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

2020 - 2021
WEBR4-DIN

Développement Internet

Architecture MVC pour une application Web

Dans ce chapitre, nous allons voir comment appliquer les principes de l'architecture MVC (Modèle - Vue - Contrôleur) au développement d'une application Web.

1 Rappel : Architecture MVC

L'**architecture Modèle - Vue - Contrôleur** (MVC) est un patron d'architecture logicielle adapté aux logiciels utilisant des interfaces graphiques, y compris des applications Web [1]. Cette architecture divise l'application en trois composants. L'architecture MVC repose sur le principe de *responsabilité unique* : chaque élément de l'application possède un rôle unique et précis.

- Le **Modèle** représente les données *métier* de l'application, à savoir les données propres au système que l'application modélise (par opposition ux données propres à l'implémentation).
- La **Vue** contient la présentation de l'interface graphique.
- Le **Contrôleur** implémente la logique de traitement des actions effectuées par l'utilisateur.

Le fonctionnement d'une application MVC passe par une boucle d'exécution :

- L'utilisateur interagit avec le programme via le contrôleur.
- Le contrôleur détermine quelle action est demandée par le contrôleur, et la transmet au modèle.
- Le modèle effectue cette action en suivant la logique des données métier.
- Une fois modifié, le contrôleur informe la vue de se mettre à jour afin de représenter le nouvel état des données.
- La vue affiche les nouvelles informations, permettant à l'utilisateur d'effectuer une nouvelle action.

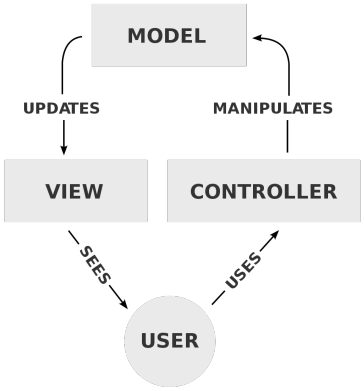

FIGURE 1 – https://en.wikipedia.org/wiki/File:MVC-Process.svg

1. https://fr.wikipedia.org/wiki/Modèle-vue-contrôleur

Version du document : 31 mars 2021

2 Dans une application Web

Lorsqu'on développe une application classique, les données métier pourraient se trouver par exemple dans des fichiers sur l'ordinateur où l'application tourne, et l'interface graphique pourrait se reposer sur un moteur de rendu au choix.

Toutefois, lors de développement d'une application Web, des contraintes supplémentaires s'ajoutent. L'application tourne en effet maintenant sur un serveur distant, accédé via le réseau Internet, et l'interface graphique sera généralement limitée au navigateur Internet de l'utilisateur.

2.1 Modèle

Les données métier doivent être stockées sur le serveur distant : pour cela, une base de données persistante est un outil que vous connaissez bien, et qui est parfaitement approprié dans ce but.

Notre application serveur devra donc être capable de faire l'interface avec cette base de données (en lecture et potentiellement en écriture), et implémenter la logique métier du modèle, c'est-à-dire les règles régissant toute interaction et opération ayant lieu au sein du modèle. Dans ce but, une architecture orienté objet est un exemple d'implémentation très appropriée : chaque classe d'un modèle orienté objet peut traduire une classe de la base de données, reprenant les données métier ainsi que l'implémentation des opérations sous la forme de méthodes.

2.2 Vue

La vue représente les informations qui seront transmises par l'application à l'utilisateur. Dans le cas d'une application Web, il s'agit donc d'envoyer une réponse aux requêtes de l'utilisateur via HTTP.

La vue de l'application aura donc pour tâche de générer des pages Web, par exemple au format HTML (agrémenté de feuilles de style CSS et scripts JavaScript éventuels) répondant à la requête de l'utilisateur.

2.3 Contrôleur

Les interactions de l'utilisateur avec le serveur d'une application Web se font toujours par l'intermédiaire d'une requête au serveur (a priori via HTTP).

Par conséquent, le rôle du contrôleur d'une application Web est de traduire une requête. C'est ce qu'on appelle un **Router** : il fait le lien entre une adresse URL appelée avec une méthode HTTP (GET, POST, etc.) et les manipulation de modèle à effectuer et/ou la vue à afficher en réponse.

3 En pratique : implémentation MVC en PHP

3.1 Modèle

PHP possède plusieurs librairies permettant d'interagir avec des bases de données persistantes. Nous montrerons ici PDO [2], qui a l'avantage de permettre des interactions avec de nombreux schémas de bases de données différents.

Il faudra s'assurer que PDO est installé dans votre exécutable avant de vous en servir (les versions récentes de XAMPP l'incluent normalement par défaut). Une fois cela fait, vous pouvez créer un objet de la classe PDO comme suit (en supposant que la base de données utilise un schéma MySQL) :

```php
<?php
$servername = "localhost";
$username = "username";
$password = "password";

try {
  $conn = new PDO("mysql:host=$servername;dbname=myDB", $username,
                  $password);
  $conn->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
} catch(PDOException $e) {
  // Gestion d'ereur avec $e->getMessage(); pour voir le message
}
?>
```

Une fois l'objet `$conn` créé, sa méthode `PDO::query($sql)` [3] permet de transmettre une requête de sélection contenue dans la variable à la base de données. Le résultat de cette méthode, de type PDOStatement [4]. On peut alors parcourir les résultats de celle-ci, par exemple via la méthode `PDOStatement::fetchAll()`, qui renvoie une matrice mixte contenant les données demandées.

De même, pour une requête qui met à jour les données de la base de données (INSERT, UPDAT, DELETE), la méthode `PDO::exec($sql)` [5] permet de transmettre la requête à la base de données. Cette méthode renvoie comme valeur le nombre de rangs modifiés dans la base de données.

Comme expliqué dans la section précédente, il conviendra de créer des classes représentant chaque objet du modèle. Ces classes se chargeront de générer et d'exécuter les requêtes à la base de données nécessitées par la logique métier du modèle. Il sera souvent pratique de créer une classe abstraite `Model` qui reprendra les informations nécessaires pour communiquer à la base de données (comme la création de l'objet PDO), et de laisser vos classes de modèles hériter de cette classe afin de factoriser votre code.

2. https://www.w3schools.com/php/php_mysql_connect.asp
3. https://www.php.net/manual/en/pdo.query.php
4. https://www.php.net/manual/en/class.pdostatement.php
5. https://www.php.net/manual/en/pdo.exec.php

3.2 Vue

Comme nous l'avons vu au chapitre précédent, PHP est un langage très approprié pour générer des pages HTML de façon dynamique. On notera toutefois une règle de bonne pratique pour structurer une application plus complexe : l'utilisation de gabarits.

Un gabarit est un fichier PHP qui définit la structure d'une page HTML, avec des emplacements prévus pour insérer directement les valeurs de certaines variables. Par exemple, un fichier `template.php` comme ci-dessous :

```
<html>
<head>
    <title><?= $title ?></title>
    <link rel="stylesheet" href="style.css" />
    <script src="script.js"></script>
</head>
<body>
    <header>Website header</header>
    <nav>Website nav</nav>
    <main>
        <h2><?= $pagetitle ?></h2>
        <?= $pagetext ?>
    </main>
    <footer>Website footer</footer>
</body>
```

Note : la syntaxe `<?= $title ?>` est un raccourci pour `<?php echo $title; ?>`

Nos autres fichiers PHP peuvent alors donner les valeurs aux variables placées dans ce template, puis d'inclure le gabarit (*template* en anglais) pour générer immédiatement la page HTML avec la structure voulue. Pour rappel : l'inclusion en PHP via **`require`** ajoute tout code HTML dans la page générée. Par exemple :

```
<?php
    $title="Ma page";
    $pagetitle="Bienvenue sur ma page";
    $pagetext = "<p>Texte de la page</p>";

    require "template.php";
?>
```

Procéder avec cette structure permet de factoriser le code autant que possible :

- La définition de la structure de la page est faite dans des fichiers gabarits, alors que la transmission des données modèle est faite dans d'autres fichiers de vue
- La structure de la page est définie (a priori) à un seul endroit.
- Les éléments communs ne sont définis qu'une fois (pas de code redondant).

Pour des contenus de taille plus conséquente, il conviendra d'utiliser les fonctions de buffer `ob_start()` [6] et `ob_get_clean` [7]. Ces fonctions permettent d'ouvrir un buffer dans lequel vous pouvez placer du code HTML incluant des balises PHP. Par exemple :

```
<?php ob_start(); ?>
    <h1>Tous les messages</h1>
    <table class='messages'>
    <tr><th>Date</th><th>Auteur</th><th>Titre</th></tr>
    <?php foreach ($allMessages as $row): ?>
        <tr>
            <td><?= $row["msg_date"]?></td>
            <td><?= $row["name"]?></td>
            <td><?= $row["title"]?></td>
        </tr>
    <?php endforeach; ?>
    </table>
<?php $content = ob_get_clean(); ?>
```

A l'exécution de ce code, la variable `$content` contiendra tout le code HTML situé entre l'appel aux fonctions de buffer ; en outre, le contenu de la table HTML aura été généré de façon itérative avec le contenu du tableau `$allMessages`.

3.3 Contrôleur

On pourrait, en guise de router, se contenter de créer un fichier PHP pour chaque URL qu'on estime être valide, et appeler au sein de ce fichier PHP la vue qui correspond. Toutefois, on rencontrerait vite un problème d'échelle lorsque notre application grandit en taille, et cette méthode laisserait l'accès aux autres fichiers PHP qui constituent l'application, ce qui n'est a priori pas désirable.

Sur un serveur Apache comme celui créé par XAMPP, le fichier `.htaccess`, placé à la racine des fichiers du serveur, permet de réécrire les URL demandées par des requêtes HTTP. Grâce à cela, nous pouvons rediriger toutes les requêtes vers un fichier PHP unique (typiquement nommé `index.php`). Par exemple, pour transformer toute URL de type `xxx/yyy` en `index.php?action=xxx&id=yyy`.

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

Ce fichier emploie une expression régulière pour réécrire l'URL demandée.

6. https://www.php.net/manual/en/function.ob-start.php
7. https://www.php.net/manual/en/function.ob-get-clean.php

Dès lors, un Router peut être défini :

```php
<?php
require "ControllerActions.php";
function routeRequest() {
    $action = $_GET["action"];
    try {
        if (function_exists($action)) {
            $action();
        } else {
            //Error
        }
    } catch (Exception $e) {
        //Error
    }
}
```

`index.php` peut simplement faire appel à ce Router :

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

Ajouter une page au site revient à ajouter un cas dans la fonction de router, et ajouter une méthode d'action, définies ici dans le fichier `ControllerActions.php` (pas montré pour cet exemple-ci.) Ces actions, faisant aussi partie du contrôleur, font appel au modèle pour récupérer les données utiles ou effectuer les modifications voulues, puis affichent la vue nécessaire pour la réponse via un **`require`**. Elles peuvent également lire les autres paramètres reçues durant la requête HTTP (ou créés par `.htaccess`).

Dans des cas plus complexes, il peut être utile de définir le Router comme une classe qui comprend cette fonction de routing ainsi que des fonctions annexes, par souci de factorisation.

4 Des frameworks pour se simplifier la vie...

Dans la pratique, de nombreux frameworks existent afin de créer des applications Web dotées d'une architecture MVC. Ces frameworks fournissent des librairies de classes et fonctions implémentant déjà de nombreuses fonctionnalités de base communes à toute application Web (router, accès à la base de données, gestion de gabarits,...). Ces frameworks reposent aussi chacun sur des langages de programmation différents, permettant aux développeurs de choisir les outils qui leur conviennent le mieux.

Notons par exemple Laravel et Symfony (PHP), Ruby on Rails (Ruby), Django (Python), ASP.NET (C#.NET - Microsoft), Sails.js (Node.js, une librarie JavaScript permettant de faire tourner un serveur Web en JavaScript).

Ces frameworks sont cités à titre informatif, et cette liste n'est pas exhaustive ; il s'agit ici simplement d'une survol de technologies existantes.