




MVC

Model / View / Controller



LE CONTEXTE

Posons les bases



Pourquoi ?

- La différence entre le mauvais code qui marche et le bon code, c'est le fait que le bon code se doit d'être résilient au changement.
- Je m'explique : Si vous voulez rajouter une fonctionnalité à votre code et que vous risquez de tout faire péter en faisant ça, c'est plutôt mauvais signe
- De plus, un code fouillis est difficile à travailler en équipe, adopter une structure claire pour vos fichiers va vous permettre de travailler à plusieurs, sans risquer de vous marcher sur les pieds.
- Une dernière chose aussi, ça permet d'avoir un code plus facile à relire, personne n'aime lire des fichiers de plus de 1000 lignes, c'est juste imblairable !
- Donc pour résumer :
 - *Plus stable au changement*
 - *Plus facile pour collaborer*
 - *Plus facile à relire*

La logique

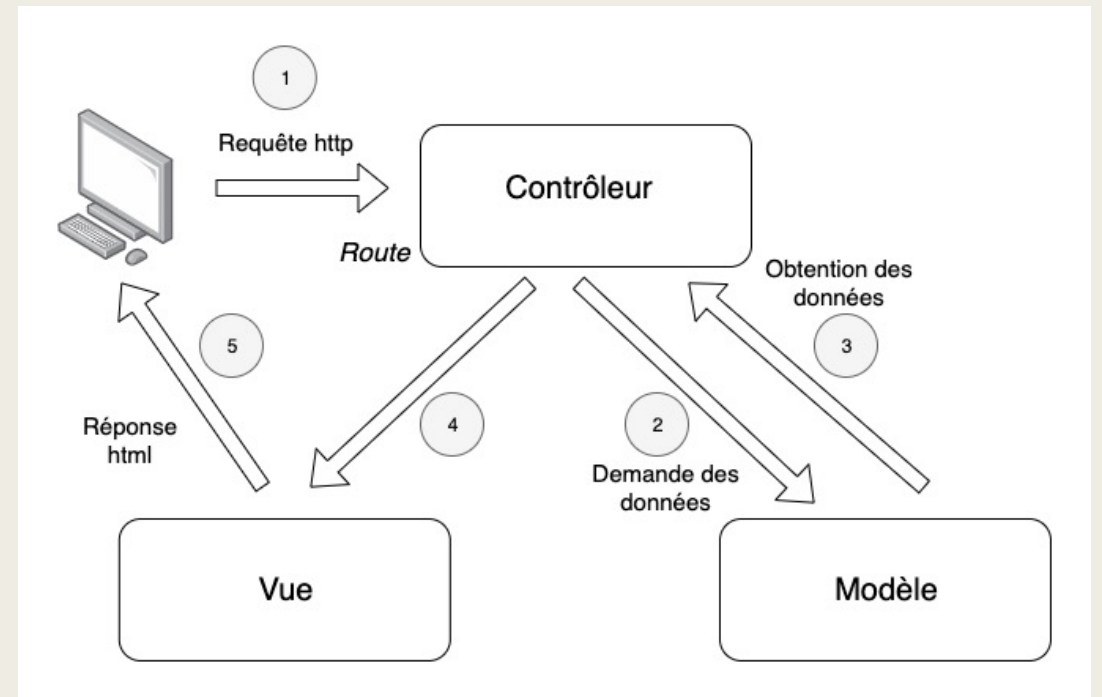
- Une fois ces bases posées, on est en droit de se demander : comment je résous ce problème ?
- La bonne nouvelle c'est qu'avec les années, des milliers de développeurs se sont déjà posé ces questions et ont fini par arriver avec des solutions : **les design patterns**, des « bonnes pratiques » pour résoudre des problèmes précis
- Le design pattern le plus connu est le MVC pour **Model - View - Controller**
- Le but va être de séparer notre code en 3 parties distinctes :
 - *Les models*
 - *Les views*
 - *Les controllers*

Les composantes

- **Model** : C'est la partie qui s'occupe de nos informations brutes, c'est lui qui sera chargé d'aller chercher nos données en DB
- **View** : C'est la partie chargée de l'affichage, elle ne fait quasiment aucun calcul, elle va juste récupérer des variables et les afficher correctement. C'est majoritairement du HTML avec du PHP de base (des boucles et quelques conditions)
- **Controller** : C'est la partie qui gère les conditions et qui prend les décisions. C'est aussi lui qui fait la liaison entre le Model et la View.
Son rôle va donc être de demander les données au Model, de les analyser, prendre les décisions avant de la passer à la View. C'est typiquement ici que va se faire le contrôle d'accès

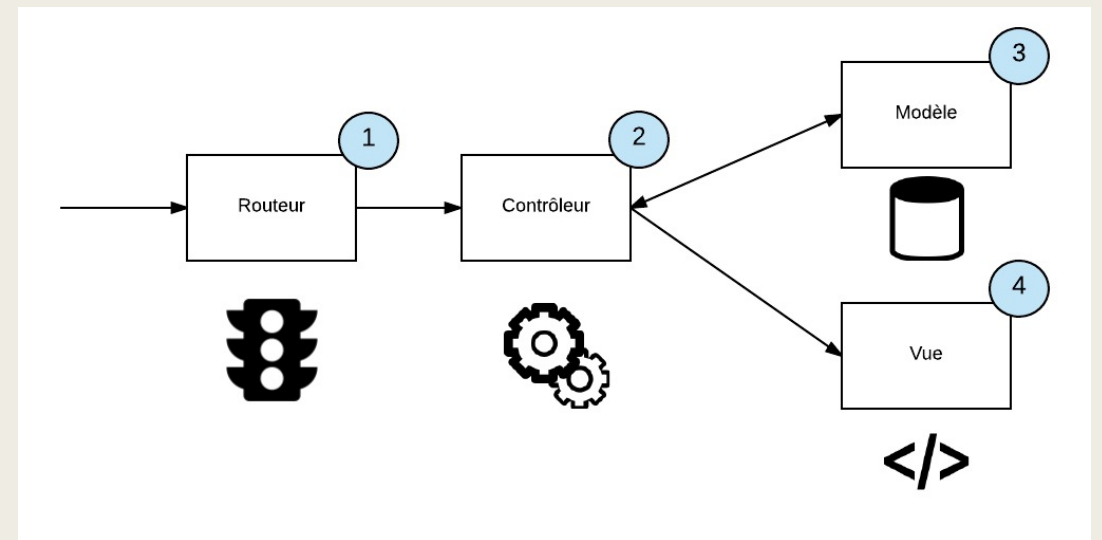
Comment ça parle ?

- On peut donc bien comprendre que le Controller va avoir une discussion à deux sens avec le Model : Il va lui demander des données et le Model va lui répondre
- En revanche, le Controller n'aura une discussion qu'unilatérale avec la View, il ne fait que lui passer des trucs sans rien attendre en retour
- Le Controller sera le chef de toutes ces opérations, c'est à lui que l'utilisateur va demander des trucs et c'est lui qui devra prendre les décisions des acteurs à faire intervenir pour répondre correctement à la demande



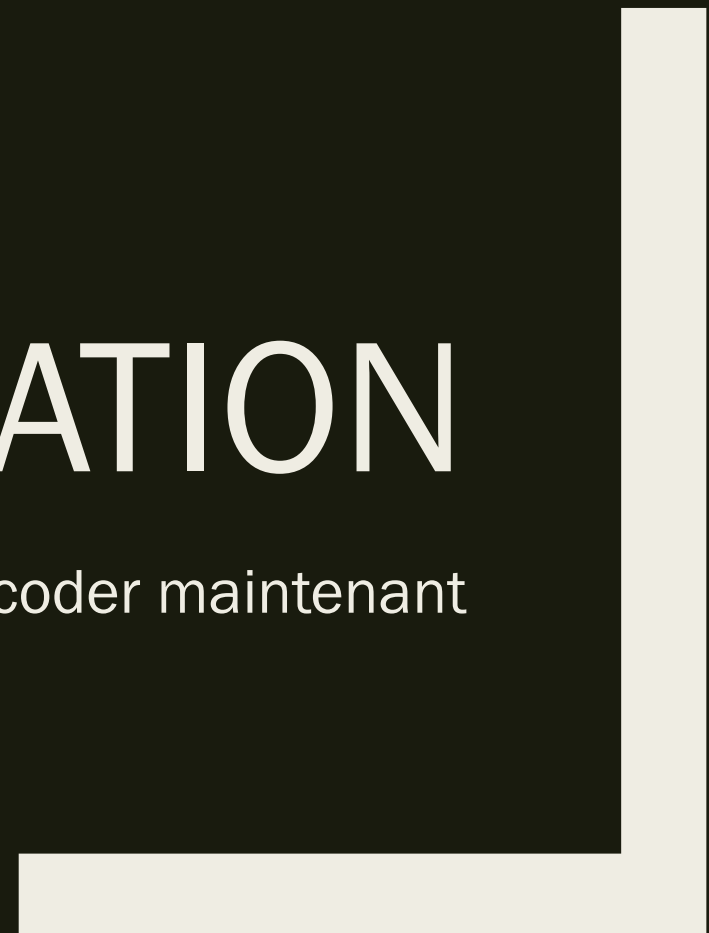
Le routeur

- En regardant le schéma, on est en droit de se poser une question : Comment le controller sait ce que l'on veut faire ?
- La réponse sera évidemment via l'URL mais dans ce cas, il nous manque un élément qui enverra la requête au bon controller : le routeur
- L'utilisateur arrivera donc en premier lieu sur le routeur, qui l'enverra vers le bon controller, qui interrogera le model et passera les données à la view qui sera récupérée par l'utilisateur



EN APPLICATION

Commençons à coder maintenant



Une page d'index

- Imaginons que nous voulions faire un blog, sur la première page, je vais vouloir récupérer mes 5 derniers articles.
- Essayons de faire ça, en adoptant une architecture MVC
- Je vais donc arriver sur index.php qui va devoir me router vers le bon controller
- Ce controller va devoir demander au model les 5 derniers articles
- Ces articles seront récupérées par le model qui interrogera la DB
- Une fois ces articles récupérés, le controller va devoir les passer à la view qui les affichera correctement

Posons nous deux secondes

- On va se poser la question suivante : Comment vais-je représenter ma donnée brute ?
- On sait que l'on veut travailler OO...

```

namespace Entity;

use Model\CommentManager;
use Model\UserManager;

class Post extends BaseEntity
{
    private $id;
    private $date;
    private $title;
    private $content;
    private $authorId;

    /**
     * @return mixed
     */
    public function getId()
    {
        return $this->id;
    }

    /**
     * @return \DateTime
     */
    public function getDate(): \DateTime
    {
        return new \DateTime($this->date);
    }
}

```

L'entité

- Commençons par le concept le plus simple : l'entité
- Elle représente la donnée que vous allez manipuler. Dans le cas d'un billet blog, ça sera un objet avec les mêmes propriétés que celles inscrites en DB
- A cela on va ajouter les getter et les setters

Le model

- On pourrait aussi l'appeler « Manager », c'est lui qui va être chargé de faire les appels en DB et de les calquer sur nos entités pour pouvoir les manipuler en PHP
- On trouve donc ici les appels SQL pour toutes les opérations (Create, Read, Update, Delete)
- Notre model nous renvoie soit une Entité seule, soit un array d'entités

```
class PostManager extends BaseManager
{
    /**
     * @param int|null $number
     * @return array
     */
    public function getPosts(int $number = null): array
    {
        if ($number) {
            $query = $this->db->prepare('SELECT * FROM posts ORDER BY id DESC LIMIT :limit');
            $query->bindValue(':limit', $number, \PDO::PARAM_INT);
            $query->execute();
        } else {
            $query = $this->db->query('SELECT * FROM posts ORDER BY id DESC');
        }
        $query->setFetchMode(\PDO::FETCH_CLASS | \PDO::FETCH_PROPS_LATE, 'Entity\Post');

        return $query->fetchAll();
    }

    /**
     * @param int $id
     * @return Post|bool
     */
    public function getPostById(int $id)
    {
        $query = $this->db->prepare('SELECT * FROM posts WHERE id = :id');
        $query->bindValue(':id', $id, \PDO::PARAM_INT);
        $query->execute();
        $query->setFetchMode(\PDO::FETCH_CLASS | \PDO::FETCH_PROPS_LATE, 'Entity\Post');
        return $query->fetch();
    }
}
```

Stop !

- Encore une fois, posons nous et demandons nous comment nous allons communiquer avec notre base de données

Tirer partie de l'héritage

- Vous l'aurez probablement remarqué, toutes mes classes héritent d'une classe de base, cette classe apportera toutes les fonctions nécessaires au bon fonctionnement de tous les enfants
- Par exemple, tous mes managers auront besoin d'un accès DB, il sera apporté ici
- J'utilise ici le design pattern « Factory »

```
namespace Model;  
  
use Vendor\Core\PDFactory;  
  
abstract class BaseManager  
{  
    protected $db;  
  
    public function __construct()  
    {  
        $this->db = PDFactory::getMysqlConnexion();  
    }  
}
```

On reprend la réflexion

- On va attaquer le controller, essayons de nous demander : A quoi va-t-il ressembler ?

Le controller

```
namespace Controller;

use Model\PostManager;

class FrontController extends BaseController
{
    public function executeIndex(int $number = 5)
    {
        $manager = new PostManager();
        $index = $manager->getPosts($number);

        return $this->render('Page d\'accueil', $index, 'Frontend/index');
    }

    public function executeShow()
    {
        $manager = new PostManager();
        $article = $manager->getPostById($this->params['id']);

        if (!$article) {
            header('Location: /');
            exit();
        }

        return $this->render($article->getTitle(), ['article' => $article], 'Frontend/show');
    }
}
```

- Il sera chargé de faire le pont entre tout le monde, il va recevoir la requête du routeur, interrogera le model et passera les informations à la view
- C'est également ici que se situe la logique d'accès aux données

Le render

- C'est cette fonction qui va faire le pont avec la view, il existe autant de façon de la faire que de développeurs, je vous montre juste la mienne
- Elle est située dans ma classe BaseController est donc disponible dans tous mes controllers
- J'utilise les fonctions de temporisation de sortie pour encapsuler ma view dans une variable qui sera ensuite réinjectée dans mon template

```
/**
 * @param string $title
 * @param array $vars
 * @param string $view
 * @return mixed
 */
public function render(string $title, array $vars, string $view)
{
    $view = $this->viewsDir . $view . '.view.php';
    ob_start();
    require $view;
    $content = ob_get_clean();
    return require $this->template;
}
```

Le template

```
<!DOCTYPE html>
<html lang="fr">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title><?= $title; ?></title>
</head>
<body>
<?= $content; ?>
</body>
</html>
```

- On passe du côté des views. Je les compose généralement de deux fichiers :
 - *Le template qui contiendra le <head> et toute l'ossature « technique » de mon document html*
 - *La view qui ne contient elle, que le contenu dynamique de ma page*
- Je l'ai volontairement laissé très simple pour que le cours reste lisible mais c'est ici que je vais mettre mes liens CSS, Js...

La view

- C'est ici que on va retrouver tout le contenu de la page
- Le but est de n'avoir que très peu de logique dedans (seulement quelques conditions et boucles, rien de plus)

```
<h1>Home Page</h1>
<?php
foreach ($vars as $article) :
    ?>
    <div>
        <h2><?= $article->getTitle(); ?></h2>
        <p><?= substr($article->getContent(), 0, 200); ?></p>
        <a href="/article/<?= $article->getId(); ?>">Lire plus</a>
    </div>
<?php endforeach; ?>
```

Un dernière pose

- Enfin, avant d'aller plus loin, on reprend 5 secondes pour réfléchir à la tronche que va avoir notre `index.php` maintenant

Index.php

```
<?php
session_start();

// Getting the Autoload
require 'Vendor/Core/SplClassLoader.php';
require_once 'Config/autoload.php';

$router = new \Vendor\Core\Router();
$router->getController();
```

- Vous remarquerez que pour l'instant je vous présente des classes, mais je ne vous ai toujours pas dit comment l'info arrive jusqu'à elles dans un premier temps
- Dans un MVC, vous serez toujours sur index.php (ou en tout cas une page unique qui fera office de front controller)
- Cette page va juste recevoir la requête, lancer les sessions, charger les autoload et appeler le router qui sera chargé de redistribuer l'info vers le bon controller

```
isset($_GET['p']) ? $path = $_GET['p'] : $path = null;

switch ($path) {
    // Frontend
    case null:
        $controller = new \Controller\FrontendController();
        $controller->index();
        break;

    case 'show':
        $controller = new \Controller\FrontendController();
        $controller->show($_GET['articleId']);
        break;

    // Admin
    case 'admin':
        $controller = new \Controller\AdminController();
        $controller->index();
        break;

    // Articles
    case 'addArticle':
        $controller = new \Controller\ArticleController();
        $controller->addArticle();
        break;

    case 'deleteArticle':
        $controller = new \Controller\ArticleController();
        $controller->deleteArticle();
        break;
}
```

Le routeur

- Voilà une partie qui devient plus intéressante. Encore une fois il existe autant de façon de faire un routeur que de développeurs
- Le but du jeu va être de passer nos routes en `$_GET` dans l'URL, le routeur va lire l'URL et voir si il a une route associée
- Par exemple « `index.php?p=article&id=12` » devra être lu comme « montre moi un article unique, celui avec l'ID 12 »
- Une solution très bourrin mais efficace est un bête « Switch »

```

use Controller\ErrorController;

class Router
{
    public function getController()
    {
        $xml = new \DOMDocument();
        $xml->load('./Config/routes.xml');
        $routes = $xml->getElementsByTagName('route');

        isset($_GET['p']) ? $path = htmlspecialchars($_GET['p']) : $path = '';

        foreach ($routes as $route) {
            if ($path === $route->getAttribute('p')) {
                $controllerClass = 'Controller\' . $route->getAttribute('controller');
                $action = $route->getAttribute('action');
                $params = [];
                if ($route->hasAttribute('params')) {
                    $keys = explode(',', $route->getAttribute('params'));
                    foreach ($keys as $key) {
                        $params[$key] = $_GET[$key];
                    }
                }
                return new $controllerClass($action, $params);
            }
        }

        return new ErrorController('noRoute');
    }
}

```

Le routeur

- Bon, ok, ça marche mais c'est pas super optimal. Votre routeur déjà ne sera pas réutilisable, en plus on ne sépare pas la logique des données... pas fou
- Une autre façon de faire c'est de mettre les routes dans un fichiers (un XML par exemple) et de faire un routeur qui va lire l'URL d'un côté et comparer aux chemins disponibles sur le XML

Retour vers le controller

- Vous pouvez voir que les noms des actions et du controller sont générés automatiquement à partir des infos du XML
- Je me sers du constructeur du controller pour router vers la bonne méthode

```
abstract class BaseController
{
    protected $params;
    protected $template = __DIR__ . '/../Views/template.php';
    protected $viewsDir = __DIR__ . '/../Views/';

    /**
     * BaseController constructor.
     * @param string $action
     * @param null $id
     */
    public function __construct(string $action, array $params = [])
    {
        $this->params = $params;

        $method = 'execute' . ucfirst($action);
        if (!is_callable([$this, $method])) {
            throw new RuntimeException('L\'action "' . $method . '" n\'est pas définie sur ce module');
        }
        $this->$method();
    }
}
```


Du côté du XML

- Du côté du XML, on garde les choses simples :
 - La route
 - Le controller à interroger
 - L'action à faire
 - Les params à aller chercher en URL
- C'est de suite bien plus propre, on a un routeur réutilisable et la logique séparée des données

```
<?xml version="1.0" encoding="utf-8" ?>
<routes>
  <!-- FrontEnd -->
  <route p="" controller="FrontController" action="index"/>
  <route p="article" controller="FrontController" action="show" params="id"/>
  <!-- Admin -->
  <route p="admin" controller="AdminController" action="index"/>
  <route p="post-comment" controller="AdminController" action="postComment"/>
  <route p="delete-comment" controller="AdminController" action="deleteComment" params="id"/>
  <route p="write-article" controller="AdminController" action="writePost"/>
  <route p="delete-article" controller="AdminController" action="deletePost" params="id"/>
  <route p="update-article" controller="AdminController" action="updatePost" params="id"/>
  <route p="userlist" controller="AdminController" action="userlist"/>
  <!-- Security -->
  <route p="login" controller="SecurityController" action="login"/>
  <route p="logout" controller="SecurityController" action="logout"/>
  <route p="signup" controller="SecurityController" action="signup"/>
  <route p="update-user" controller="SecurityController" action="updateUser" params="id"/>
  <route p="delete-user" controller="SecurityController" action="deleteUser" params="id"/>
</routes>
```

LES MESSAGES D'ALERTE

Passer des alertes à nos utilisateurs

Les flashes

```
class Flash
{
    public static function setFlash(string $message): void
    {
        $_SESSION['flash'] = htmlspecialchars($message);
    }

    public static function hasFlash(): bool
    {
        return isset($_SESSION['flash']);
    }

    public static function getFlash()
    {
        if (isset($_SESSION['flash'])) {
            $message = $_SESSION['flash'];
            unset($_SESSION['flash']);
            return $message;
        }
    }
}
```

- Le plus simple pour passer des messages flash à nos utilisateurs, c'est de les passer en session et de les détruire après la lecture pour éviter la persistance
- Notez que j'utilise des méthodes statiques pour y accéder plus simplement depuis la view

Lire les messages flash

- Du côté de la view, j'ai juste à appeler cette méthode pour afficher et détruire le message si il existe

```
<h1>Please LogIn</h1>

<?php if (\Vendor\Core\Flash::hasFlash()) : ?>
|   <?= \Vendor\Core\Flash::getFlash(); ?>
<?php endif; ?>

<form method="post">
|   <label for="email">Email</label>
|   <input type="email" name="email" id="email" required/> <br/>
|
|   <label for="password">Password</label>
|   <input type="password" name="password" id="password" required/> <br/>
|   <input type="submit" value="LogIn"/>
</form>
```