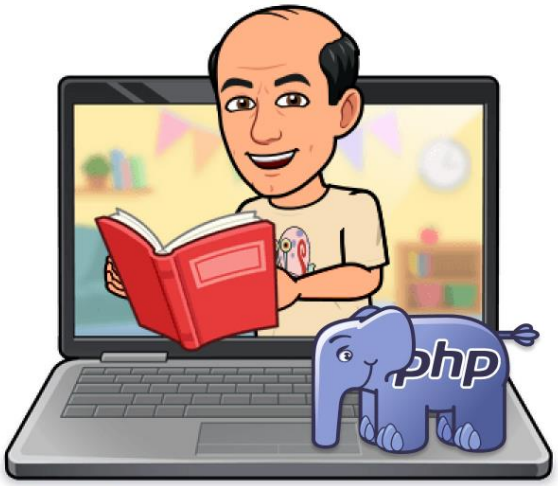
L’architecture MVC en php

Model
Vue
Controller



Dans cette partie du cours nous allons progressivement passer d'un projet simple procédural vers une architecture.

Tutoriel MVC Réalisation :	Germain SIPIERE Formateur AFPA Guillaume DELACROIX Formateur AFPA 15 décembre 2022	Afpa

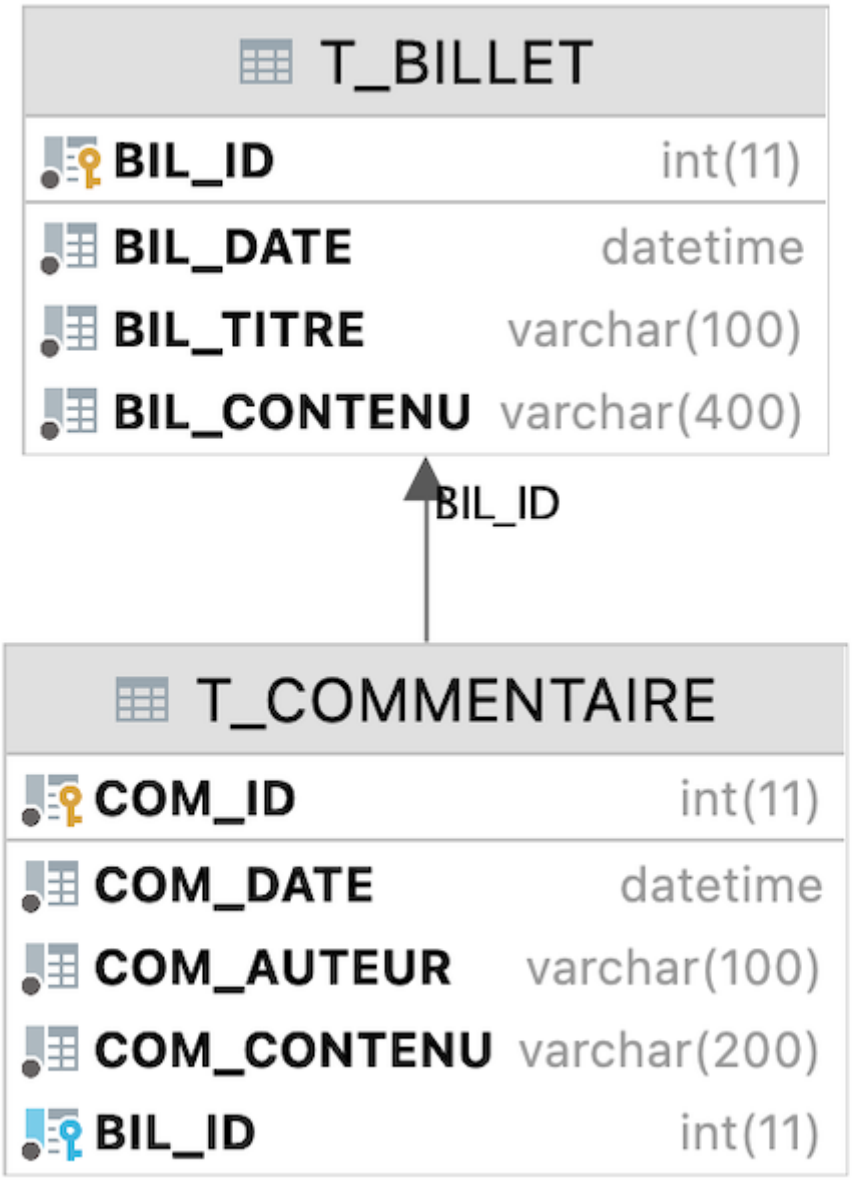
Table des matières

En route vers le MVC

Nous allons réaliser ici un mini blog à titre d'exemple. Il ne sera pas très fonctionnel mais l'objectif est de comprendre l'architecture MVC.

Mise en place du blog en version procédurale

Pour la persistance, nous allons utiliser la base de données suivante:



Vous pouvez utiliser le script fourni pour créer et alimenter votre base.

Affichons simplement la liste des billets dans une page d'accueil :

```
// index.php

<!doctype html>
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <meta name="viewport"
          content="width=device-width, user-scalable=no, initial-scale=1.0,
maximum-scale=1.0, minimum-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <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>Hello et bienvenue !!!!</p>
    </header>
    <div id="contenu">
        <?php
        $bdd = new PDO("mysql:host=database:3306;dbname=boggy;charset=utf8",'VOTRE
USER','VOTRE MDP');
        $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>
    <footer id="piedBlog">
        Blog exercice
    </footer>
</div>
</body>
</html>
```

Et voici la feuille de style qui va avec

```
/* 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;
}

a:link {
    text-decoration: none;
}

#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;
}
```

Et si tout fonctionne vous avez ça:

Les défauts

`index.php` contient du php et du html. De plus son code parait difficilement réutilisable et surtout va devenir difficile à maintenir au fil du temps et des évolutions de l'application.
En fait, en général, une application va devoir gérer plusieurs problématiques :

- la **présentation** : interaction avec l'extérieur.
- le **traitement** : tout ce qui est calculs en lien avec les règles métier.
- les **données** : tout ce qui est accès et manipulations des données.

Nous allons donc être obligé de ranger tout ça.

Première approche du MVC

Séparation de l'affichage

La première chose à faire va être de séparer l'affichage du reste de notre code:

```php
<?php
// index.php
// accès aux données
$bdd = new PDO("mysql:host=database:3306;dbname=boggy;charset=utf8",'VOTRE
USER','VOTRE MDP');
$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';
```

et

```php
<!--vueAccueil.php-->
<!doctype html>
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <meta name="viewport"
          content="width=device-width, user-scalable=no, initial-scale=1.0,
maximum-scale=1.0, minimum-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <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>Hello et bienvenue !!!!</p>
    </header>
    <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>
    <footer id="piedBlog">
```

```
        Blog exercice
    </footer>
</div>
</body>
</html>
```

Séparation de l'accès aux données

Isolons de la même manière l'accès aux données en créant un fichier `Model.php`

```
<?php
function getBillets(){
    $bdd = new PDO("mysql:host=database:3306;dbname=boggy;charset=utf8",'VOTRE
USER','VOTRE MDP');
    $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;
}
```

De ce fait, `index.php` devient:

```
<?php
require 'Model.php';

// accès aux données
$billets = getBillets();

// affichage
require 'vueAccueil.php';
```

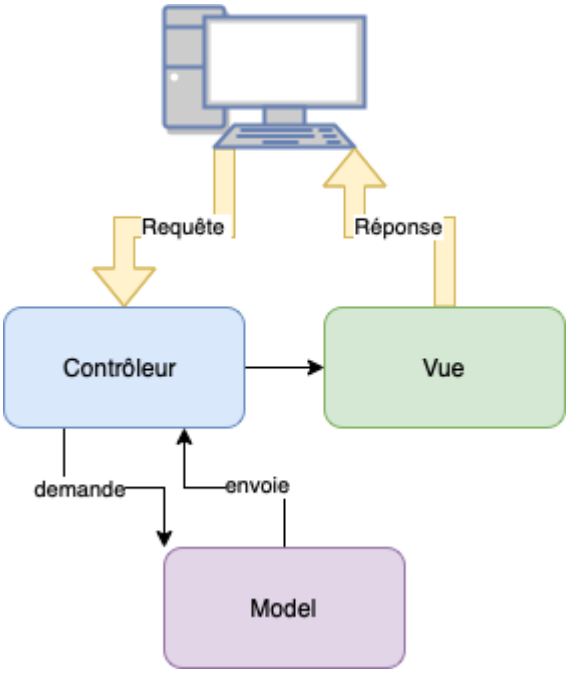
Nous avons maintenant 3 fichier pour notre page:

- `Model.php` pour l'accès aux données.
- `vueAccueil.php` pour l'affichage.
- `index.php` qui fait le lien entre les deux autres.

Nous pouvons alors commencer à parler de MVC.

Le MVC

M pour model **V** pour vue et **C** pour contrôleur. Le principe est le suivant:



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.

Quelques améliorations supplémentaires

Un site web se réduit rarement à une seule page. Nous allons donc mettre en place l'utilisation d'un modèle de page (Template). Ce modèle contiendra les éléments communs de nos vues.

```
<!--gabarit.php -->
<!doctype html>
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <meta name="viewport"
```

```
          content="width=device-width, user-scalable=no, initial-scale=1.0,
maximum-scale=1.0, minimum-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <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>Hello et bienvenue !!!!</p>
    </header>
    <div id="contenu">
        <?= $contenu; ?>
    </div>
    <footer id="piedBlog">
        Blog exercice
    </footer>
</div>
</body>
</html>
```

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

```
<?php $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'; ?>
```

La méthode `ob_start` est une méthode de mise en tampon. Ce tampon est récupéré dans `$contenu` après la fin de la boucle grâce à `ob_get_clean()`. LE rendu est alors déclenché avec l'appel du gabarit qui reprend les valeurs de `$titre` et de `$contenu`

On va aussi factoriser la connexion à la base de données. Pour le moment, un seul appel est fait à la base. Si nous faisons évoluer l'application, il y aura d'autres appelles, c'est pourquoi il faut factoriser la phase de connexion.

```
<!--Model.php-->
<?php
```

```php
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;
}

function getBdd()
{
    $bdd = new PDO("mysql:host=database:3306;dbname=boggy;charset=utf8",'VOTRE
USER','VOTRE MDP',array(PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION));

    return $bdd;
}
```

Gestion des erreurs

C'est le **contrôleur** qui va s'occuper de cette problématique.

J'avais déjà modifié le `PDO` pour qu'il renvoie des exceptions avec le paramètre supplémentaire: `array(PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION)`. Nous pouvons donc récupérer cette exception dans `index.php`

```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.

```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.

```php
<?php

require 'Modele.php';

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

Bilan intermédiaire

Où en sommes-nous?

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.

Vous devriez avoir ces fichiers :

```
.
├── Model.php
├── gabarit.php
├── index.php
├── style.css
├── vueAccueil.php
└── vueErreur.php
```

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.

Affichage d'un Billet avec l'architecture mise en place

Suivons les règles précédentes:

- Ajoutons une fonction dans `Model.php` pour accéder aux données dont nous avons besoin.

```php
// ...

// récupère un billet avec son id
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();
    }
    else{
        throw new Exception("Aucun billet ne correspond à cet identifiant");
    }
}
// récupère les commentaires associés à un billet

//Attention ! : si vous n'êtes pas sous PHP 8.0, ne pas écrire "function
getComments($idBillet): bool|PDOStatement"
//mais simplement "function getComments($idBillet)"

function getComments($idBillet): bool|PDOStatement
{
    $bdd = getBdd();
    $comments = $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 =?');
    $comments->execute(array($idBillet));
    return $comments;
}
```

- Créons ensuite la vue `vueBillet.php` dédiée aux informations du billet

```
<?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'; ?>
```

cette vue ne définit que ses éléments particuliers, le reste est géré par le gabarit (*dernière ligne*).

- Enfin, il nous faut un contrôleur `billet.php` qui fera le lien entre le model et la vue

```
<?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';
}
```

Afin d'atteindre cette nouvelle fonctionnalité, nous devons modifier `vueAccueil.php` en ajoutant un lien qui nous mène au contrôleur

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

Ajoutez également une classe au css

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

Si tout s'est bien passé, vous devriez avoir ce résultat pour un billet


Mon BLog
Hello et bienvenue !!!!
Premier billet
2022-03-30 07:46:17
Bonjour monde ! Ceci est le premier billet sur mon blog.
Réponses à Premier billet
A. Nonyme dit :
Bravo pour ce début
Moi dit :
Merci ! Je vais continuer sur ma lancée
Blog exercice pour mettre en évidence les modifications pour obtenir un MVC

Amélioration avec un contrôleur frontal

Point d'étape

Pour le moment nous avons la structure suivante:

```
mvcevo
 ├ Model.php
 ├ billet.php
 ├ gabarit.php
 ├ index.php
 ├ style.css
 ├ vueAccueil.php
 ├ vueBillet.php
 └ vueErreur.php
```

- `Model.php` pour l'accès aux données.
- `vueAccueil.php`, `vueBillet.php`, et `vueErreur.php` s'occupent de la vue.
- `index.php` et `billet.php` correspondent à la partie Contrôleur.

Un contrôleur frontal

Nous avons une architecture à n contrôleurs, ce n'est pas idéal.

- cela expose le nom des fichiers php.
- la maintenabilité du site va vite devenir difficile.

Pour résoudre ce problème, nous allons mettre en place un **contrôleur frontal**.
L'objectif de ce contrôleur est de centraliser les requetes entrantes puis d'utiliser d'autres contrôleurs pour réaliser l'action et de renvoyer le résultat sous la forme d'une vue.

En principe on utilise `index.php` comme contrôleur frontal. C'est cet `index.php` qui va recevoir les différentes actions à effectuer. Contrairement à ce que nous avons pour le moment. (`index.php` pour afficher les billets et `billet.php?id=<id du billet>` pour afficher un billet en particulier).
Commençons à mettre en place deux actions:

- si action vaut "billet", le contrôleur affichera un billet.
- si action n'est pas valorisé, le contrôleur affichera tous les billets.

```
// Controleur.php
<?php
```

```php
require 'Model.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);
    $commentaires = getComments($idBillet);
    require 'vueBillet.php';
}

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

Il faut aussi modifier `index.php`

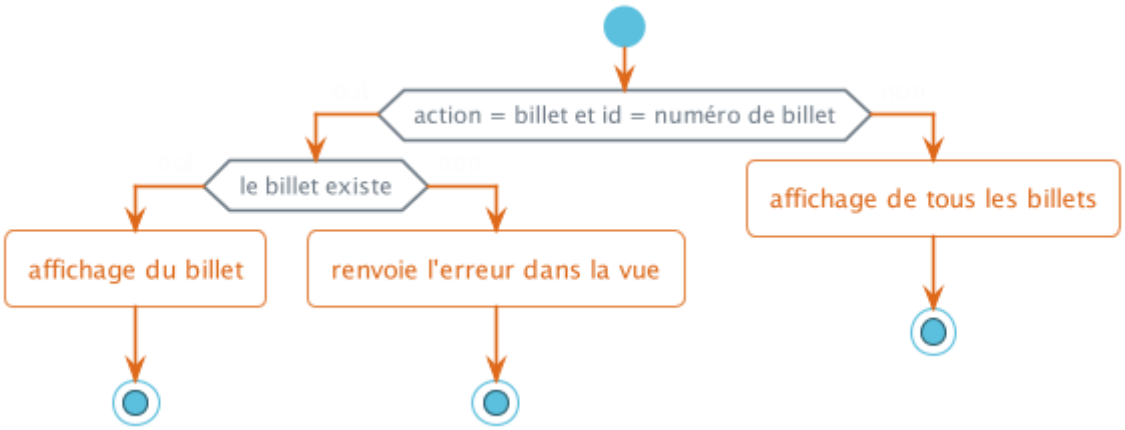
```php
<?php

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("id billet non valide");
            }else
                throw new Exception("id billet non défini");
        }else
            throw new Exception("action non valide");
    }else {
        accueil();
    }

}catch(Exception $e) {
    erreur($e->getMessage());
}
```

Vous pouvez supprimer `billet.php` il est devenu inutile

Il faut aussi modifier les liens dans la vue Accueil (`vueAccueil.php`)

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

En résumé, notre contrôleur suit la logique suivante



Finissons par un peu de rangement

Il faut toujours penser à structurer vos sources. D'une part pour vous, mais aussi pour d'autres développeurs qui pourraient travailler sur le projet.

Je vous propose la structure suivante:

```
mvcevo
├ BD
│ └ blog.sql
├ assets
│ └ style.css
├ controller
│ └ Controleur.php
├ model
│ └ Model.php
├ vue
│ ├ gabarit.php
│ ├ vueAccueil.php
│ ├ vueBillet.php
│ └ vueErreur.php
└ index.php
```

Après ce ménage, faites bien attention à verifier et corriger les chemins.

Un peu d'objet avant d'aller plus loin

La Programmation Orientée Objet (POO) est essentielle aujourd'hui en développement et se retrouve dans la majorité des langages. Php ne fait pas exception et nous allons voir comment en profiter.

Les objets

Un objet peut être comparé à une boite où nous pourrons stocker des variables (*propriétés*) et des fonctions (*méthodes*).

Vous avez surement déjà utilisé des objets sans vous en rendre compte.
Le `DateTime` est un type objet. Vous pouvez stocker une date et utiliser les fonctions associées à cette date.

```php
<?php
$ce_jour = new DateTime();
echo 'Nous sommes le ';
echo $ce_jour->format('d-m-Y');
```

Mais nous pouvons également créer nos propre types. Pour créer un type d'objet il faut créer une **classe**. Cette classe est comme un moule d'où on va extraire les objets que l'on appelle instance de la classe.

Définissons par exemple un type `Produit` qui aura un **nom**, une **quantité**, un **prix** et un booléen **rupture** pour savoir s'il est en stock ou non. Ces valeurs sont appelées **propriétés** ou **membres**.

```php
<?php
class Produit
{
public $nom;
public $quantite ;
public $prix;
public $rupture ;

function afficheProduit(){
    return "Affichage du produit";
}

function ajouterProduit(){
    return "Ajouter du produit";
}

function supprimerProduit(){
    return "Supprimer du produit";
}
}
>
```

Utilisons cette classe dans un fichier php:

```php
<?php
// ICI déclarez la classe

$mobile = new Produit; //Instanciation

$mobile->nom = "iphone";
$mobile->quantite = "10";
$mobile->prix = 900;
$mobile->rupture = false;

echo $mobile->afficheProduit();
>
```

Le `$this`

Il est souvent utile d'accéder aux propriétés d'une instance de la classe. Malheureusement, nous ne pouvons pas connaitre d'avance le nom de cette instance, c'est pourquoi nous avons `$this` à notre disposition:

```php
<?php

class Produit
{
private $nom;
public $quantite ;
public $prix;
public $rupture ;

function afficheProduit(){
    return
    "Nom: ".$this->nom."<br/>".
    "Quantité: ".$this->quantite."<br/>".
    "Prix: ".$this->prix."<br/>".
    (($this->rupture)?"Rupture de stock":"En stock");
}

function ajouterProduit(){
    $this->quantite++;
    if ($this->quantite > 0) $this->rupture = false;
}

function supprimerProduit(){
    $this->quantite--;
    if ($this->quantite <= 0) {
        $this->rupture = true;
        $this->quantite = 0;
    }
}
```

L'encapsulation

En programmation objet on ne doit pas laisser accéder directement aux propriétés. Elles ne doivent être accessibles que de l'intérieur de l'instance de l'objet.

Nous pouvons déclarer les propriétés comme privés pour en interdire l'accès.

Notre classe devient

```php
<?php

class Produit
{
private $nom;
private $quantite ;
private $prix;
private $rupture ;

public function ajouterProduit(){
    $this->quantite++;
    if ($this->quantite > 0) $this->rupture = false;
}

public function __toString()
{
    return
    "Nom: ".$this->nom."<br/>".
    "Quantité: ".$this->quantite."<br/>".
    "Prix: ".$this->prix."<br/>".
    (($this->rupture)?"Rupture de stock":"En stock");
}

public function supprimerProduit(int $nb = 1){
    $this->quantite -= $nb;
    if ($this->quantite <= 0) {
        $this->rupture = true;
        $this->quantite = 0;
    }
}

}
```

Cependant, si nous devons modifier une propriété il va falloir ouvrir un accès. Ceci se fait à l'aide des assesseurs (getter et setter).

Par exemple pour la propriété nom, les assesseurs seront:

```php
/**
 * Get the value of nom
 */
public function getNom()
{
return $this->nom;
}

/**
 * Set the value of nom
 *
 * @return  self
 */
public function setNom($nom)
{
$this->nom = $nom;
```

```
return $this;
}
```

Nous ne sommes pas obligés de créer tous les assesseurs, seulement ceux dont nous avons besoin.

Ajoutez les assesseurs manquant à notre classe et pensez bien à corriger dans `index.php` les accès aux propriétés.

```
$mobile = new Produit; //Instanciation

$mobile->setNom("iphone");
$mobile->setQuantite(10);
$mobile->setPrix(900);
$mobile->setRupture(false);

echo $mobile->afficheProduit();
```

Les méthodes magiques

Il existe des fonctions ou méthodes en php que l'on appelle "magiques" car elles peuvent s'exécuter lors de certains évènements. (cf manuel PHP)

La méthode `__toString()`

La méthode `__toString()` transforme notre objet en chaine de caractère directement. Modifions notre classe pour en profiter.

```
class Produit
{
private $nom;
private $quantite ;
private $prix;
private $rupture ;

public function __toString()
{
    return
    "Nom: ".$this->nom."<br/>".
    "Quantité: ".$this->quantite."<br/>".
    "Prix: ".$this->prix."<br/>".
    (($this->rupture)?"Rupture de stock":"En stock");
}

public function ajouterProduit(){
    $this->quantite++;
    if ($this->quantite > 0) $this->rupture = false;
}

public function supprimerProduit(int $nb = 1){
    $this->quantite -= $nb;
    if ($this->quantite <= 0) {
        $this->rupture = true;
        $this->quantite = 0;
    }
```

```
}

// les Assesseurs ...
}
```

Cette fois si on l'utilise dans un fichier, cela donne:

```
$mobile = new Produit; //Instanciation

$mobile->setNom("iphone");
$mobile->setQuantite(10);
$mobile->setPrix(900);
$mobile->setRupture(false);

echo $mobile;

$mobile->supprimerProduit(10);

echo "<br/><hr>";

echo $mobile;
```

La méthode `__construct`

Quand nous créons l'instance de l'objet il serait plus sympa de pouvoir alimenter directement les propriétés. Le constructeur est là pour ça. Il s'agit de la méthode `__construct`.

```
 public function __construct($nom, $prix, $quantite=0)
    {
        $this->nom = $nom;
        $this->quantite = $quantite;
        $this->prix = $prix;
        $this->rupture = $quantite>=0;
    }
```

Et le même exemple que tout à l'heure devient

```
$mobile = new Produit("iphone",900,10); //Instanciation

$imprimante = new Produit("hp",300);

echo $mobile;
echo "<br/><hr>";
echo $imprimante;

$mobile->supprimerProduit(10);

echo "<br/><hr>";

echo $mobile;
```

Vous pouvez remarquer que j'ai donné la possibilité d'instancier également sans remplir la quantité grâce aux paramètres optionnels.

L'héritage

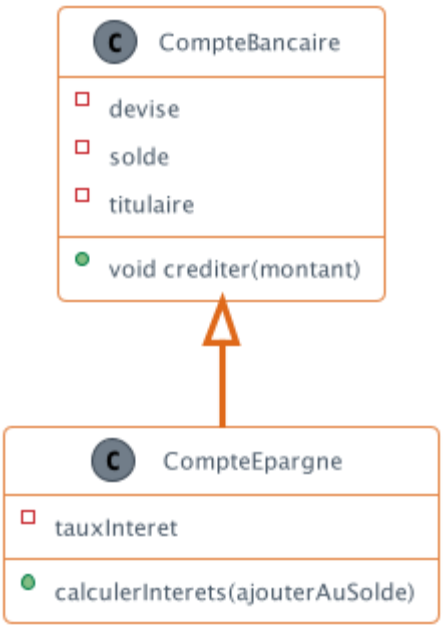
Nous pouvons également trouver des similarités d'une classe à une autre. L'idée de l'héritage est de réutiliser le code ou les fonctionnalités communes en regroupant les classe par familles.

Prenons l'exemple du compte bancaire et du compte épargne.

```php
// ConpteBancaire.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;
    }
}
```

```php
// CompteEpargne.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 . '%.';
    }
}
```

L'uml correspondant:



Et un exemple d'utilisation de ces deux classes:

```php
// poo.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 />';
```

En executant ces fichiers vous obtenez:

```
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%.
```

Passage à la POO

D'abord le Model

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 dans la classe `Model.php`.

```php
<?php
class Model {
    /**
     * @throws Exception
     */
    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();
        }
        else{
            throw new Exception("Aucun billet ne correspond à cet identifiant");
        }
    }

    public function getBillets(): bool|PDOStatement
    {
        $bdd = $this->getBdd();
        return $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');
    }

    private function getBdd(): PDO
    {
        return new PDO("mysql:host=database:3306;dbname=boggy;charset=utf8",
'root', 'tiger',array(PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION));
    }

    public function getComments($idBillet): bool|PDOStatement
    {
        $bdd = $this->getBdd();
        $comments = $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 =?');
        $comments->execute(array($idBillet));
        return $comments;
    }
}
```

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.php`, en charge de l'accès aux données des billets, comme ceci.

```php
//model/Billet.php
<?php
require_once 'model/Model.php';

class Billet extends Model
{
    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);
    }

    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 $billet->fetch();
        else
            throw new Exception("Aucun billet ne correspond à l'identifiant
'$idBillet'");

    }

}
```

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

```php
<?php
abstract class Model {
    private $bdd;

    private function getBdd(): PDO
    {
        if ($this->bdd == null){
            $this->bdd = new
PDO("mysql:host=database:3306;dbname=boggy;charset=utf8", 'VOTRE USER','VOTRE
MDP',array(PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION));
        }
```

```
        return $this->bdd;
    }

    protected function executerRequete($sql, $params = null): bool|PDOStatement
    {
        if ($params == null) {
            $resultat = $this->getBdd()->query($sql);    // exécution directe
        }
        else {
            $resultat = $this->getBdd()->prepare($sql);  // requête préparée
            $resultat->execute($params);
        }
        return $resultat;
    }
}
```

Une classe abstraite ne peut pas être instanciée

Sur le même principe, nous pouvons écrire la classe `Commentaire.php`

```
<?php
require_once 'model/Model.php';
class Commentaire extends Model
{
    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 = ?';
        return $this->executerRequete($sql,array($idBillet));
    }
}
```

Passons aux vues

Créons d'abord une classe `Vue.php` qui va assurer la génération des vues

```
<?php
//Vue/Vue.php
class Vue
{
private $fichier;

private $titre;

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

// Génération et affichage de la vue
public function generer($donnees){
    $contenu = $this->genererFichier($this->fichier, $donnees);

    $vue = $this->genererFichier('Vue/gabarit.php',array('titre' => $this-
>titre,'contenu' => $contenu));

    echo $vue;
}
```

```
    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 $fichier;
            // Arrêt de la temporisation et renvoi du tampon de sortie
            return ob_get_clean();
        }
        else {
            throw new Exception("Fichier '$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 `extract()` 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.php`. 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()`.

Et le contrôleur

Notre partie Contrôleur actuelle se compose d'une série d'actions écrites sous la forme de fonctions et du contrôleur frontal `index.php`.
Toute évolution du site Web va faire augmenter le nombre d'actions possibles, jusqu'à rendre les fichiers Controleur.php et index.php difficiles à lire et à maintenir.

Une solution plus modulaire consiste à répartir les actions dans plusieurs classes contrôleur, en fonction du contexte associé aux actions. Ici, nous pourrions créer une classe ControleurAccueil pour gérer l'accueil et une classe 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
<?php

require_once './model/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 './model/Billet.php';
require_once './model/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.

Router.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");
```

```
      }
      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.

Bilan

Voilà ce que vous devriez avoir maintenant:

```
mvcevo
├ BD
│ └ blog.sql
├ assets
│ └ style.css
├ controleur
│ ├ ControleurAccueil.php
│ ├ ControleurBillet.php
│ └ Routeur.php
├ model
│ ├ Billet.php
│ ├ Commentaire.php
│ └ Model.php
├ vue
│ ├ Vue.php
│ ├ gabarit.php
│ ├ vueAccueil.php
│ ├ vueBillet.php
│ └ vueErreur.php
├ boggy.png
└ index.php
```

Nous avons fortement complexifié la structure du projet. À ce prix, nous possédons maintenant un projet plus robuste et surtout plus facilement maintenable.

Dorénavant pour ajouter une fonctionnalité il faudra:

1. ajouter ou enrichir la classe modèle associée ;
2. ajouter ou enrichir une vue utilisant le gabarit pour afficher les données ;
3. ajouter ou enrichir une classe contrôleur pour lier le modèle et la vue.

Un dernier exemple

Ajoutons une fonctionnalité d'ajout de commentaire:

- Dans l'affichage d'un billet on doit pouvoir ajouter un commentaire
- Les champs auteur et commentaire sont obligatoires
- La validation du commentaire déclenche son enregistrement et l'actualisation de la page

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

```
// 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(?, ?, ?, ?)';
        $date = date("Y-m-d H:i:s");  // Récupère la date courante
        $this->executerRequete($sql, array($date, $auteur, $contenu, $idBillet));
    }
```

2. 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>
```

et on ajoute un peu de css

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

3. 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");
  }
...
```

Et on met à jour la méthode de routage d'une 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'])) {
            $idBillet = intval($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");
  }
}
```