Haute École Bruxelles-Brabant
École Supérieure d'Informatique
Bachelor en Informatique

2020 - 2021
WEBR4-DIN

Développement Internet

Mise en pratique MVC : construction d'une première application Web

Durant cette séance, nous allons utiliser les concepts d'architecture MVC en PHP afin de construire progressivement une application Web.

Ce chapitre s'inspire de "Évoluer vers une architecture MVC en PHP" de Baptiste Pesquet, par l'intermédiaire de M. Codutti et N. Richard.

1 Point de départ

Nous allons travailler sur une page affichant des billets de blogs, décrite par le code PHP suivant :

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


Version du document : 2 avril 2021

```
            </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>
```

Ce code affiche la page d'accueil d'un blog. Elle récupère les informations des différents billets de blog dans une base de données via la librairie PDO.

Un tel code peut faire affaire sur un projet de petite taille, mais si nous espérons pouvoir faire grandir l'application en complexité, une telle architecture ne fera pas l'affaire. Nous voudrions donc faire évoluer le code de cette application afin d'implémenter une architecture MVC. Il faudra pour cela atteindre une séparation du code en modules à responsabilité unique : un modèle, une vue, et un contrôleur.

2 Modification 1 : Séparer les modules

La page `index.php` effectue deux actions : la récupération des billets de blog dans la base de données, qui fait partie de la **logique métier** du système, et la génération de code HTML en vue de l'affichage. Nous pouvons séparer ces deux composants : un fichier `Model.php` en charge de la logique métier :

```
<?php
$db = new PDO('mysql:host=localhost;dbname=monblog;charset=utf8',
'root', '');
$billets = $db->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');
```

Et un fichier `vueAccueil.php` en charge de l'affichage :

```
<!doctype html>
<html lang="fr">
  <head>
    <meta charset="UTF-8" />
    <link rel="stylesheet" href="style.css" />
    <title>Mon Blog</title>
  </head>
  <body>
    <div id="global">
      <header>
        <a href="index.php"><h1 id="titreBlog">Mon Blog</h1></a>
```

```
        <p>Je vous souhaite la bienvenue sur ce modeste blog.</p>
      </header>
      <div id="contenu">
        <?php 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>
```

La page `index.php` invoque ces autres fichiers PHP pour générer la page :

```
<?php
//Récupération des variables du modèle
require "Model.php";

//Affichage
require "vueAccueil.php";
```

3 Modification 2 : Isolation de l'accès aux données

Le module `Model.php` dispose de peu de modularité : tout fichier PHP qui le charge va automatiquement lancer une requête pour récupérer tous les billets de blog dans la base de données. Si l'application grandit, de nombreuses autres requêtes pourraient être définies, et il sera peu désirable de les effectuer toutes sur chaque page. Par conséquent, il vaut mieux placer notre requête dans une fonction :

```
<?php
function getAllPosts() {
    $db = new PDO('mysql:host=localhost;dbname=monblog;charset=utf8',
    'root', '');
    $billets = $db->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;
}
```

Et adapter `index.php` en conséquence :

```php
<?php
//Récupération des variables du modèle
require "Model.php";
$billets = getAllPosts();

//Affichage
require "vueAccueil.php";
```

4 Modification 3 : Utilisation d'un gabarit

L'affichage tel qu'il est défini ici est exclusivement utilisable par la page d'accueil. Si on souhaite ajouter d'autres pages au blog, et conserver une structure commune sur chaque page, il sera utile de se servir d'un gabarit.

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

Il s'agit simplement de la vue précédemment définie, où toutes les informations spécifique à la page d'accueil sont remplacées par des variables.

Notre fichier `vueAccueil.php` n'aura plus pour rôle que de récupérer les données du modèle, et de les placer dans des variables que notre garabit s'attend à recevoir.

```
<?php $title = "Mon Blog"; ?>
<?php ob_start();?>
<?php foreach ($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 $content = ob_get_clean(); ?>
<?php require "template.php"; ?>
```

5 Modification 4 : Factoristion du modèle

Notre modèle possède pour l'instant deux rôles : se connecter à la base de données, et récupérer les billets de blog. Cette première fonctionnalité est commune à tous les accès à la base de données ; il sera donc approprié de le factorisé.

Pour cela, nous ferons la transition vers une classe abstraite `Model` en charge de la connexion, et d'une classe `Post` en charge des opérations spécifiques aux billets de blog.

La classe Model :

```
<?php
abstract class Model {
    protected function executeRequest($sql, $params=null) {
        $db = new PDO('mysql:host=localhost;dbname=monblog;charset=utf8',
        'root', '');
        if ($params == null) {              // exécution directe
            $result = $db->query($sql);
        } else {                            // requête préparée
            $result = $db->prepare($sql);
            $result->execute($params);
        }
        return $result;

    }
}
```

La classe Post :

```php
<?php
require "Model.php";
class Post extends Model {
    function getAllPosts() {
        $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';
        $this->executeRequest($sql);
    }
}
```

Cette classe pourrait aussi, à terme, posséder d'autres fonctions, par exemple

- `Post::getPost($id)` pour récupérer un seul billet avec son identifiant dans la base de données
- `Post::addPost(/* params */)` pour créer un nouveau billet
- etc.

6 Modification 5 : Gestion des erreurs

Pour l'instant, aucune erreur de connexion avec la base de données n'est prise en charge. Il faudra remédier à cela au plus vite. On peut faire cette gestion d'erreur dans notre contrôleur, qui est pour l'instant `index.php` :

```php
<?php
try {
    require "Post.php";
    $post = new Post();
    $billets = $post->getAllPosts();
    require "vueAccueil.php";
} catch (PDOException $e) {
    require "ViewError.php";
}
```

Avec une vue d'erreur simple (mais qui pourra bien sûre être remise en forme plus tard !)

```
<html><body>Erreur ! ' . <?= $e->getMessage() ?> . '</body></html>
```

7 Modification 6 : Contrôleur unique

Comme nous venons de le dire, `index.php` est pour l'instant notre seul contrôleur. Toute page qu'on souhaite ajouter nécessitera d'écrire un nouveau fichier de contrôleur, ce qui n'est pas faisable dans le cadre d'une mise à l'échelle.

À la place, nous allons conserver textttindex.php comme point d'accès unique à notre site, mais se servir d'un Router et récupérer des actions en paramètre de la méthode GET afin de déterminer la page exacte à afficher. Par exemple :

- `index.php?action=allPosts` pour tous les billets
- `index.php?action=Post&id=2` pour affichet le billet d'identifiant 2 dans la base de données, etc.

Nous définirons donc des actions de contrôleur, `ControllerActions.php`

```php
<?php
require "Post.php";

function allPosts() {
    $post = new Post();
    $billets = $post->getAllPosts();
    require "vueAccueil.php";
}
```

Appelées via le Router, `Router.php`

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

function routeRequest() {
    define("DEFAULT_ACTION", "allMessages");
    //Permet de choisir une action si aucune n'est renseignée dans le GET
    $action = isset($_GET['action']) ? $_GET['action'] : DEFAULT_ACTION;
    try {
        if (function_exists($action)) {
            $action();
        } else {
            require "ViewError.php";
        }
    } catch (Exception $e) {
        require "ViewError.php";
    }
}
```

Notez que la gestion d'erreur est déplacée ici afin de gérer à la fois les actions mal renseignées et les erreurs de connexions ; nous n'en avons donc plus besoin dans `index.php`, qui devient simplement :

```php
<?php
require "Router.php";
routeRequest();
```

Nous pouvons aussi définir un fichier `.htaccess` afin de réécrire automatiquement certaines URL ; nous en ferons ici l'abstraction.

8 Ajout de fonctionnalités

Notre structure MVC est prête ! Il pourrait être utile, à terme, de séparer les actions en plusieurs classes, mais nous arrêterons le processus ici.

Si on souhaite ajouter une fonctionnalité à notre application, il faudra :

- Ajouter cette fonctionnalité dans le modèle, c'est-à-dire :
  - Soit créer une classe représentant l'objet à manipuler (et héritant de `Model` pour disposer des accès à la base de données), soit récupérer une classe existante.
  - Ajouter une fonction dans cette classe représentant cette fonctionnalité.
- Ajouter une fonction d'action dans `ControllerActions.php` (ou dans une classe Action).
- Créer la vue correspondant à cette action, et y invoquer le gabarit `template.php`.
- Créer des liens vers cette nouvelle page via des liens, un menu de navigation, etc.

Essayez par exemple de créer une page `index.php?action=Post&id=2` pour afficher un seul billet de blog avec tous ses détails.