BILLET'
      . ' order by BIL_ID desc');
    return $billets;
  }

  // Renvoie les informations sur un billet
  public function getBillet($idBillet) {
    $bdd = $this->getBdd();
    $billet = $bdd->prepare('select BIL_ID as id, BIL_DATE as date,'
      . ' BIL_TITRE as titre, BIL_CONTENU as contenu from T_BILLET'
      . ' where BIL_ID=?');
    $billet->execute(array($idBillet));
    if ($billet->rowCount() == 1)
      return $billet->fetch();  // Accès à la première ligne de résultat
    else
      throw new Exception("Aucun billet ne correspond à l'identifiant '$idBillet'");
  }

  // Renvoie la liste des commentaires associés à un billet
  public function getCommentaires($idBillet) {
    $bdd = $this->getBdd();
    $commentaires = $bdd->prepare('select COM_ID as id, COM_DATE as date,'
    . ' COM_AUTEUR as auteur, COM_CONTENU as contenu from T_COMMENTAIRE'
    . ' where BIL_ID=?');
    $commentaires->execute(array($idBillet));
    return $commentaires;
  }

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

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.

Billet
```php
<?php

require_once 'Modele/Modele.php';

class Billet extends Modele {

  // Renvoie la liste des billets du blog
  public function getBillets() {
    $sql = 'select BIL_ID as id, BIL_DATE as date,'
      . ' BIL_TITRE as titre, BIL_CONTENU as contenu from T_BILLET'
      . ' order by BIL_ID desc';
    $billets = $this->executerRequete($sql);
    return $billets;
  }

  // Renvoie les informations sur un billet
  public function getBillet($idBillet) {
    $sql = 'select BIL_ID as id, BIL_DATE as date,'
      . ' BIL_TITRE as titre, BIL_CONTENU as contenu from T_BILLET'
      . ' where BIL_ID=?';
    $billet = $this->executerRequete($sql, array($idBillet));
    if ($billet->rowCount() == 1)
      return $billet->fetch();  // Accès à la première ligne de résultat
    else
      throw new Exception("Aucun billet ne correspond à l'identifiant '$idBillet'");
    }
}
```

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
<?php

abstract class Modele {

  // Objet PDO d'accès à la BD
  private $bdd;

  // Exécute une requête SQL éventuellement paramétrée
  protected function executerRequete($sql, $params = null) {
    if ($params == null) {
      $resultat = $this->getBdd()->query($sql);    // exécution directe
    }
    else {
      $resultat = $this->getBdd()->prepare($sql);  // requête préparée
      $resultat->execute($params);
    }
    return $resultat;
  }

  // Renvoie un objet de connexion à la BD en initialisant la connexion au besoin
  private function getBdd() {
    if ($this->bdd == null) {
      // Création de la connexion
      $this->bdd = new PDO('mysql:host=localhost;dbname=monblog;charset=utf8',
        'root', '', array(PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION));
    }
    return $this->bdd;
  }

}
```

On remarque au passage que la technologie d'accès à la base est totalement masquée aux modèles concrets, et que **Modele** utilise la technique du chargement tardif (« **/http://en.wikipedia.org/wiki/Lazy_loading** ») pour retarder l'instanciation de l'objet **$bdd** à sa première utilisation.

On peut écrire une classe **Commentaire** sur le même modèle que la classe **Billet**.

Commentaire.php
```php
<?php

require_once 'Modele/Modele.php';

class Commentaire extends Modele {

  // Renvoie la liste des commentaires associés à un billet
  public function getCommentaires($idBillet) {
    $sql = 'select COM_ID as id, COM_DATE as date,'
      . ' COM_AUTEUR as auteur, COM_CONTENU as contenu from T_COMMENTAIRE'
      . ' where BIL_ID=?';
    $commentaires = $this->executerRequete($sql, array($idBillet));
    return $commentaires;

  }
}
```

À présent, l'architecture de la partie **Modele** tire parti des avantages de la POO (encapsulation, héritage). Cette architecture facilite les évolutions : si le contexte métier s'enrichit (exemple : gestion des auteurs de billets), il suffit de créer une nouvelle classe modèle dérivée de **Modele** (ici : **Auteur**) qui s'appuiera sur les services communs fournis par sa superclasse.

IV-B-3 - Passage à une Vue orientée objet

Pour l'instant, nos vues sont des fichiers HTML/PHP qui exploitent des variables PHP contenant les données dynamiques. Elles utilisent un gabarit commun regroupant les éléments d'affichage communs. Voici par exemple la vue d'affichage d'un billet et de ses commentaires.

vueBillet.php
```php
<?php $titre = "Mon Blog - " . $billet['titre']; ?>

<?php ob_start(); ?>
<article>
    <header>
        <h1 class="titreBillet"><?= $billet['titre'] ?></h1>
        <time><?= $billet['date'] ?></time>
    </header>
    <p><?= $billet['contenu'] ?></p>
</article>
<hr />
<header>
    <h1 id="titreReponses">Réponses à <?= $billet['titre'] ?></h1>
</header>
<?php foreach ($commentaires as $commentaire): ?>
    <p><?= $commentaire['auteur'] ?> dit :</p>
    <p><?= $commentaire['contenu'] ?></p>
<?php endforeach; ?>
<?php $contenu = ob_get_clean(); ?>

<?php require 'gabarit.php'; ?>
```

Le gabarit centralise les éléments d'affichage communs et utilise les variables **$titre** et **$contenu** pour intégrer les éléments spécifiques.

gabarit.php
```php
<!doctype html>
```

gabarit.php
```
<html lang="fr">
    <head>
        <meta charset="UTF-8" />
        <link rel="stylesheet" href="Contenu/style.css" />
        <title><?= $titre ?></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">
                <?= $contenu ?>
            </div> <!-- #contenu -->
            <footer id="piedBlog">
                Blog réalisé avec PHP, HTML5 et CSS.
            </footer>
        </div> <!-- #global -->
    </body>
</html>
```

Cette approche simple souffre de plusieurs limitations :

- les appels aux fonctions PHP **ob_start** et **ob_get_clean** sont dupliqués ;
- la génération des fichiers vue (y compris dans le contrôleur) utilise directement la fonction PHP **require**, sans protection contre une éventuelle absence du fichier demandé.

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

Vue.php
```
class Vue {

  // Nom du fichier associé à la vue
  private $fichier;
  // Titre de la vue (défini dans le fichier vue)
  private $titre;

  public function __construct($action) {
    // Détermination du nom du fichier vue à partir de l'action
    $this->fichier = "Vue/vue" . $action . ".php";
  }

  // Génère et affiche la vue
  public function generer($donnees) {
    // Génération de la partie spécifique de la vue
    $contenu = $this->genererFichier($this->fichier, $donnees);
    // Génération du gabarit commun utilisant la partie spécifique
    $vue = $this->genererFichier('Vue/gabarit.php',
      array('titre' => $this->titre, 'contenu' => $contenu));
    // Renvoi de la vue au navigateur
    echo $vue;
  }

  // Génère un fichier vue et renvoie le résultat produit
  private function genererFichier($fichier, $donnees) {
    if (file_exists($fichier)) {
      // Rend les éléments du tableau $donnees accessibles dans la vue
      extract($donnees);
      // Démarrage de la temporisation de sortie
      ob_start();
      // Inclut le fichier vue
      // Son résultat est placé dans le tampon de sortie
      require $fichier;
      // Arrêt de la temporisation et renvoi du tampon de sortie
      return ob_get_clean();
    }
```

Vue.php
```
    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 **$titre**) et son contenu (variable locale **$contenu**). 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->titre** et supprimer les appels aux fonctions PHP de temporisation. Voici par exemple la nouvelle vue d'accueil.

vueAccueil.php
```
<?php $this->titre = "Mon Blog"; ?>

<?php foreach ($billets as $billet):
    ?>
    <article>
        <header>
            <a href="<?= "index.php?action=billet&id=" . $billet['id'] ?>">
                <h1 class="titreBillet"><?= $billet['titre'] ?></h1>
            </a>
            <time><?= $billet['date'] ?></time>
        </header>
        <p><?= $billet['contenu'] ?></p>
    </article>
    <hr />
<?php endforeach; ?>
```

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

IV-B-4 - 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**.

Controleur.php
```
<?php

require 'Modele/Modele.php';

// Affiche la liste de tous les billets du blog
function accueil() {
  $billets = getBillets();
  require 'Vue/vueAccueil.php';
}

// Affiche les détails sur un billet
function billet($idBillet) {
  $billet = getBillet($idBillet);
  $commentaires = getCommentaires($idBillet);
  require 'Vue/vueBillet.php';
}

// Affiche une erreur
function erreur($msgErreur) {
  require 'Vue/vueErreur.php';
```

Controleur.php
```
}
```

index.php
```
<?php

require 'Controleur/Controleur.php';

try {
  if (isset($_GET['action'])) {
    if ($_GET['action'] == 'billet') {
      if (isset($_GET['id'])) {
        $idBillet = intval($_GET['id']);
        if ($idBillet != 0) {
          billet($idBillet);
        }
        else
          throw new Exception("Identifiant de billet non valide");
      }
      else
        throw new Exception("Identifiant de billet non défini");
    }
    else
      throw new Exception("Action non valide");
  }
  else {  // aucune action définie : affichage de l'accueil
    accueil();
  }
}
catch (Exception $e) {
  erreur($e->getMessage());
}
```

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 **ControleurBillet** 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/Billet.php';
require_once 'Vue/Vue.php';

class ControleurAccueil {

  private $billet;

  public function __construct() {
    $this->billet = new Billet();
  }

  // Affiche la liste de tous les billets du blog
  public function accueil() {
    $billets = $this->billet->getBillets();
    $vue = new Vue("Accueil");
    $vue->generer(array('billets' => $billets));
  }
}
```

ControleurBillet.php
```php
<?php

require_once 'Modele/Billet.php';
require_once 'Modele/Commentaire.php';
require_once 'Vue/Vue.php';

class ControleurBillet {

  private $billet;
  private $commentaire;

  public function __construct() {
    $this->billet = new Billet();
    $this->commentaire = new Commentaire();
  }

  // Affiche les détails sur un billet
  public function billet($idBillet) {
    $billet = $this->billet->getBillet($idBillet);
    $commentaires = $this->commentaire->getCommentaires($idBillet);
    $vue = new Vue("Billet");
    $vue->generer(array('billet' => $billet, '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 'Controleur/ControleurAccueil.php';
require_once 'Controleur/ControleurBillet.php';
require_once 'Vue/Vue.php';

class Routeur {

  private $ctrlAccueil;
  private $ctrlBillet;

  public function __construct() {
    $this->ctrlAccueil = new ControleurAccueil();
    $this->ctrlBillet = new ControleurBillet();
  }

  // Traite une requête entrante
  public function routerRequete() {
    try {
      if (isset($_GET['action'])) {
        if ($_GET['action'] == 'billet') {
          if (isset($_GET['id'])) {
            $idBillet = intval($_GET['id']);
            if ($idBillet != 0) {
              $this->ctrlBillet->billet($idBillet);
            }
            else
              throw new Exception("Identifiant de billet non valide");
          }
          else
            throw new Exception("Identifiant de billet non défini");
        }
        else
          throw new Exception("Action non valide");
      }
```

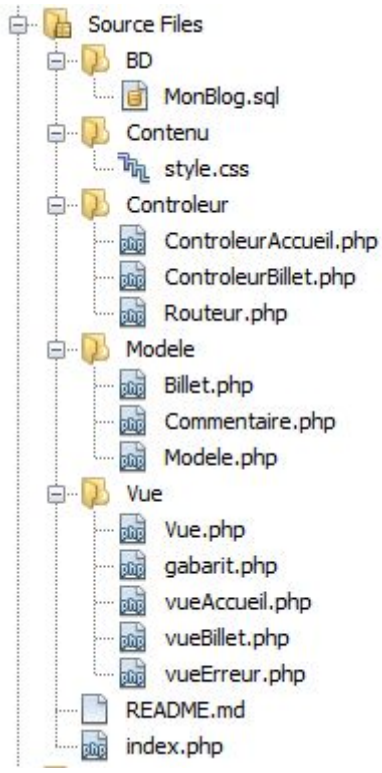
Routeur.php
```
        else {  // aucune action définie : affichage de l'accueil
          $this->ctrlAccueil->accueil();
        }
      }
      catch (Exception $e) {
        $this->erreur($e->getMessage());
      }
    }

    // Affiche une erreur
    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.

index.php
```
<?php

require 'Controleur/Routeur.php';

$routeur = new Routeur();
$routeur->routerRequete();
```

IV-C - 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.



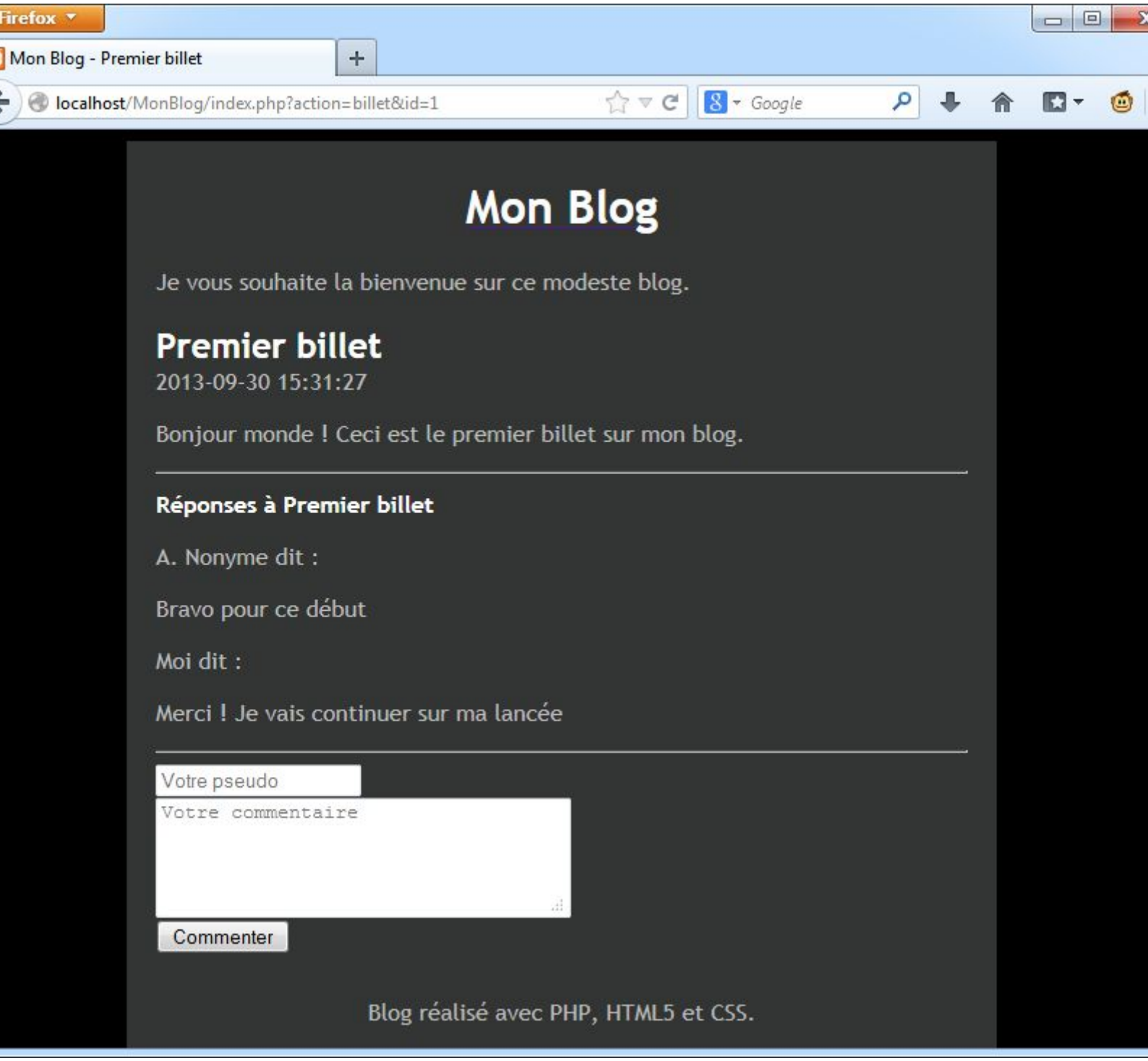
L'ajout de nouvelles fonctionnalités se fait à présent en trois étapes :

- 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.

IV-D - Application : ajout d'un commentaire

IV-D-1 - Description du nouveau besoin

On souhaite maintenant que l'affichage des détails sur un billet permette d'ajouter un nouveau commentaire. Le remplissage des champs Auteur et Commentaire est obligatoire. Le clic sur le bouton Commenter déclenche l'insertion du commentaire dans la base de données et la réactualisation de la page Web.



IV-D-2 - Prise en compte du nouveau besoin

On commence par ajouter à la classe **Commentaire** une méthode permettant d'insérer un nouveau commentaire dans la BD.

Commentaire.php

```
...
  // Ajoute un commentaire dans la base
  public function ajouterCommentaire($auteur, $contenu, $idBillet) {
    $sql = 'insert into T_COMMENTAIRE(COM_DATE, COM_AUTEUR, COM_CONTENU, BIL_ID)'
      . ' values(?, ?, ?, ?)';
```

Commentaire.php
```
    $date = date(DATE_W3C);  // Récupère la date courante
    $this->executerRequete($sql, array($date, $auteur, $contenu, $idBillet));
  }
...
```

On ajoute ensuite à la vue d'un billet le formulaire HTML nécessaire pour saisir un commentaire.

vueBillet.php
```
...
<form method="post" action="index.php?action=commenter">
    <input id="auteur" name="auteur" type="text" placeholder="Votre pseudo"
           required /><br />
    <textarea id="txtCommentaire" name="contenu" rows="4"
              placeholder="Votre commentaire" required></textarea><br />
    <input type="hidden" name="id" value="<?= $billet['id'] ?>" />
    <input type="submit" value="Commenter" />
</form>
```

Remarque : l'action associée à la soumission du formulaire est nommée **commenter**.

Au passage, on met à jour la feuille de style afin de définir la taille par défaut de la zone de texte du commentaire.

style.css
```
...

#txtCommentaire {
    width: 50%;
}
```

Il faut également ajouter au contrôleur une méthode associée à cette action.

ControleurBillet.php
```
...
  // Ajoute un commentaire à un billet
  public function commenter($auteur, $contenu, $idBillet) {
    // Sauvegarde du commentaire
    $this->commentaire->ajouterCommentaire($auteur, $contenu, $idBillet);
    // Actualisation de l'affichage du billet
    $this->billet($idBillet);
  }
...
```

Cette action consiste à appeler un service du Modèle, puis à exécuter l'action d'affichage du billet afin d'obtenir un résultat actualisé.

Enfin, on met à jour le routeur afin de router une requête d'ajout de commentaire vers la nouvelle action. Au passage, on en profite pour simplifier la méthode de routage (qui tend à devenir complexe) en faisant appel à une méthode privée de recherche d'un paramètre dans un tableau. Cette méthode permet de rechercher un paramètre dans **$_GET** ou **$_POST** en fonction du besoin.

Routeur.php
```
...
  // Recherche un paramètre dans un tableau
  private function getParametre($tableau, $nom) {
    if (isset($tableau[$nom])) {
      return $tableau[$nom];
    }
    else
      throw new Exception("Paramètre '$nom' absent");
  }
...
```

La méthode de routage d'une requête est mise à jour.

Routeur.php
```
...
  if (isset($_GET['action'])) {
    if ($_GET['action'] == 'billet') {
      $idBillet = intval($this->getParametre($_GET, 'id'));
      if ($idBillet != 0) {
        $this->ctrlBillet->billet($idBillet);
      }
      else
        throw new Exception("Identifiant de billet non valide");
    }
    else if ($_GET['action'] == 'commenter') {
      $auteur = $this->getParametre($_POST, 'auteur');
      $contenu = $this->getParametre($_POST, 'contenu');
      $idBillet = $this->getParametre($_POST, 'id');
      $this->ctrlBillet->commenter($auteur, $contenu, $idBillet);
    }
    else
      throw new Exception("Action non valide");
  }
  else {  // aucune action définie : affichage de l'accueil
    $this->ctrlAccueil->accueil();
  }
...
```

Vous trouverez les fichiers source associés à l'adresse **https://github.com/bpesquet/MonBlog/tree/mvc-objet**.

V - Construction d'un framework MVC

V-A - Où aller maintenant ?

Nous avons parcouru beaucoup de chemin depuis le début de cet article. D'une simple page PHP, notre blog d'exemple s'est transformé en un site Web architecturé selon les principes du modèle MVC. Il dispose d'un contrôleur frontal, d'un routeur orienté objet, ainsi que de classes abstraites fournissant des services communs. Pourquoi ne pas aller au bout de la logique objet en isolant ces services communs au sein d'un ***framework*** dont les bases sont déjà construites ?

V-A-1 - Intérêt d'un framework

Un *framework* fournit un ensemble de services de base, généralement sous la forme de classes en interaction. À condition de respecter l'architecture qu'il préconise (pratiquement toujours une déclinaison du modèle MVC), un *framework* PHP libère le développeur de nombreuses tâches techniques comme le routage des requêtes, la sécurité, la gestion du cache, etc. Cela lui permet de se concentrer sur l'essentiel, c'est-à-dire ses tâches métier. Il existe une grande quantité de *frameworks* PHP. Parmi les plus connus, citons **Symfony**, **Zend Framework** ou encore **CodeIgniter**.

Notre petit *framework* n'atteindra évidemment pas la richesse fonctionnelle et le niveau de qualité des exemples précédents. Son but est d'illustrer « de l'intérieur » et aussi simplement que possible les bases du fonctionnement d'un *framework* PHP.

V-A-2 - Limites de l'architecture actuelle

Nous allons profiter de la mise en place du framework pour remédier à certains points faibles de l'architecture actuelle :

- les paramètres d'accès à la base de données (serveur, nom de la BD, identifiant de connexion, mot de passe) ne sont pas configurables ;

- chaque classe modèle instancie un objet PDO d'accès à la BD, ce qui est sous-optimal du point de vue des performances ;
- les URL actuelles, du type **monsite.fr/index.php?action=yyy&id=zzz**, sont moins lisibles que des URL du type **monsite.fr/action/id** ;
- le routage de la requête (choix de l'action à exécuter) est fait manuellement par le routeur ;
- les paramètres des requêtes ne sont pas filtrés en entrée ;
- les données insérées dans les vues ne sont pas nettoyées, ce qui accroît le risque de **failles XSS**.

V-B - Étapes de construction du framework

Ce paragraphe constitue la partie la plus complexe de l'article. Il fait appel à des concepts avancés du développement Web et de la POO.
Toutefois, il n'est pas nécessaire de comprendre tout son détail pour pouvoir utiliser le framework ainsi construit.

V-B-1 - Accès générique aux données

Commençons la construction du *framework* par la partie Modèle. Les classes **Billet** et **Commentaire** sont directement liées à notre blog d'exemple et ne peuvent pas être réutilisées dans un autre contexte. En revanche, la classe abstraite **Modele** fournit des services totalement indépendants du schéma relationnel. On peut envisager de l'intégrer à notre *framework*.

Avant cela, il nous reste un problème à résoudre. Comme nous l'avons dit plus haut, l'accès à la base de données dans la classe abstraite **Modele** n'est pas paramétrable : les valeurs de la base MonBlog sont « en dur » dans son code source.

Modele.php
```php
<?php

abstract class Modele {

  // Objet PDO d'accès à la BD
  private $bdd;

  // Exécute une requête SQL éventuellement paramétrée
  protected function executerRequete($sql, $params = null) {
    if ($params == null) {
      $resultat = $this->getBdd()->query($sql);    // exécution directe
    }
    else {
      $resultat = $this->getBdd()->prepare($sql);  // requête préparée
      $resultat->execute($params);
    }
    return $resultat;
  }

  // Renvoie un objet de connexion à la BD en initialisant la connexion au besoin
  private function getBdd() {
    if ($this->bdd == null) {
      // Création de la connexion
      $this->bdd = new PDO('mysql:host=localhost;dbname=monblog;charset=utf8',
        'root', '', array(PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION));
    }
    return $this->bdd;
  }
}
```

Pour que cette classe soit totalement générique et donc intégrable à un *framework*, il faudrait pouvoir définir les paramètres de connexion à la BD sans modifier son code source. Pour cela, nous allons créer un nouveau composant dont le rôle sera de gérer la configuration du site. Ce composant prend la forme d'une classe appelée logiquement **Configuration**.

Configuration.php
```php
<?php

class Configuration {

  private static $parametres;

  // Renvoie la valeur d'un paramètre de configuration
  public static function get($nom, $valeurParDefaut = null) {
    if (isset(self::getParametres()[$nom])) {
      $valeur = self::getParametres()[$nom];
    }
    else {
      $valeur = $valeurParDefaut;
    }
    return $valeur;
  }

  // Renvoie le tableau des paramètres en le chargeant au besoin
  private static function getParametres() {
    if (self::$parametres == null) {
      $cheminFichier = "Config/prod.ini";
      if (!file_exists($cheminFichier)) {
        $cheminFichier = "Config/dev.ini";
      }
      if (!file_exists($cheminFichier)) {
        throw new Exception("Aucun fichier de configuration trouvé");
      }
      else {
        self::$parametres = parse_ini_file($cheminFichier);
      }
    }
    return self::$parametres;
  }
}
```

Cette classe encapsule un tableau associatif clés/valeurs (attribut **$parametres**) stockant les valeurs des paramètres de configuration. Ce tableau est statique (un seul exemplaire par classe), ce qui permet de l'utiliser sans instancier d'objet **Configuration**.

La classe dispose d'une méthode statique publique nommée **get**() qui permet de rechercher la valeur d'un paramètre à partir de son nom. Si le paramètre en question est trouvé dans le tableau associatif, sa valeur est renvoyée. Sinon, une valeur par défaut est renvoyée. On rencontre au passage le mot-clé PHP **self** qui permet de faire référence à un membre statique.

Enfin, la méthode statique privée **getParametres**() effectue le chargement tardif du fichier contenant les paramètres de configuration. Afin de faire cohabiter sur un même serveur une configuration de développement et une configuration de production, deux fichiers sont recherchés dans le répertoire **Config** du site : **dev.ini** (cherché en premier) et **prod.ini**. La lecture du fichier de configuration utilise la fonction PHP **parse_ini_file**(). Celle-ci instancie et renvoie un tableau associatif immédiatement attribué à l'attribut **$parametres**.

Grâce à cette classe, on peut externaliser la configuration d'un site en dehors de son code source. Voici par exemple le fichier de configuration correspondant à notre blog d'exemple.

dev.ini
```ini
; Configuration pour le développement

[BD]
dsn = 'mysql:host=localhost;dbname=monblog;charset=utf8'
login = root
mdp =
```

Un changement de paramètres de connexion, par exemple pour employer un autre utilisateur que **root**, nécessite uniquement une mise à jour de ce fichier de configuration.

De plus, nous pouvons rendre la classe abstraite **Modele** totalement générique et réutilisable.

Modele.php

```php
<?php

require_once 'Configuration.php';

/**
 * Classe abstraite Modèle.
 * Centralise les services d'accès à une base de données.
 * Utilise l'API PDO de PHP
 *
 * @version 1.0
 * @author Baptiste Pesquet
 */
abstract class Modele {

    /** Objet PDO d'accès à la BD
        Statique donc partagé par toutes les instances des classes dérivées */
    private static $bdd;

    /**
     * Exécute une requête SQL
     *
     * @param string $sql Requête SQL
     * @param array $params Paramètres de la requête
     * @return PDOStatement Résultats de la requête
     */
    protected function executerRequete($sql, $params = null) {
        if ($params == null) {
            $resultat = self::getBdd()->query($sql);    // exécution directe
        }
        else {
            $resultat = self::getBdd()->prepare($sql); // requête préparée
            $resultat->execute($params);
        }
        return $resultat;
    }

    /**
     * Renvoie un objet de connexion à la BDD en initialisant la connexion au besoin
     *
     * @return PDO Objet PDO de connexion à la BDD
     */
    private static function getBdd() {
        if (self::$bdd === null) {
            // Récupération des paramètres de configuration BD
            $dsn = Configuration::get("dsn");
            $login = Configuration::get("login");
            $mdp = Configuration::get("mdp");
            // Création de la connexion
            self::$bdd = new PDO($dsn, $login, $mdp,
                    array(PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION));
        }
        return self::$bdd;
    }

}
```

Au passage, on transforme l'attribut **$bdd** en attribut de classe (mot-clé ***static***) afin de ne créer qu'une seule instance de la classe **PDO** partagée par toutes les classes dérivées de **Modele**. Ainsi, l'opération de connexion à la base de données ne sera réalisée qu'une seule fois.

On remarque que la syntaxe d'appel d'une méthode de classe (ici **Configuration::get**()) utilise la notation **::** comme en C++.

V-B-2 - Automatisation du routage de la requête

À présent, intéressons-nous à la partie **Contrôleur** de notre exemple. Les actions définies (affichage des billets, d'un billet, commentaire) sont spécifiques à notre contexte. En revanche, le routage d'une requête (choix de l'action à exécuter en fonction des paramètres de la requête) pourrait être rendu générique et intégré au *framework*.

Pour atteindre cet objectif complexe, nous allons commencer par ajouter une classe **Requete** dont le rôle est de modéliser une requête. Pour l'instant, le seul attribut de cette classe est un tableau rassemblant les paramètres de la requête. Par la suite, on pourrait y ajouter d'autres informations sur la requête : en-têtes HTTP, session, etc.

Requete.php
```php
<?php

class Requete {

  // paramètres de la requête
  private $parametres;

  public function __construct($parametres) {
    $this->parametres = $parametres;
  }

  // Renvoie vrai si le paramètre existe dans la requête
  public function existeParametre($nom) {
    return (isset($this->parametres[$nom]) && $this->parametres[$nom] != "");
  }

  // Renvoie la valeur du paramètre demandé
  // Lève une exception si le paramètre est introuvable
  public function getParametre($nom) {
    if ($this->existeParametre($nom)) {
      return $this->parametres[$nom];
    }
    else
      throw new Exception("Paramètre '$nom' absent de la requête");
  }
}
```

Au début du routage, un objet **Requete** sera instancié afin de stocker les paramètres de la requête reçue.

Le routage d'une requête entrante consiste à analyser cette requête afin d'en déduire le contrôleur à utiliser et l'action (méthode du contrôleur) à appeler. Ce travail est réalisé par la classe **Routeur**, dont voici la version actuelle.

Routeur.php
```php
<?php

require_once 'Controleur/ControleurAccueil.php';
require_once 'Controleur/ControleurBillet.php';

require_once 'Vue/Vue.php';

class Routeur {

  private $ctrlAccueil;
  private $ctrlBillet;

  public function __construct() {
    $this->ctrlAccueil = new ControleurAccueil();
    $this->ctrlBillet = new ControleurBillet();
  }

  // Route une requête entrante : exécute l'action associée
  public function routerRequete() {
    try {
      if (isset($_GET['action'])) {
```

Routeur.php

```
        if ($_GET['action'] == 'billet') {
          $idBillet = intval($this->getParametre($_GET, 'id'));
          if ($idBillet != 0) {
            $this->ctrlBillet->billet($idBillet);
          }
          else
            throw new Exception("Identifiant de billet non valide");
        }
        else if ($_GET['action'] == 'commenter') {
          $auteur = $this->getParametre($_POST, 'auteur');
          $contenu = $this->getParametre($_POST, 'contenu');
          $idBillet = $this->getParametre($_POST, 'id');
          $this->ctrlBillet->commenter($auteur, $contenu, $idBillet);
        }
        else
          throw new Exception("Action non valide");
      }
      else {  // aucune action définie : affichage de l'accueil
        $this->ctrlAccueil->accueil();
      }
    }
    catch (Exception $e) {
      $this->erreur($e->getMessage());
    }
  }

  // Affiche une erreur
  private function erreur($msgErreur) {
    $vue = new Vue("Erreur");
    $vue->generer(array('msgErreur' => $msgErreur));
  }

  // Recherche un paramètre dans un tableau
  private function getParametre($tableau, $nom) {
  if (isset($tableau[$nom])) {
    return $tableau[$nom];
  }
  else
    throw new Exception("Paramètre '$nom' absent");
  }
}
```

Pour l'instant, le choix de l'action s'effectue par comparaison du paramètre **action** de la requête avec différentes valeurs prédéfinies (« billet » et « commenter »). Cette opération est manuelle et devient complexe avec l'augmentation du nombre des actions possibles.

Un routage générique consisterait à déduire automatiquement le constructeur et la méthode d'action en fonction de la requête. Pour atteindre cet objectif, nous allons enrichir les URL de notre site. Jusqu'à présent, elles étaient de la forme **index.php?action=yyy&id=zzz**. Nous allons ajouter un troisième paramètre identifiant le contrôleur à utiliser. Nos URL sont maintenant de la forme **index.php?controleur=xxx&action=yyy&id=zzz**.

On peut à présent modifier en profondeur le code du routeur afin de rendre le routage automatique et donc générique.

Routeur.php

```
<?php

require_once 'Requete.php';
require_once 'Vue.php';

class Routeur {

  // Route une requête entrante : exécute l'action associée
  public function routerRequete() {
    try {
      // Fusion des paramètres GET et POST de la requête
      $requete = new Requete(array_merge($_GET, $_POST));
```

Routeur.php

```
      $controleur = $this->creerControleur($requete);
      $action = $this->creerAction($requete);

      $controleur->executerAction($action);
    }
    catch (Exception $e) {
      $this->gererErreur($e);
    }
  }

  // Crée le contrôleur approprié en fonction de la requête reçue
  private function creerControleur(Requete $requete) {
    $controleur = "Accueil";  // Contrôleur par défaut
    if ($requete->existeParametre('controleur')) {
      $controleur = $requete->getParametre('controleur');
      // Première lettre en majuscule
      $controleur = ucfirst(strtolower($controleur));
    }
    // Création du nom du fichier du contrôleur
    $classeControleur = "Controleur" . $controleur;
    $fichierControleur = "Controleur/" . $classeControleur . ".php";
    if (file_exists($fichierControleur)) {
      // Instanciation du contrôleur adapté à la requête
      require($fichierControleur);
      $controleur = new $classeControleur();
      $controleur->setRequete($requete);
      return $controleur;
    }
    else
      throw new Exception("Fichier '$fichierControleur' introuvable");
  }

  // Détermine l'action à exécuter en fonction de la requête reçue
  private function creerAction(Requete $requete) {
    $action = "index";  // Action par défaut
    if ($requete->existeParametre('action')) {
      $action = $requete->getParametre('action');
    }
    return $action;
  }

  // Gère une erreur d'exécution (exception)
  private function gererErreur(Exception $exception) {
    $vue = new Vue('erreur');
    $vue->generer(array('msgErreur' => $exception->getMessage()));
  }
}
```

La méthode principale **routerRequete**() de cette classe instancie un objet Requete en fusionnant les données des variables superglobales $_GET et $_POST, afin de pouvoir analyser toute requête issue soit d'une commande HTTP GET, soit d'une commande HTTP POST.

Ensuite, cette méthode fait appel à deux méthodes internes afin d'instancier le contrôleur approprié et d'exécuter l'action correspondant à la requête reçue.

La méthode **creerControleur**() récupère le paramètre **controleur** de la requête reçue et le concatène pour construire le nom du fichier contrôleur (celui qui contient la classe associée) et renvoyer une instance de la classe associée. En l'absence de ce paramètre, elle cherche à instancier la classe **ControleurAccueil** qui correspond au contrôleur par défaut.

La méthode **creerAction**() récupère le paramètre action de la requête reçue et le renvoie. En l'absence de ce paramètre, elle renvoie la valeur « **index** » qui correspond à l'action par défaut.

Cela n'est possible qu'en imposant à tous les contrôleurs des contraintes de nommage strictes : chaque contrôleur doit résider dans le sous-répertoire **Controleur/** sous la forme d'un fichier définissant une classe nommée **Controleur*XXX*** (***XXX*** correspondant à la valeur du paramètre **controleur** dans la requête). Le fichier doit porter le même nom que la classe.

Enfin, la méthode privée **gererErreur**() permet d'afficher la vue d'erreur.

Les plus attentifs d'entre vous auront remarqué que notre nouveau routeur fait appel aux méthodes **setRequete**() et **executerAction**() de l'objet contrôleur instancié. Il serait maladroit de dupliquer la définition de cette méthode dans tous nos contrôleurs. Nous allons donc définir une classe abstraite **Controleur** regroupant les services communs aux contrôleurs.

Controleur.php

```php
<?php

require_once 'Requete.php';
require_once 'Vue.php';

abstract class Controleur {

  // Action à réaliser
  private $action;

  // Requête entrante
  protected $requete;

  // Définit la requête entrante
  public function setRequete(Requete $requete) {
    $this->requete = $requete;
  }

  // Exécute l'action à réaliser
  public function executerAction($action) {
    if (method_exists($this, $action)) {
      $this->action = $action;
      $this->{$this->action}();
    }
    else {
      $classeControleur = get_class($this);
      throw new Exception("Action '$action' non définie dans la classe $classeControleur");
    }
  }

  // Méthode abstraite correspondant à l'action par défaut
  // Oblige les classes dérivées à implémenter cette action par défaut
  public abstract function index();

  // Génère la vue associée au contrôleur courant
  protected function genererVue($donneesVue = array()) {
    // Détermination du nom du fichier vue à partir du nom du contrôleur actuel
    $classeControleur = get_class($this);
    $controleur = str_replace("Controleur", "", $classeControleur);
    // Instanciation et génération de la vue
    $vue = new Vue($this->action, $controleur);
    $vue->generer($donneesVue);
  }
}
```

Cette classe a pour attributs l'action à réaliser et la requête. Sa méthode **executerAction**() met en œuvre le concept de **réflexion** : elle utilise les fonctions PHP **method_exists** et **get_class** afin de faire appel la méthode ayant pour nom l'action à réaliser.

La méthode **index**() est abstraite. Cela signifie que tous nos contrôleurs, qui hériteront de **Controleur**, devront obligatoirement définir une méthode **index**() qui correspond à l'action par défaut (quand le paramètre **action** n'est pas défini dans la requête).

Enfin, la méthode **genererVue**() permet d'automatiser le lien entre contrôleur et vue : les paramètres de création de la vue sont déduits du nom du contrôleur et de l'action à réaliser. Au passage, le constructeur de la classe **Vue** doit évoluer.

Vue.php
```
...
  public function __construct($action, $controleur = "") {
    // Détermination du nom du fichier vue à partir de l'action et du constructeur
    $fichier = "Vue/";
    if ($controleur != "") {
      $fichier = $fichier . $controleur . "/";
    }
    $this->fichier = $fichier . $action . ".php";
  }
...
```

Ici encore, on impose une convention de nommage aux vues :

- chaque vue doit résider dans le sous-répertoire **Vue/** ;
- dans ce répertoire, chaque vue est stockée dans un sous-répertoire portant le nom du contrôleur associé à la vue ;
- chaque fichier vue ne contient plus le terme « vue », mais porte directement le nom de l'action aboutissant à l'affichage de cette vue.

V-B-3 - Mise en place d'URL génériques

L'automatisation du routage nous a conduit à définir pour notre site des URL de la forme **monsite.fr/index.php?controleur=xxx&action=yyy&id=zzz**. Afin de faciliter leur lisibilité et leur référencement, il serait souhaitable de les remplacer par des URL de la forme **monsite.fr/controleur/action/id**. Ce format d'URL est adopté par la plupart des plates-formes PHP modernes (*frameworks*, CMS, etc.). Par exemple, l'URL pour accéder à l'action **index** sur le billet 3 sera **monsite.fr/billet/index/3**.

Cette amélioration peut être réalisée grâce au mécanisme de **réécriture d'URL** (« *URL rewriting* »). Il consiste à configurer le serveur Web pour transformer à la volée l'URL reçue.

Le serveur Web Apache dispose d'un puissant module de réécriture d'URL nommé **mod_rewrite**. Pour le mettre en œuvre, il faut ajouter à la racine de notre site un fichier de configuration nommé **.htaccess**.

.htaccess
```
# Réécrit une URL de type xxx/yyy/zzz en index.php?controleur=xxx&action=yyy&id=zzz
RewriteEngine on
RewriteRule ^([a-zA-Z]*)/?([a-zA-Z]*)?/?([a-zA-Z0-9]*)?/?$ index.php?controleur=$1&action=$2&id=$3
 [NC,L]
```

Cependant, ce nouveau format d'URL provoque des erreurs avec les liens relatifs de nos vues (inclusion de feuilles de style, d'images, de fichiers JavaScript, etc.). En effet, le navigateur résout ces liens à partir du chemin relatif défini par la nouvelle URL et non plus depuis la racine du site. Pour résoudre ce problème, il faut ajouter dans toutes nos vues la balise HTML **base**. La solution naturelle est d'inclure cette balise dans notre gabarit commun à toutes les vues.

gabarit.php
```
<!doctype html>
<html lang="fr">
  <head>
    <meta charset="UTF-8" />
    <base href="<?= $racineWeb ?>" >
    ...
```

La valeur de l'élément **base** est donnée par la variable PHP **$racineWeb** qui doit être définie au moment de la génération d'une vue. Pour cela, on modifie la méthode associée dans la classe **Vue**.

Vue.php
```
...
  public function generer($donnees) {
    // Génération de la partie spécifique de la vue
    $contenu = $this->genererFichier($this->fichier, $donnees);
    // On définit une variable locale accessible par la vue pour la racine Web
    // Il s'agit du chemin vers le site sur le serveur Web
    // Nécessaire pour les URI de type controleur/action/id
    $racineWeb = Configuration::get("racineWeb", "/");
    // Génération du gabarit commun utilisant la partie spécifique
    $vue = $this->genererFichier('Vue/gabarit.php',
      array('titre' => $this->titre, 'contenu' => $contenu,
            'racineWeb' => $racineWeb));
    // Renvoi de la vue générée au navigateur
    echo $vue;
  }
  }
...
```

Grâce au composant de configuration créé plus haut, la valeur de racine Web est récupérée depuis le fichier de configuration du site. Dans ce fichier, elle doit correspondre au chemin relatif de déploiement sur le serveur Web. Par exemple, pour un site déployé dans **monsite.fr/MonBlog**, la valeur de **racineWeb** doit être la suivante :

dev.ini
```
; Configuration pour le développement

[Installation]
racineWeb = /MonBlog/
...
```

V-B-4 - Sécurisation des données reçues et affichées

Jusqu'à présent, nous n'avons que très peu abordé les problématiques liées à la sécurité du site. Il s'agit d'un vaste sujet aux ramifications nombreuses. En matière de développement Web, la règle de sécurité essentielle est de **ne jamais faire confiance aux données reçues**. Le Web est un monde ouvert où il est facile de forger une requête HTTP contenant des paramètres intentionnellement incorrects. Deux exemples fréquents sont l'injection de code sur la page Web finale (**http://fr.wikipedia.org/wiki/Cross-site_scripting** ou XSS) et **l'injection SQL** (exécution par le SGBD de requêtes SQL non prévues). Afin de se prémunir contre de telles attaques, il est indispensable de « nettoyer » (« *sanitize* ») les données reçues ou affichées par le site.

Concernant les injections SQL visant la base de données, la solution consiste à utiliser des requêtes paramétrées plutôt que de construire la requête SQL par concaténation d'instructions et de paramètres. Dans ce cas, le moteur du SGBD effectue un nettoyage automatique des paramètres. Notre partie Modèle utilise déjà des requêtes paramétrées : nous n'avons donc pas de précaution supplémentaire à prendre à ce sujet.

La méthode de lutte contre les injections de code sur la page Web finale consiste à nettoyer (« échapper ») systématiquement toute donnée insérée dans la page. Autrement dit, nous devons ajouter à nos vues un mécanisme de nettoyage des valeurs PHP insérées. Une solution simple est d'ajouter à la classe **Vue** une méthode de nettoyage.

Vue.php
```
...
  // Nettoie une valeur insérée dans une page HTML
  private function nettoyer($valeur) {
    return htmlspecialchars($valeur, ENT_QUOTES, 'UTF-8', false);
  }
  ...
```

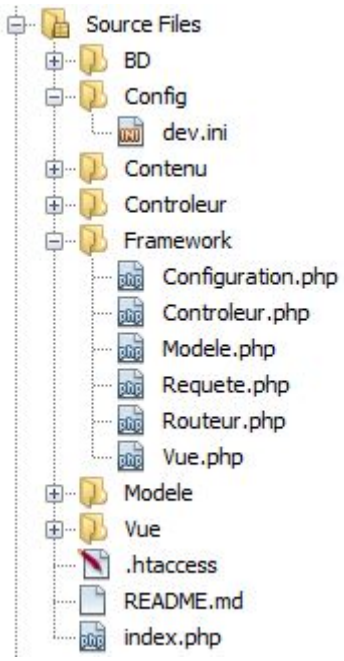
Il existe plusieurs solutions pour nettoyer une valeur avant son insertion dans une page HTML. La fonction **htmlspecialchars** utilisée ici remplace les caractères spéciaux comme « < » ou « > » par les entités HTML associées (ici, « < » deviendra « < » et « > » deviendra « > »). Ainsi, si la valeur est un script JavaScript (« <script>…</

script> »), son nettoyage empêchera l'exécution du script par le moteur JavaScript du navigateur affichant la page HTML.

Bien entendu, toutes les vues doivent faire appel à cette méthode pour chacune des données dynamiques insérées dans les pages HTML.

V-B-5 - Contraintes sur l'architecture du site

La dernière étape de la construction de notre *framework* consiste à regrouper les éléments qui le composent. Pour cela, on crée un répertoire **Framework/** dans lequel on déplace les fichiers source des six classes qui le composent.



Profitons-en pour énumérer les contraintes que le *framework* impose à l'architecture du site qui l'utilise :

- un répertoire **Config/** contient le ou les fichiers de configuration (**dev.ini** et **prod.ini**) lues par la classe **Configuration** ;
- un répertoire **Controleur/** contient tous les contrôleurs ;
- chaque classe contrôleur (ainsi que le fichier PHP associé) commence par le terme « Controleur » (exemple : **ControleurAccueil**).
- à chaque action réalisable correspond une méthode publique dans la classe contrôleur associée ;
- une méthode **index**() (action par défaut) est définie dans chaque contrôleur ;
- un répertoire **Vue/** contient toutes les vues ;
- chaque vue est définie dans un sous-répertoire de Vue/ portant le même nom que le contrôleur associé à la vue (exemple : **Vue/Accueil/index.php** pour la vue associée à l'action par défaut du contrôleur d'accueil) ;
- les fichiers **gabarit.php** (gabarit commun à toutes les vues) et **erreur.php** (affichage du message d'erreur) sont stockés dans le répertoire **Vue/** ;
- le fichier **.htaccess** et le fichier **index.php** résident à la racine du site.

V-C - Application : utilisation du framework sur le contexte d'exemple

À présent que notre *framework* est défini, nous pouvons refactoriser notre blog d'exemple pour l'utiliser. Pour commencer, voici le fichier de configuration utilisé.

dev.ini
```
; Configuration pour le développement
```

dev.ini

```
[Installation]
racineWeb = /MonBlog/

[BD]
dsn = 'mysql:host=localhost;dbname=monblog;charset=utf8'
login = root
mdp =
```

Puisque seule une méthode privée de la classe **Modele** a évolué au cours de la construction du *framework*, aucune classe de la partie **Modèle** ne doit être modifiée. La seule mise à jour consiste à actualiser les inclusions du fichier PHP contenant la classe **Modele**.

Billet.php, Commentaire.php

```
<?php

require_once 'Framework/Modele.php';
...
```

Nos deux contrôleurs (**ControleurAccueil** et **ControleurBillet**) héritent maintenant de la classe **Controleur** du *framework*. Ils doivent définir une action par défaut sous la forme d'une méthode **index**(). Chaque méthode d'action utilise la méthode **genererVue**() de **Controleur** pour générer la vue associée à l'action. Les inclusions de fichiers du *framework* doivent également être actualisées.

ControleurAccueil.php

```
<?php

require_once 'Framework/Controleur.php';
require_once 'Modele/Billet.php';

class ControleurAccueil extends Controleur {

  private $billet;

  public function __construct() {
    $this->billet = new Billet();
  }

  // Affiche la liste de tous les billets du blog
  public function index() {
    $billets = $this->billet->getBillets();
    $this->genererVue(array('billets' => $billets));
  }
}
```

La classe **ControleurBillet** utilise les services de la classe **Requete** pour accéder aux paramètres de la requête. Elle utilise également la méthode **executerAction**() de sa superclasse afin d'actualiser l'affichage des détails sur le billet.

ControleurBillet.php

```
<?php

require_once 'Framework/Controleur.php';
require_once 'Modele/Billet.php';
require_once 'Modele/Commentaire.php';

class ControleurBillet extends Controleur {

  private $billet;
  private $commentaire;

  public function __construct() {
    $this->billet = new Billet();
    $this->commentaire = new Commentaire();
  }
```

ControleurBillet.php
```
  // Affiche les détails sur un billet
  public function index() {
    $idBillet = $this->requete->getParametre("id");

    $billet = $this->billet->getBillet($idBillet);
    $commentaires = $this->commentaire->getCommentaires($idBillet);

    $this->genererVue(array('billet' => $billet,
      'commentaires' => $commentaires));
  }

  // Ajoute un commentaire sur un billet
  public function commenter() {
    $idBillet = $this->requete->getParametre("id");
    $auteur = $this->requete->getParametre("auteur");
    $contenu = $this->requete->getParametre("contenu");

    $this->commentaire->ajouterCommentaire($auteur, $contenu, $idBillet);

    // Exécution de l'action par défaut pour actualiser la liste des billets
    $this->executerAction("index");
  }
}
```

Enfin, on déplace les fichiers **vueAccueil.php** et **vueBillet.php** dans les sous-répertoires **Vue/Accueil** et **Vue/Billet** respectivement, puis on les renomme tous les deux **index.php** puisque ces vues sont associées à l'action par défaut de chacun des deux contrôleurs. Le fichier **vueErreur.php** est renommée **erreur.php**. Le contenu des vues n'est pas modifié, à l'exception notable des appels systématiques à la méthode **nettoyer**() pour chaque donnée incluse dans la vue. Voici par exemple un extrait de la vue d'affichage des détails d'un billet.

Vue/Billet/index.php
```
<?php $this->titre = "Mon Blog - " . $this->nettoyer($billet['titre']); ?>

<article>
    <header>
        <h1 class="titreBillet"><?= $this->nettoyer($billet['titre']) ?></h1>
        <time><?= $this->nettoyer($billet['date']) ?></time>
    </header>
    <p><?= $this->nettoyer($billet['contenu']) ?></p>
...
```

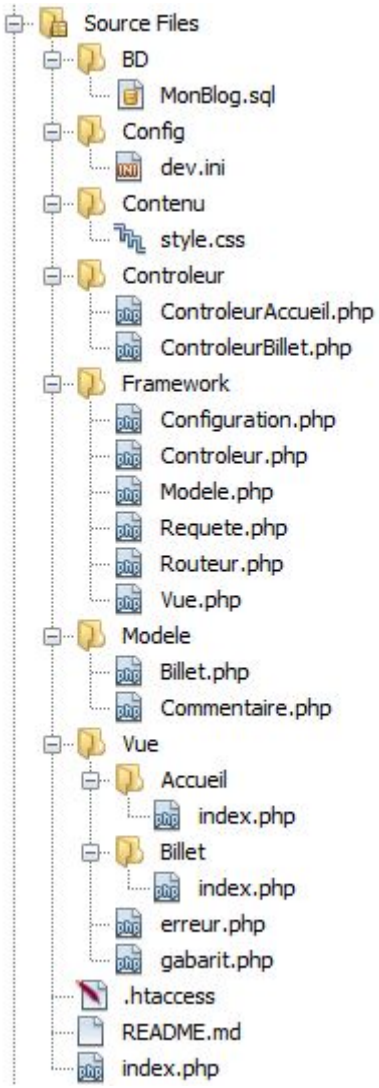
Vous trouverez les fichiers source du contexte final à l'adresse **https://github.com/bpesquet/MonBlog/**.

VI - Conclusion et perspectives

VI-A - Bilan final

Nous arrivons au terme de notre chantier de *refactoring*. Notre blog d'exemple utilise un *framework* MVC orienté objet qui lui fournit plusieurs services essentiels : configuration, interactions avec la base de données, routage des requêtes, URL génériques, sécurisation des données.

L'image ci-dessous illustre l'architecture finale de notre exemple.



Le *framework* que nous avons bâti est totalement générique et réutilisable dans un autre contexte : il suffit de récupérer ses fichiers source (le contenu de **Framework/** ainsi que les fichiers **.htaccess** et **index.php** de la racine), puis de respecter les contraintes qu'il impose à l'architecture du site (voir plus haut).

VI-B - Pour aller encore plus loin

Afin de limiter sa complexité, notre *framework* reste minimaliste et donc largement perfectible. Les choix effectués au cours de sa création sont discutables et il est possible d'arriver à un résultat similaire par d'autres voies. Voici quelques suggestions d'améliorations qui seront autant d'exercices enrichissants :

- mettre en place des **espaces de noms** (*namespaces*) ;
- utiliser la fonctionnalité PHP d'**autochargement de classes** (« *autoloading* ») ;
- ajouter des mécanismes de validation des données entrantes (exemple : si j'attends un entier, je vérifie que je reçois bien un entier) ;
- intégrer un mécanisme de journalisation des événements et des erreurs dans des fichiers journaux (*log files*) ;
- utiliser un moteur de *templates* comme **Twig** pour les vues ;
- intégrer des composants existants qui ont fait leurs preuves, par exemple l'excellent **HttpFoundation** issu du *framework* Symfony.

Ainsi se termine cet article qui vous aura, j'espère, apporté une meilleure compréhension des possibilités d'application en PHP du modèle MVC.

VI-C - Liens utiles

Voici les principales sources utilisées pendant l'écriture de cet article :

- **Symfony2 versus flat PHP** ;
- **Custom PHP MVC Tutorial** ;
- **An Introduction to the Front Controller Pattern**.

Quelques articles complémentaires pour aller plus loin :

- **PHP tips & tricks** ;
- **Create your own framework... on top of the Symfony2 Components**.

Enfin, une sélection de quelques projets instructifs :

- **http://mvcmusicstore.codeplex.com/** (une introduction à ASP.NET MVC) ;
- **https://github.com/ndavison/Nathan-MVC** ;
- **https://github.com/fguillot/simpleFramework** ;
- **https://github.com/lstrojny/EPHPMVC**.

VII - Remerciements

Nous tenons à remercier **Benjamin Delespierre** et **Frédéric Guillot** pour leurs conseils avisés, ainsi que **ClaudeLELOUP** pour sa relecture attentive.