W2 - Frameworks MVC

W-PHP-502

PiePHP

Un framework MVC de A à Z.

1.1

PiePHP

language: PHP

- The totality of your source files, except all useless files (binary, temp files, obj files,...), must be included in your delivery.

Préambule

PiePHP est un framework *home made* basé sur la structure de CakePHP et la norme de Symfony. Il vous permet d'aborder sereinement les projets suivants sur CakePHP, Laravel, et Symfony.

Ce projet est le projet fondateur du module MVC, ne le négligez pas !

Introduction

Vous avez n'avez pas encore appris comment utiliser CakePHP et Laravel, mais ce n'est pas grave car votre curiosité est sans limite !

Nous allons voir dans ce projet, une partie des mécanismes nécessaires au fonctionnement d'un framework MVC (Model/View/Controller). Plus que jamais, vous comprendrez l'intégralité du code que vous produirez : rien ne sera fait au hasard dans votre framework, sinon celui-ci ne fonctionnera pas, c'est une certitude. Tout ce qui vous sera proposé dans ce sujet sera parfaitement adapté à une configuration LAMP classique sur Ubuntu, à vous de l'adapter en fonction de votre configuration.

N'hésitez pas à pousser votre projet au delà des limites proposées par le sujet. En informatique, votre seule limite sera toujours votre imagination.

Ce sujet s'articule sous la forme d'un tutoriel.
Notez qu'il est à interpréter et vous donne uniquement les directions à adopter. À vous de vous renseigner sur les bonnes pratiques et de rendre votre framework utilisable facilement par un autre développeur. Le but est de limiter le code que l'utilisateur devra réaliser au sein de votre framework, en utilisant le principe d'abstraction.

Norme

Pour ce projet, il sera plus que jamais nécessaire de respecter une norme stricte : revoyez le document de norme PHP (PSR-12) de votre piscine !
Jusqu'à présent, vous avez développé vos projets pour vous seul. Cependant, un framework est amené à être utilisé par de nombreux développeurs. Vous allez donc, pour ce projet, devoir respecter une certaine rigueur dans la façon de produire et de documenter votre code.

Il est vital que vous adoptiez cette norme pour ce projet. En respectant la syntaxe et la documentation, vous parviendrez à mieux comprendre votre code et par conséquent, le fonctionnement d'un framework.

Notions

- PHP – POO
- .htaccess
- Framework MVC
- ORM
- Requêtes HTTP
- Moteur de template

Procédure

Projet

Projet

Vous connaissez déjà le design pattern MVC ? C'est très bien, nous allons l'utiliser. Vous devez donc commencer par créer les dossiers, en respectant l'architecture suivante :

Terminal

```
.
|-- config.json
|-- Core
|   |-- autoload.php
|   |-- Controller.php
|   |-- Core.php
|   |-- Database.php
|   |-- Entity.php
|   |-- TemplateEngine.php
|   |-- ORM.php
|   |-- Request.php
|   |-- Router.php
|-- index.php
|-- src
|   | routes.php
|   |-- Controller
|   |   |-- AppController.php
|   |   |-- UserController.php
|   |-- Model
|   |   |-- UserModel.php
|   |-- View
|       |-- App
|       |   |-- index.php
|       |-- Error
|       |   |-- 404.php
|       |-- Flash
|       |   |-- default.php
|       |   |-- error.php
|       |   |-- success.php
|       |-- index.php
|       |-- User
|           |-- index.php
|           |-- login.php
|           |-- register.php
|           |-- show.php
|-- webroot
    |-- assets
    |-- css
    |-- js
```

Vous devez implémenter dans le fichier « /src/View/User/login.php », une page valide W3C contenant un formulaire de type POST avec les champs « email » et « password », dont l'action s'effectuera sur « /index.php ».
Afin de « débuguer » le début de votre conception, vous devez afficher dans le fichier « /index.php », ce

que contiennent les variables « $_POST », « $_GET » et « $_SERVER ».
Utilisez la balise HTML « pre » qui permet d'afficher le contenu des variables de manière plus lisible.
Ouvrez votre navigateur à l'URL « http://localhost/PiePHP/src/View/User/login.php », puis vérifiez votre travail. La page « /index.php » devrait afficher le contenu des variables envoyées (`$_POST`), les arguments de l'URL (`$_GET`), ainsi que toutes les données d'environnement du serveur (`$_SERVER`).

Checkpoint : Attention, il est inutile de poursuivre si vous n'êtes pas arrivé correctement à ce stade. En cas de doute, vérifiez à nouveau votre travail. Au cours de toutes vos expérimentations, pensez toujours à afficher $_GET, $_POST, et $_SERVER.

Lancement de l'application

Vous devez créer une classe « Core » dans un fichier « /Core/Core.php » :

Terminal

```php
<?php

namespace Core;

class Core
{
    public function run()
    {
        echo __CLASS__ . " [OK]" . PHP_EOL;
    }
}
```

Et modifier votre fichier « /index.php » comme suit :

Terminal

```php
<?php

define('BASE_URI', str_replace('\\', '/', substr(__DIR__, strlen($_SERVER['
    DOCUMENT_ROOT']))));
require_once(implode(DIRECTORY_SEPARATOR, ['Core', 'autoload.php']));

$app = new Core\Core();
$app->run();
```

Posez-vous les questions nécessaires pour comprendre parfaitement ce code, puis ouvrez votre navigateur à l'URL « http://localhost/PiePHP/ ». Vous devez constater une erreur `Fatal error: Class 'Core\Core' not found in ...\PiePHP\index.php on line 9`
. Est-ce bien le cas ? Pourquoi ?

Développez maintenant le code du fichier « /Core/autoload.php » qui permettra d'inclure automatiquement toutes les classes qui seront utilisées dans votre framework. Vous devez pour cela utiliser la fonction « spl_autoload_register ».

Attention, si pour l'instant nous n'avons qu'un seul fichier de classe dans le dossier « /Core/ », nous en aurons à l'avenir beaucoup plus, notamment dans « Core » mais aussi dans les dossiers « /src/Model/ », « /src/View/ » et « /src/Controller/ » qui seront respectivement namespacés Model, View et Controller.
Par la suite, vous devrez donc mettre à jour votre autoloader afin qu'il inclue les classes de votre frameworks situées dans le « Core », mais également celles situées dans « src/Controller » pour les classes de controllers du type MyController.php et celles situées dans « src/Model » pour les classes de model du type MyModel.php, etc.
Votre fichier « /Core/autoload.php » sera correct quand vous n'aurez plus aucune erreur dans la page de votre navigateur et que celle-ci affichera « CoreCore [OK] ».

Checkpoint : Attention, il est inutile de poursuivre si vous n'êtes pas arrivé correctement à ce stade. En cas de doute, vérifiez à nouveau votre travail.

Information : Les seuls fichiers écrits en procédural de ce projet sont « /index.php », « /Core/autoload.php » et « /src/routes.php », tous les autres seront orientés objets.

Sécurité de l'application

Ajoutez quelques images dans le dossier « /webroot/assets/ », qui seront utilisées plus tard dans votre site. Mettez un reset CSS (https://goo.gl/ZWNVea) dans le dossier « /webroot/css/ », puis la dernière librairie à jour de jQuery dans le dossier « /webroot/js/ ».

Essayez d'accéder à ces différents fichiers en déduisant les URL de votre navigateur. Tout fonctionne. En réalité, il existe pour l'instant un problème de taille :

- Essayez maintenant d'accéder à l'URL « http://localhost/PiePHP/webroot/ ».
  Quel est le problème ?
- Encore pire, ouvrez l'URL « http://localhost/PiePHP/Core/ ».
  Comprenez-vous quel est le problème ?

En réalité, on ne devrait pas pouvoir se balader dans votre architecture en changeant les URL dans son navigateur. Les visiteurs connaissent maintenant tout de vos fichiers, ce n'est pas bien !

Autre chose, avez-vous essayé une URL qui ne devrait pas exister ?
Par exemple, l'URL « http://localhost/PiePHP/inexistant.php » vous renvoie vers une page d'erreur 404 par défaut, que vous ne pouvez a priori pas éditer.

Qu'à cela ne tienne, vous voulez prendre le contrôle sur ce que voient, ou non, vos visiteurs. Vous devez donc créer un fichier « /.htaccess » qui servira à configurer le serveur HTTP Apache correctement en fonction de votre politique de sécurité. Le vôtre devra contenir le code suivant :

Terminal

```
RewriteEngine On
RewriteBase /my/path/to/PiePHP/
RewriteCond %{REQUEST_FILENAME} !-f [OR]
RewriteCond %{REQUEST_URI} !^/my/path/to/PiePHP/(webroot/.+|index.php|robots.txt)$
RewriteRule ^ index.php [QSA,L]
```

Ainsi, les seuls fichiers autorisés à s'afficher dans votre navigateur sont ceux contenus dans le dossier « /webroot/ », les fichiers « /index.php » et « /robots.txt ». Ce dernier n'existe pas encore, c'est à vous de le créer, maintenant. En effet, vous devrez savoir expliquer en soutenance son utilité.

Vous pouvez à présent tester n'importe quelle URL, vous devriez toujours tomber sur votre page « /index.php » (sauf pour un fichier existant dans « /webroot/ » ainsi que pour le fichier « /robot.txt ») :

- http://localhost/PiePHP/
- http://localhost/PiePHP/Core/Core.php
- http://localhost/PiePHP/webroot/
- http://localhost/PiePHP/un/autre/exemple
- http://localhost/PiePHP/articles?exemple=42

Faites d'autres tests que ceux fournis ci-dessus, allez-y !

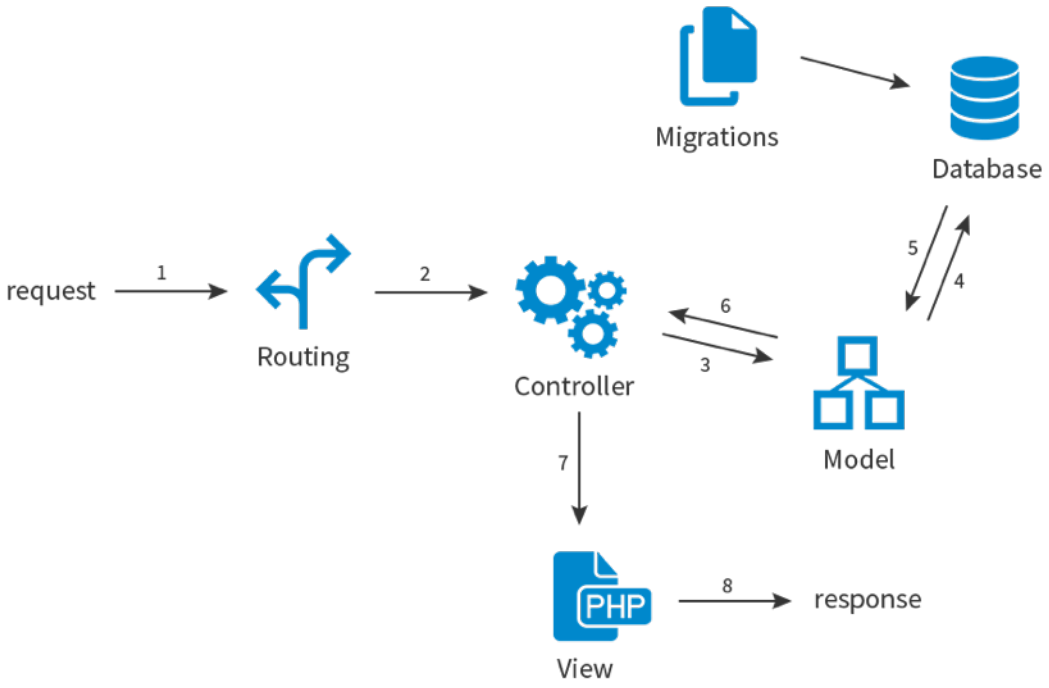
N'est-ce pas fort intéressant ? Vous comprendrez un peu plus tard tout l'intérêt de cette configuration. Pour l'instant, retenez que tout est centralisé sur votre page « /index.php », qui lance votre application et dans laquelle vous pourrez récupérer les données « $_GET », « $_POST » et « $_SERVER ».

Checkpoint : Attention, il est inutile de poursuivre si vous n'êtes pas arrivé correctement à ce stade. En cas de doute, vérifiez à nouveau votre travail.

Fonctionnement de l'application

Maintenant que notre application est lancée et que notre sécurité est posée, nous allons pouvoir mettre en relation les dossiers « /models/ », « /views/ » et « /controllers/ ». Vous souvenez-vous de l'interaction entre ces 3 « dossiers » ? Si vous ne savez pas, faites des recherches avant de continuer.

Architecture MVC



- Le routing permet de récupérer les requêtes utilisateur (1) et de lui fournir les ressources qu'il demande (8).
- Le controller est la classe qui contient la logique. Elle gère le model (6, 3) et affiche les views (7). Un controller contrôle !
- Chaque table de votre base de données à son propre model qui va gérer les requêtes en BDD (4, 5) sur cette table. C'est ici que vous devez définir les relations entre vos tables.
- Les migrations sont des fichiers contenant la configuration de votre BDD. Ils permettent de mettre à jour la BDD sans accès direct à celle-ci (optionnelle dans ce sujet mais très courante dans les frameworks MVC).
- Les views sont des fichiers PHP qui ne contiennent que de la logique d'affichage.

Le routing

Pour illustrer le sujet et tester son code, nous allons créer notre première entité. Créez donc une table « users » qui contient une clé primaire « id », un « email », et un « password ».
Créez également les classes du model (« /src/Model/UserModel.php ») et du controller (« /src/Controller/User-Controller ») qui seront associés à cette table ; ces trois éléments sont le cœur de ce que l'on appelle une entité.

Routage statique

Nous allons tous d'abord créer un router statique qui abstrait le nom de la route du nom du controller.
Créez une classe Router et une classe Controller dans « /Core ». Tous vos controlleurs doivent hériter de `\Core\Controller`.

Votre classe Router doit avoir une méthode connect qui permet d'ajouter une route à votre framework, et une méthode get qui renvoie un tableau contenant le controller et l'action à appeler en fonction des routes pré-définies dans connect et de l'URL courante.

Terminal

```
class Router
{
    private static $routes;
    public static function connect ($url, $route)
    {
        self::$routes[$url] = $route;
    }

    public static function get ($url)
    {
        // retourne un tableau associatif contenant
        // - le controller a instancier
        // - la methode du controller a appeler
    }
}
```

Mettez à jour la méthode run de votre Core pour qu'elle charge votre fichier « /src/routes.php » qui contient les appels à la méthode connect, puis qu'elle utilise la méthode get de la classe Router afin de récupérer la route adéquate.

Par exemple

Après avoir ajouté ce contenu au fichier src/routes.php :

Terminal

```
Router::connect('/', ['controller' => 'app', 'action' => 'index']);
Router::connect('/register', ['controller' => 'user', 'action' => 'add']);
```

L'URL http://localhost/PiePHP/ instanciera le controller AppController et appellera sa méthode indexAction.
L'URL http://localhost/PiePHP/register instanciera le controller UserController et appellera sa méthode addAction.

Routage dynamique

Gardez bien le code de vos deux routeurs, ils vous sera demandé de présenter les deux lors de la soutenance.

Ok, c'est pas mal, mais ça pourrait peut-être être pratique d'avoir un router pour lequel on ne doit pas définir chaque route à la main.
Si ce n'est pas déjà fait, mettez à jour le Core pour qu'il instancie le Controller fourni en premier paramètre dans l'URL et appelle sa méthode fournie en deuxième paramètre (ce qu'on appelle l'action).

Par exemple : L'URL http://localhost/PiePHP/user/add doit instancier le controller UserController et appeler sa méthode addAction (vous pouvez commencer vos expérimentations en testant l'URL http://localhost/PiePHP/?c=user&a=add). N'oubliez pas de mettre à jour votre autoloader.

Lorsque le controller ou l'action n'est pas présent, il faudra les remplacer par « app » et « index » respectivement. Donc /user appellerait le UserController et la Méthode indexAction.

Si le controller ou l'action fournie n'existe pas, il faut afficher le message : « 404 ».

Choix de routeur

Maintenant que vous avez codé vos deux routeurs, libre à vous de choisir celui que vous préférez utiliser. Si vous n'arrivez pas à vous décider peut-être pourriez-vous essayer de faire un routeur hybride.

Le Controller

Créez les views index, login, register, et show dans « /src/View/User », et le layout « index » dans « /src/View ».

Votre Controller doit implémenter une fonction render qui doit afficher la view passée en paramètre dans le layout index.
L'attribut $_render sera affiché à la toute fin de l'exécution du script.

Savez-vous ce qu'est un layout et à quoi cela sert-il ?

La vue doit être rendue à la fin afin d'éviter l'envoi de plusieurs headers HTML.
Trouvez un moyen astucieux de faire cela. N'existe t-il pas l'inverse de __construct qui est appelé à la construction d'un objet ?

La méthode render :

Terminal

```
protected function render($view, $scope = []) {
    extract($scope);
    $f = implode(DIRECTORY_SEPARATOR, [dirname(__DIR__), 'src', 'View', str_replace
        ('Controller', '', basename(get_class($this))), $view]) . '.php';
    if (file_exists($f)) {
        ob_start();
        include($f);
        $view = ob_get_clean();
        ob_start();
        include(implode(DIRECTORY_SEPARATOR, [dirname(__DIR__), 'src', 'View', '
            index']) . '.php');
        self::$_render = ob_get_clean();
    }
}
```

Exemple de layout :

Terminal

```
<!doctype html>
<html lang="en">
<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" />
    <title>Pie PHP</title>
</head>
<body>
    <?= $view ?>
</body>
</html>
```

Exemple d'utilisation:

Terminal

```
<?php
class UserController extends \Core\Controller {
  function addAction() {
    $this->render('register') // Va rendre la vue src/View/User/register.php
  }
}
```

Checkpoint : Attention, il est inutile de poursuivre si vous n'êtes pas arrivé correctement à ce stade. En cas de doute, vérifiez à nouveau votre travail.

Mise en pratique

Nous allons créer un formulaire d'inscription simpliste.

Le Model

Créez un model UserModel dans « /src/Model » qui contient deux attributs privés email et password et une méthode save qui ajoute un enregistrement en BDD avec les attributs du model

La View

Dans votre view register, créez un formulaire prenant en paramètre une adresse email et un password dont l'action redirige vers l'action registerAction de votre UserController.

Le Controller

Créez une registerAction dans votre UserController qui instancie votre UserModel, récupère les paramètres de la requête POST, met à jour les attributs du model et appelle sa méthode save.

Pour aller plus loin

Créez une fonctionnalité de login en utilisant votre view login et en mettant à jour votre controller et votre model.

Le Model

Le model est une interface directe avec la BDD. Il s'occupe du CRUD des entités de votre framework (une entité est généralement composée d'un controller, d'un model, et de leurs vues).

En général, un model contient donc les fonctions suivantes :

- create (créé une nouvelle entrée en base avec les champs passés en paramètres et retourne son id)
- read (récupère une entrée en base suivant l'id de l'user)
- update (met à jour les champs d'une entrée en base suivant l'id de l'user)
- delete (supprime une entrée en base suivant l'id de l'user)
- read_all (récupère toutes les entrées de la table user)

Initialisez une connexion à la base de données dans le constructeur et implémentez ces méthodes dans votre model user.

Request

Créez une classe Request dans « /Core ». Cette classe doit récupérer les paramètres des requêtes HTTP POST et GET et sécuriser ces inputs (trim, stripslashes, htmlspecialchars, etc).
L'objet Request doit être instancié dans le constructeur de tous vos controllers.

Avancé - L'ORM

Checkpoint : Si vous ne souhaitez pas étudier l'implémentation d'un ORM simple, lisez tout de même l'énoncé, puis passez à la partie suivante.

L'ORM est une technique permettant de manipuler vos entités plus facilement en faisant abstraction des requêtes SQL.

Ici nous vous demandons de créer un mini-ORM (qui se présentera en réalité sous la forme d'un helper) permettant d'insérer, voir, modifier, et supprimer un enregistrement sans passer par les requêtes SQL dans vos models.

Pour cela vous devez créer une classe ORM dans « /Core ». Cette classe contiendra les méthodes create, read, update, delete, et find.

Les prototypes de ces fonctions sont :

Terminal

```
public function create ($table, $fields) {}          // retourne un id
public function read ($table, $id) {}                // retourne un tableau
    associatif de l'enregistrement
public function update ($table, $id, $fields) {}     // retourne un booleen
public function delete ($table, $id) {}              // retourne un booleen
public function find ($table, $params = array(
    'WHERE' => '1',
    'ORDER BY' => 'id ASC',
    'LIMIT' => ''
)) {}                                                // retourne un tableau d'
    enregistrements
```

Usage :

Terminal

```php
$orm = new ORM();
$orm->create('articles', array(
    'titre' => "un super titre",
    'content' => 'et voici une super article de blog',
    'author' => 'Rodrigue'
));
$orm->update('articles', 1, array(
    'titre' => "un super titre",
    'content' => 'et voici un super article de blog',
    'author' => 'Rodrigue'
));
$orm->delete('articles', 1);
```

LES ENTITéS

À présent nous allons abstraire les CRUDS de l'ORM en utilisant des entités. Créez une classe Entity dans le « /Core » et faites extends vos models de cette classe.

Le constructeur de votre Entity doit :

- Prendre en paramètre un tableau associatif contenant les attributs d'un model (ici : titre, content, author).
  OU
- Prendre en paramètre un tableau associatif contenant une entrée dont la clé est « id » et la valeur l'id de l'enregistrement géré par l'entité. Le retour du read de l'ORM doit remplacer le tableau associatif.
  PUIS
- Créer pour chaque attribut fourni dans le tableau associatif un attribut public dont le nom sera fourni par la clé et la valeur fourni par la valeur de l'entrée du tableau.

Usage dans le cadre de registerAction :

Terminal

```php
$params = $this->request->getQueryParams();
$user = new UserModel($params);
if (!$user->id) {
    $user->save();
    self::$_render = "Votre compte a ete cree." . PHP_EOL;
}
```

Avancé - Les relations de model

Nous allons enfin mettre en place les relations de model. Les relations permettent de lier deux models entre eux afin que l'un soit accessible dans l'autre (relation unidirectionnelle) et inversement (relation bidirectionnelle).

Par exemple, dans un blog, les articles ont une relation avec leurs commentaires. C'est ce que l'on appelle une relation « one to many » ou « 1 : N ».

Il existe trois types de relations :

- 1 : 1 (Un téléphone est associé à un utilisateur, et un utilisateur est associé uniquement à ce téléphone)
- 1 : N (Un article peut avoir plusieurs commentaires, et un commentaire appartient à un article)
- N : N (Un film peut avoir plusieurs genres, et un genre peut avoir plusieurs films)

Ici, la relation 1 : 1, d'un point de vue fonctionnel, peut être réduite à la relation 1 : N et nous ne l'aborderons donc pas.

One to Many

Une relation 1 : N peut se traduire comme tel : « un article possède plusieurs commentaires et un commentaire appartient à un article ».

En BDD, la table commentaires doit donc avoir un champ article_id qui fera le lien avec la table article.
En SQL pour obtenir les commentaires d'un article il faudra utiliser une jointure (INNER JOIN).

Vous devez donc représenter ce mécanisme dans vos models, et dans votre ORM.
Ajoutez un attribut privé statique $relations dans vos models. Cet attribut sera un tableau qui peut contenir les valeurs :

- 'has many' . $table
- 'has one' . $table

Par exemple :

- Le model ArticleModel doit avoir une relation 'has many comments'
- Le model CommentModel doit avoir une relation 'has one article'

Enfin, vous devez permettre à l'utilisateur de votre framework, lorsqu'il utilise les méthodes read et find de votre ORM d'obtenir tous enregistrements liés à l'enregistrement demandé.

Par exemple, si votre utilisateur read un article, le tableau associatif retourné doit contenir une entrée comments.

Cette même entrée comments contient un tableau.
Ce dernier contient tous les enregistrements de la table comments lié à votre enregistrement d'article.

Many to Many

Une relation « many to many » ou « N : N » peut aussi être traduite comme une relation 1 : N combinée à une relation N : 1.

Par exemple, dans le cas d'un blog, on pourrait envisager d'établir une relation many to many entre les tags et les articles.
On a une table tags sans duplicata, et une table articles.
Un article peut posséder plusieurs tags, et un tag peut appartenir à plusieurs articles.
Dans ce cas-là, et de manière simpliste, comme dans une relation one to many classique on aura un champ article_id dans la table tags mais également, en plus, un champ tag_id dans la table articles.

Pour une utilisation plus logique de la BDD, il est d'usage de regrouper ces deux clés étrangères dans une table séparée que l'on appelle table pivot (elle peut être nommée articles_tags) et qui contient ces deux clés auxquelles on peut adjoindre certaines informations du type date, etc.

Faites-en sorte que vos models et votre ORM supportent les relations many to many de la façon suivante :

- La variable $relation du model ArticleModel a la valeur 'has many tags'
- La variable $relation du model TagModel a la valeur 'has many articles'

Avancé - Routeur avec paramètre

Occasionnellement, vous aurez besoin de mettre des paramètres à votre Route, afin de masquer des paramètres GET ou POST.

Paramètres obligatoires

Souvent, vous devrez récupérer la valeur d'un paramètre d'une route, par exemple pour récupérer l'ID d'un utilisateur.

Terminal

```
Router::connect('/user/{id}', ['controller' => user, 'action' => 'show']);
```

Terminal

```
<?php

namespace Controller;

class UserController
{
    public function show($id)
    {
        echo "ID de l'utilisateur a afficher : $id" . PHP_EOL;
    }
}
```

Paramètre optionnel

Ces paramètres peuvent aussi être optionnels, en plaçant un « ? » après le nom du paramètre. Assurez-vous de donner une valeur par défaut à la variable correspondante, dans la méthode de votre controller.

Avancé - le moteur de templates

Checkpoint : Si vous ne souhaitez pas étudier l'implémentation d'un moteur de template, lisez tout de même l'énoncé, puis passez à la dernière partie (my_cinema by PiePHP).

Mettez à jour votre fonction render afin qu'elle parse la view à la recherche de tags et qu'elle remplace le contenu de ces tags de la façon suivante :

{{ $welcome_text }}	< ?= htmlentities ($welcome_text) ?>
@if (count($records) === 1) I have one record! @elseif (count($records) > 1) I have multiple records! @else I don't have any records! @endif	< ?php if (count($records) === 1): ?> I have one record! <?php elseif (count($records) > 1): ?> I have multiple records! <?php else: ?> I don't have any records! < ?php endif ; ?>
@foreach ($users as $user) This is user {{ $user->id }} @endforeach	< ?php foreach ($users as $user): ?> This is user <?= htmlentities($user->id) ?> <?php endforeach; ?>
@isset($records) <!-- $records is defined and is not null... --> @endisset	< ?php if (isset($records)): ?> <!-- $records is defined and is not null --> <?php endif; ?>
@empty($records) <!-- $records is "empty"... --> @endempty	< ?php if (empty($records)): ?> <!-- $records is "empty"... --> <?php endif; ?>

Il est fortement conseillé pour cet exercice d'être à l'aise avec les regex et les fonctions preg_replace de PHP. Vous trouverez un cours rapide sur les regex ici : https://regexone.com/ et un excellent site de test ici : https://regex101.com/

Ce comportement est basé sur un moteur de template bien connu, blade (https://laravel.com/docs/master/blade). Renseignez-vous donc bien dessus !

Pour les plus aventureux, vous pouvez implémenter un système de caching des vues, c'est-à-dire de les stocker, préparsées, dans un dossier spécifique pour un certain laps de temps et les recompiler par moment.

My cinema by PiePHP

L'objectif de cette dernière partie est de mettre en lumière l'apport d'un framework en termes de simplicité et de rapidité de programmation dans un projet aux entités simples telles que my_cinema.

Voyez-le comme un site communautaire où les utilisateurs peuvent mettre à jour une banque de films et mettre en avant leurs historiques. Implémentez ces fonctionnalités :

- Entité User
    - Inscription

- Connexion
- Voir son profil
- Modifier son profil
- Supprimer son profil

- Entité Film
  Film 'has one' Genre.
  - Afficher les films de la BDD dans une page d'accueil
    - Au clic sur un film, afficher le détail du film dans une nouvelle page
  - Ajouter un film
  - Modifier un film
  - Supprimer un film

- Entité Genre
  Genre 'has many films'.
  - Ajouter le genre à la description détaillée du film
  - Ajouter un genre
  - Modifier un genre
  - Supprimer un genre

- Entité Historique
  User 'has many historiques' et Historique 'has one film'.
  - Ajouter un film que l'on a vu dans la table historique
  - Supprimer un film que l'on a vu de la table historique

Conclusion

Voici qui conclut le projet PiePHP, vous êtes à présent en possession d'un framework quasiment complet, et vous avez acquis de nombreuses notions qui vous seront particulièrement utiles pour le reste de votre formation.

Documentation

Conventions

- https://www.php-fig.org/psr/psr-12/
- http://symfony.com/fr/doc/current/contributing/code/standards.html
- http://symfony.com/fr/doc/current/contributing/code/conventions.html
- https://anandarajpandey.com/2015/05/10/mysql-naming-coding-conventions-tips-on-mysql-database/

Frameworks

- https://book.cakephp.org/3.0/en/index.html
- https://symfony.com/doc/current/index.html
- https://laravel.com/docs/5.5/
- https://silex.symfony.com/doc/2.0/
- https://docs.phalconphp.com/en/3.3
- http://flightphp.com/

Tutoriels / Explications

- https://www.sitepoint.com/the-mvc-pattern-and-php-1/
- https://www.sitepoint.com/the-mvc-pattern-and-php-2/