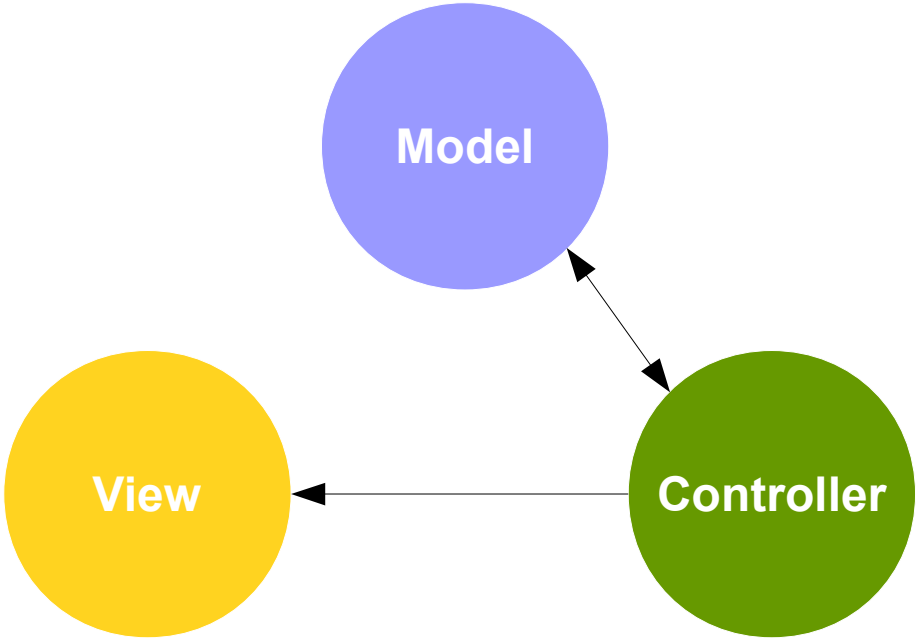
Le design pattern MVC



MVC

Introduction

Dans ce tutoriel, vous découvrirez comment développer une application PHP selon le design pattern MVC (Model-View-Controller).

Nous aborderons la mise en oeuvre de ce design pattern en Programmation Objet.

Présentation

Design pattern MVC

1
Présentation

Présentation

Un **design pattern** est un arrangement caractéristique de modules, reconnu comme bonne pratique en réponse à un problème de conception d'un logiciel.

Il décrit une solution standard, utilisable dans la conception de différents logiciels.

Un des plus célèbres design patterns s'appelle **MVC**, qui signifie :
Modèle - Vue - Contrôleur.

Design pattern MVC

Présentation

Le MVC 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.

Utilisé depuis dans de nombreux langages (Java, frameworks PHP, ASP.NET MVC, etc.), le 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**

Design pattern MVC

- **Présentation**

>> Rôles des composants

- La partie **Modèle** encapsule la logique métier ainsi que l'accès aux données.
  Il peut s'agir d'un ensemble de fonctions (modèle procédural) ou de classes (modèle orienté objet).

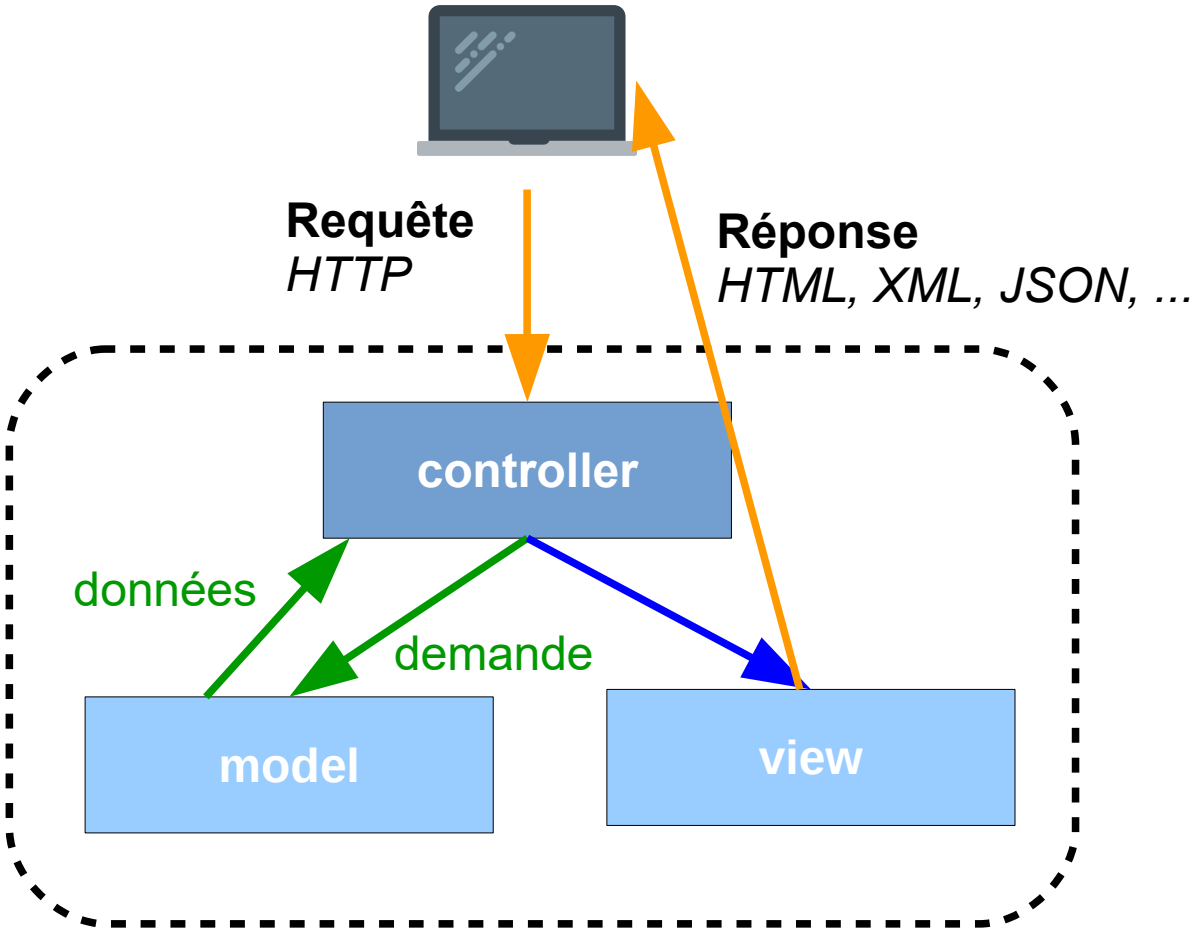
- La partie **Vue** s'occupe des interactions avec l'utilisateur: présentation, saisie et validation des données.

- La partie **Contrôleur** gère la dynamique de l'application. Elle fait le lien entre l'utilisateur et le reste de l'application.

Design pattern MVC

- **Présentation**

>> Interactions entre les composants



Design pattern MVC

- **Présentation**

>> Interactions entre les composants

Le dialogue se passe en 3 temps:

1) le **Contrôleur** reçoit la demande de l'utilisateur (exemple : requête HTTP).
2) le Contrôleur utilise les services du **Modèle** afin de préparer les données à afficher.
3) le Contrôleur fournit ces données à la **Vue**, qui les présente à l'utilisateur (par exemple sous la forme d'une page HTML).

Design pattern MVC

- **Présentation**

>> Avantages

L'intérêt d'organiser son code en MVC est

- de gagner en clareté du code
- optimiser le temps de maintenance
- faciliter l'évolution de l'application

Exemples:

- On pourra modifier l'affichage en changeant la **vue** sans toucher au traitement des données (dans le modèle)
- si on change la partie extraction des données, seul le **Modèle** sera impacté.
- si on ajoute de la logique dans notre traitement principal, seul le **Contrôleur** devra être modifié

Le contrôleur

Design pattern MVC

- **Contrôleur**

C'est le chef d'orchestre : il gère la logique du code qui prend des décisions.

C'est l'intermédiaire entre le modèle et la vue : le contrôleur va demander au modèle les données, les analyser, prendre des décisions et renvoyer le texte à afficher à la vue.

Le contrôleur contient exclusivement du PHP. Il gère notamment les droits d'accès.

Design pattern MVC

- **Contrôleur**

En Programmation Objet nous travaillons avec des classes.

Le contrôleur est donc une ***classe*** (tout comme la vue et le modèle).

C'est à partir du **contrôleur** **et de lui seul** que seront utilisés la **vue** et/ou le **modèle**.

Rappel sur les règles de nommage en POO:

=> le nom de la **classe** commence par une majuscule.

=> Les noms d'**attributs** et de **méthodes** commencent par une minuscule.

Design pattern MVC

- **Contrôleur**

Le contrôleur devra donc a minima:

1) contenir l'appel des scripts correspondant à la vue et au modèle,
2) les déclarer sous forme d'attribut pour pouvoir les manipuler ensuite,
3) instancier les classes (en POO)

Ces opérations étant systématiques, elles peuvent être inclues dans le constructeur de la classe ***Controller***.

Design pattern MVC

- **Contrôleur**

Exemple de classe Controller:

```php
<?php
include 'View.php';
include 'Model.php';
class Controller {
    private $view;
    private $model;
    public function __construct() {
        $this->view = new View();
        $this->model = new Model();
    }
}
```

1
2
3

Design pattern MVC

- **Contrôleur**

Ensuite, le contrôleur devra en fonction de ce qui est demandé (la requête) produire un résultat (la réponse).

Pour cela, on peut, par exemple, analyser les paramètres envoyés dans l'url (récupérés dans le tableau **$_GET**).

Si rien n'est spécifié dans l'url, alors on appliquera une méthode définie par défaut.

Cette partie du code peut être groupée dans une méthode appelée "dispatch".

Design pattern MVC

- **Contrôleur**

Exemple de classe Controller:

```
<?php

public function dispatch() {
    $page = (isset($_GET['page']))?$_GET['page']:"home";

    switch ($page) {
        case 'home':
            //...
            break;

        default:
            break;
    }
}
```

1

2

Design pattern MVC

- **Contrôleur**

Dans chaque cas de figure, il faudra spécifier ce qui est fait.

Ici par exemple, lorsqu'on demande la page "home", on peut afficher la page d'accueil.

S'agissant d'affichage, il faut utiliser la **vue**.

En effet, le contrôleur (tout comme le modèle) ne doit contenir ***aucune instruction d'affichage*** (pas de "echo")**.**

Design pattern MVC

- **Contrôleur**

Exemple de l'appel de la méthode ***displayHome()*** de la vue:

```php
<?php

public function dispatch() {
    $page = (isset($_GET['page']))?$_GET['page']:"home";

    switch ($page) {
        case 'home':
            $this->view->displayHome();
            break;

        default:
            break;
    }
}
```

Design pattern MVC

- **Contrôleur**

Autre exemple avec l'affichage des données d'une table.

Cela se passe en 2 temps:

1) Extraction des données de la table (avec PDO par exemple).
2) Affichage des données récupérées sous forme d'un tableau HTML (ou autres balises).

Le premier travail manipule des données, il revient donc au **Modèle**.

Le second affiche des données, il revient donc à la **Vue**.

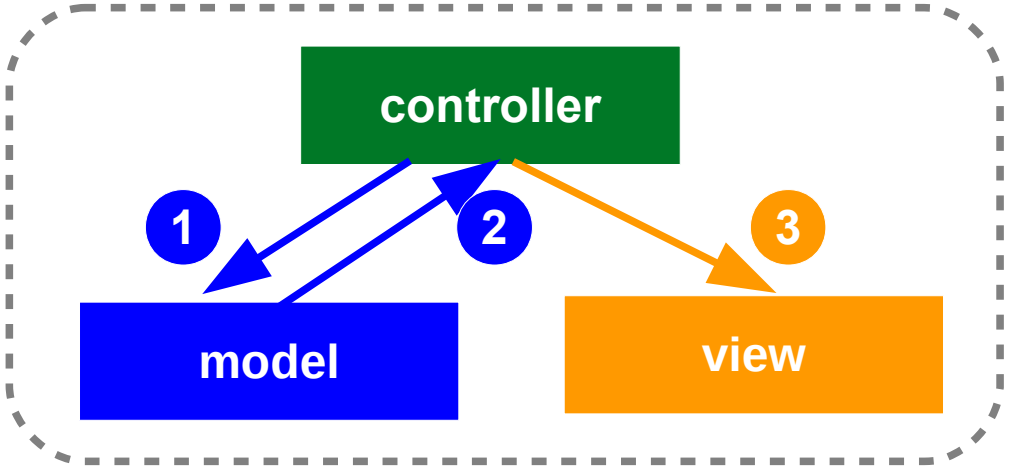
Design pattern MVC

Contrôleur

Le contrôleur va donc dans un premier temps appeler une méthode d'***extraction de données*** dans le **Modèle**.

Il récupère les données en retour de cet appel.

Il va ensuite appeler une méthode de la **Vue** en transmettant ces données réceptionnées ci-dessus afin qu'elles soient affichées.



Design pattern MVC

- **Contrôleur**

Cela se traduit ainsi:

```php
<?php

    switch ($page) {
        case 'home':
            $this->view->displayHome();
            break;

        case 'list':
            $list = $this->model->getList();
            $this->view->displayList($list);
            break;

        default:
            break;
    }
}
```

Design pattern MVC

- **Contrôleur**

Le contrôleur étant une classe, il ne peut pas être lancé tel quel.

Il faut donc mettre en place un point d'entrée de l'application qui peut se faire par exemple via le script ***index.php***

C'est lui qui va se charger d'instancier le contrôleur puis d'appeler la méthode ***dispatch()***.

Le code PHP dans l'index est volontairement minimaliste: toute la logique de l'application se trouve dans le contrôleur.

Design pattern MVC

- **Contrôleur**

Cela se passe en 3 temps:

1) Inclusion du script "**controller.php**" pour disposer de la classe
2) Instanciation d'un objet à partir de la classe **Controller**
3) Appel de la méthode **dispatch()** de cet objet.

```php
<?php
1 include "Controller.php";
2 $controller = new Controller();
3 $controller->dispatch();
```

La vue

Design pattern MVC

- **Vue**

Cette partie se concentre sur l'affichage.

Ici, pas de calcul, ou un minimum : on se contente de récupérer des variables pour savoir ce qu'il faut afficher.

On y trouve essentiellement du code HTML mais aussi un peu de PHP (boucles et conditions) pour gérer l'affichage.

Design pattern MVC

- **Vue**

Dans la vue, on a va créer des méthodes pour afficher le contenu attendu.

Par exemple pour la page d'accueil, on peut imaginer (a minima) le code suivant:

```php
<?php

class View {
    /**
     * Affichage de la page d'accueil
     */
    public function displayHome() {
        echo "Page d'accueil";
    }
}
```

Design pattern MVC

- **Vue**

Bien évidemment, on voudra afficher une page plus conséquente en HTML, avec un header et un footer.

Dans l'idée du MVC et de la POO, on a va séparer au maximum chaque composant afin de pouvoir les utiliser indépendamment les uns des autres.

On va donc créer un ***header.html*** et un ***footer.html*** qui seront stockés dans un dossier à part (appelé "html" par exemple).

On peut imaginer faire de même avec un menu (fichier ***nav.html***), etc.

Design pattern MVC

Vue

Pour que cela fonctionne bien, on va faire en sorte que les différents éléments de la page soit assemblés dans le bon ordre :

- Le **header**
- Le **corps** de la page
- Le **footer**

Pour cela, on ne va pas afficher les éléments directement avec un "***echo***" à chaque fois.

Au lieu de cela, on va les stoker dans un attribut ***$page*** qui sera alimenté au fur et à mesure de la construction de la page.

En fin de traitement, on fera un echo de cet attribut ***$page***.

Design pattern MVC

3
Vue

- **Vue**

Dans le constructeur, on fait l'extraction du code HTML stocké dans le fichier header.html que l'on stocke dans l'attribut $page.

```php
<?php

class View {

    /**
     * Attribut page (contient tout le code HTML à afficher)
     */
    private $page;

    /**
     * Dans le constructeur, affichage du header de la page
     */
    public function __construct() {
        $this->page = file_get_contents("html/header.html");
    }
```

1

Design pattern MVC

- **Vue**

```
/**
 * Alimentation de la page d'accueil
 */
public function displayHome() {
    $this->page .= "Page d'accueil";
    $this->display();
}

/**
 * Affichage de la page
 */
private function display() {
    $this->page .= file_get_contents("html/footer.html");
    echo $this->page;
}
}
```

2

3

Design pattern MVC

- **Vue**

Dans le ***displayHome()***, on ajoute du code HTML propre à cette page.

Puis on appelle la méthode ***display()*** qui va ajouter à ***$page*** le footer (stocké dans le fichier footer.html) et au final, afficher cet attribut ***$page***.

Attention à la manipulation de l'attribut ***$page***: à chaque ajout de contenu, il faut bien sûr **concaténer** le texte avec ce qui est déjà stocké dans cette variable.

En effet, si on ne fait pas cela, on perd le texte précédemment stocké à chaque nouvel ajout.

Design pattern MVC

- **Vue**

Pour l'affichage d'une liste d'items, on développera une méthode ***displayList()*** qui recevra en entrée un tableau contenant les valeurs à afficher.

L'affichage se fera à l'aide d'une boucle foreach() qui va parcourir le tableau et afficher le contenu en ajoutant les balises HTML qui conviennent (exemple <table>, etc.) :

```
/* Alimentation de la page liste
 * à partir d'un tableau reçu par paramètre */
public function displayList($list) {
    $this->page .= "<table>";
    foreach ($list as $element) {
        $this->page .= "<tr>";
        $this->page .= "<td>".
$element['idClient']."</td>";
        $this->page .= "<td>".
$element['societe']."</td>";
        $this->page .= "</tr>";
    }
    $this->page .= "</table>";
    $this->display();
}
```

Le modèle

Design pattern MVC

- **Modèle**

Cette partie gère les données du site.

Son rôle est d'aller récupérer les informations dans la base de données, de les organiser et de les assembler pour qu'elles puissent ensuite être traitées par le contrôleur.

On y trouve les requêtes SQL ou les accès fichiers.

Design pattern MVC

- **Modèle**

L'accès au données peut se faire via un objet de type PDO.
On peut le déclarer comme attribut et l'instancier dans le constructeur afin de pouvoir bénéficier de cette ressource dans toute la classe.

```
class Model {

    const SERVER   = "localhost";
    const USER     = "root";
    const PASSWORD = "";
    const BASE     ="CEFii";

    private $connection;
```

Design pattern MVC

- **Modèle**

Exemple de constructeur pour la classe Model:

```
public function __construct() {
    try {
        $this->connection = new
            PDO("mysql:host=" .
                self::SERVER . ";dbname=" .
                self::BASE, self::USER,
                self::PASSWORD);
    } catch (Exception $e) {
        die('Erreur : ' . $e->getMessage());
    }
}
```

Design pattern MVC

- **Modèle**

Ensuite, on pourra écrire la méthode ***getList()*** qui devra retourner le contenu d'une table (ici, la table "Client"):

```
/**
 * Retourne la liste de toutes les occurrences
 * de la table Client
 */
public function getList() {
    $requete  = "SELECT * FROM client";
    $result = $this->connection->query($requete);

    $list = array();
    if($result) {
        $list = $result->fetchAll(PDO::FETCH_ASSOC);
    }
    return $list;
}
```

MVC multiples

Design pattern MVC

- **MVC multiples**

>> Architecture de l'application

Une application construite sur le principe du MVC se compose a minima de trois scripts.

Cependant, il est fréquent que chaque partie soit elle-même composée en plusieurs éléments.

On peut ainsi trouver plusieurs modèles, plusieurs vues ou plusieurs contrôleurs à l'intérieur d'une application MVC.

On prendra soin alors de les regrouper dans des dossiers avec des noms explicites (ex: Controller, Model, View).

Design pattern MVC

- **MVC multiples**

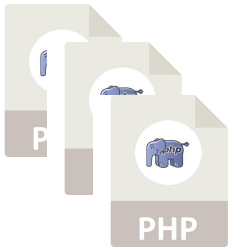
Dans chaque dossier, on aura les composants traitant une entité.

Par exemple, sur une application qui gère les notes des élèves , on peut imaginer avoir un MVC pour la gestion des élèves et un autre MVC pour la gestion des notes.

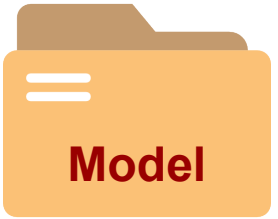
On va utiliser une règle de nommage en concaténant le nom de l'entité avec "Controller" afin que tous les noms de scripts soient construits sur le même modèle.

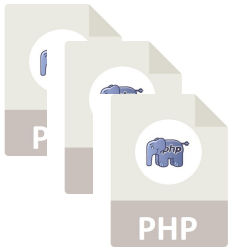
Design pattern MVC

- **MVC multiples**

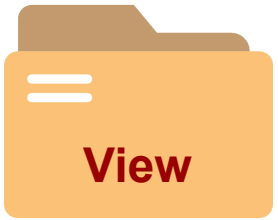


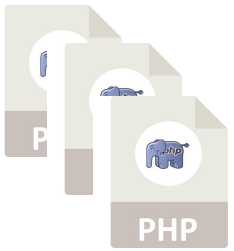
- HomeController.php
- MarkController.php
- StudentController.php





- HomeModel.php
- MarkModel.php
- StudentModel.php





- HomeView.php
- MarkView.php
- StudentView.php

Design pattern MVC

MVC multiples

Toujours dans le même ordre d'idée d'uniformisation du code, on va systématiquement appelé le nom des méthodes avec la chaîne "Action" concaténée.

Par exemple :

- pour afficher la liste des items d'une table, on aura la méthode ***listAction()***.
- pour supprimer un item, on aura ***deleteAction($id)***.
- etc.

Design pattern MVC

- **MVC multiples**

Pour gérer l'appel du bon contrôleur, il faut rajouter une couche supplémentaire dans la logique générale de l'application.

Ceci peut se faire dans l'index.php ou bien on peut créer une nouvelle classe Dispatcher qui va prendre en charge cette partie.

La démarche sera toujours la même: déterminer à partir des éléments reçus dans la requête HTTP ce qui doit être appelé.

Par exemple, on peut imaginer une url du type:

index.php?controller=student&action=list

Design pattern MVC

- **MVC multiples**

Le dispatcher aura besoin de lancer le contrôleur en lien avec l'entité à manipuler (dans notre exemple soit les élèves, les notes).

Il faudra donc inclure les scripts de ces contrôleurs dans la classe Dispatcher afin de pouvoir les instancier au besoin.

```php
<?php

include "Controller/HomeController.php";
include "Controller/StudentController.php";
include "Controller/MarkController.php";

class Dispatcher {

    public function dispatch() {
    }
}
```

Design pattern MVC

- **MVC multiples**

Dans la méthode ***dispatch()***, on récupère les paramètres reçus en GET (ou à défaut, on affecte des valeurs prédéfinies).

Ensuite, on formalise les noms en fonction des règles édictées, en ajoutant "Controller" ou "Action" à chacun des paramètres reçus.

```
public function dispatch() {
    $controller = (isset($_GET['controller']))
                            ?$_GET['controller']:"home";
    $controller = $controller."Controller";

    $action = (isset($_GET['action']))?$_GET['action']:"show";
    $action = $action."Action";
}
```

Design pattern MVC

- **MVC multiples**

Cela facilite l'appel des méthodes dans les classes concernées.

Grâce à l'uniformisation des noms, on a un seul appel pour toutes les méthodes de toutes les classes:

```
public function dispatch() {
    //..
    $my_controller = new $controller();
    $my_controller->$action();
}
```

Pour l'url évoquée en exemple, cela donne:

index.php?controller=student&action=list soit l'appel de la méthode ***listAction()*** de la classe ***StudentController***.

Design pattern MVC

- **MVC multiples**

Voici le code du contrôleur concerné:

```
include 'View/StudentView.php';
include 'Model/StudentModel.php';

class StudentController {

    private $view;
    private $model;

    public function __construct() {
        $this->view = new StudentView();
        $this->model = new StudentModel();
    }

    public function listAction() {
        $list = $this->model->getList();
        $this->view->displayList($list);
    }
}
```

Design pattern MVC

- **MVC multiples**

Pour le modèle et la vue, ça sera le même code que celui vu précédemment pour le MVC simple.

Il faudra simplement tenir compte de l'arborescence avec un niveau supplémentaire de l'application dans l'appel des scripts html dans la vue.

```
/**
 * Constructeur,avec affichage du header de la page
 */
public function __construct() {
    $this->page = file_get_contents("View/html/header.html");
}
```

CEFii
www.cefii.fr

Fin du cours
sur le design pattern MVC