

Instantly share code, notes, and snippets.



tomshahp / 01-tp.md

Last active 2 months ago

[Star 4](#) [Fork 0](#)[Code](#) [Revisions 6](#) [Stars 4](#)[Embed](#) [Raw](#) [<script src="https://gist.github.com/tomshahp/01-tp.md">](#) [Download ZIP](#)

TP : PHP-POO-MVC

[01-tp.md](#)

Cours de PHP, POO et MVC avancé

Résumé du cours

Nous allons créer un projet en MVC contenant au moins trois tables : deux tables jointes par une relation N-N, avec une table de jointure entre les deux. Le projet sera développé en architecture MVC et en utilisant des packages Composer.

Les projets sont les suivants :

Liste des projets

Projet 1 : gestion d'évènements pour Eventbrite

Vous allez créer un système de gestion d'évènements pour Eventbrite : l'utilisateur pourra créer des évènements et des utilisateurs, et les utilisateurs pourront s'inscrire à un ou plusieurs évènements. De même, on pourra ajouter un ou plusieurs utilisateurs à un évènement.

- Un évènement a plusieurs utilisateurs.
- Un utilisateur a plusieurs évènements.

Projet 2 : tracker de films Netflix

Vous allez créer un tracker pour Netflix qui permettra de répertorier les films vus par un utilisateur, ainsi que la liste des utilisateurs ayant vu un film.

- Un utilisateur a vu plusieurs films.
- Un film a été vu par plusieurs utilisateurs.

Projet 3 : gestion d'un panier de e-commerce pour Amazon

Vous allez créer un système de panier pour Amazon : on pourra ajouter plusieurs produits à un client, et on pourra ajouter plusieurs clients à un produit.

- Un client a plusieurs produits.
- Un produit a plusieurs clients.

Projet 4 : création de playlists Spotify

Vous allez créer un système de playlists pour Spotify : un utilisateur pourra ajouter plusieurs musiques à sa playlist personnelle, et une musique pourra être ajoutée dans les playlists de plusieurs utilisateurs.

- Une playlist a plusieurs musiques.
- Une musique appartient à plusieurs playlists.

Projet 5 : bibliothèque de prêts Kindle

Vous allez créer le système de gestion de prêts de la bibliothèque Kindle. Un lecteur pourra louer plusieurs livres, et un livre pourra être loué à plusieurs utilisateurs.

- Un lecteur a plusieurs livres.
- Un livre est loué par plusieurs lecteurs.

Projet d'exemple

Tous ces projets vont pouvoir se baser sur le projet suivant. Vous allez adapter les tables et modèles pour correspondre à

Tous vos projets vont pouvoir se calquer sur le projet suivant. Vous allez adapter les tables et modèles par rapport à l'exemple ci-dessous :

Projet d'exemple : système de gestion de cours pour Harvard

Nous allons créer un système de gestion de classe et de cours.

- Chaque étudiant a plusieurs cours
 - Chaque cours accueille plusieurs étudiants

Vous créerez les interfaces suivantes :

AJOUTS

- Ajout d'un nouvel étudiant
 - Ajout d'un nouveau cours
 - Attribution de plusieurs cours à un étudiant

LISTES

- Liste des étudiants
 - Liste des cours
 - Liste des inscriptions (liste de tous les cours et étudiants inscrits)

PAGES

- Page d'un étudiant et la liste de ses cours
 - Page d'un cours et la liste des étudiants inscrits
 - Dans chacune de ces pages, on pourra éditer les informations d'un étudiant ou les informations d'un cours
 - Dans chacune de ces pages, on pourra supprimer les cours d'un étudiant, ou supprimer un étudiant d'un cours

Travail avec Git

Vous travaillez en groupe avec Git :

1. Un développeur crée le dépôt Git sur Github, en privé ou public.
 2. Il invite en administrateur les autres développeurs.
 3. Les autres développeurs vont cloner le dépôt Git sur leur ordinateur.
 4. Tout le monde pourra effectuer des commits et les pusher sur le même repository.

Résultat attendu

Vous trouverez une idée du résultat attendu ici : <https://imgur.com/a/vinByNa>. N'hésitez pas à adapter à votre projet ! Pensez à soigner le front une fois le back bien avancé.

Développement de l'exercice

1. Création de la base de données

1. Vous créerez le modèle conceptuel de base de données sur papier, en listant les tables, les relations entre les tables, la liste des champs à insérer dans les tables.
 2. Une fois le modèle validé, vous créerez la base de données.

2. Préparation des routes nécessaires

1. Vous listerez dans un document toutes les routes nécessaires à ce projet et leurs méthodes (`GET students` , `POST student` ...).

3. Préparation du projet

1. Vous utiliserez le projet `base-mvc` que vous installerez (ouverture du projet avec VSCode puis `composer install` dans la console, et modification des fichiers de config de sorte à ce qu'ils correspondent à votre base de données, vos URL...).

4. Rédaction des routes

1. Traduisez les routes que vous aurez préparé au point 2. dans le fichier de routes de votre projet.
 2. C'est l'occasion de prévoir les contrôleurs nécessaires et les méthodes associées !

∅ 5. Création des controllers

1. Créez les controllers nécessaires au projet et les méthodes associées.

6. Creation des vues

1. Dans les controllers, appelez toutes les vues nécessaires aux affichages (`view('students.index')` par exemple), et créez les fichiers de vues.

7. Création des models

1. Créez un fichier Model par table. Pensez bien à hériter de la classe `Db` en déclarant le model ainsi : `class Student extends Db { }`

2. Pour chaque Model :

- Déclarez la constante `TABLE_NAME`
- Déclarez des attributs `protected` pour chacun des champs de la table correspondante
- Déclarez vos setters
- Déclarez vos getters

Rappel **setters** :

```
public function setName($name) {
    $this->name = $name;
    return $name;
}
```

Rappel **getters** :

```
public function getName() {
    return $this->name;
}
```

3. Ajoutez les méthodes suivantes à tous vos models en les adaptant correctement pour chaque model :

Méthode `save()` : (sauvegarder un nouvel objet en bdd)

```
public function save() {
    $data = [
        "firstname" => $this->firstname(),
        "surname" => $this->surname()
    ];
    if ($this->id > 0) return $this->update();
    $nouvelId = Db::dbCreate(self::TABLE_NAME, $data);
    $this->setId($nouvelId);
    return $this;
}
```

Méthode `update()` : (mettre à jour l'objet en bdd)

```
public function update() {
    if ($this->id > 0) {
        $data = [
            "firstname" => $this->firstname(),
            "surname" => $this->surname()
        ];
        Db::dbUpdate(self::TABLE_NAME, $data);
        return $this;
    }
    return;
}
```

Méthode `delete()` : (supprimer l'objet de la bdd)

```
public function delete() {
    $data = [
        'id' => $this->id(),
    ];
    Db::dbDelete(self::TABLE_NAME, $data);
    return;
}
```

Méthode `findAll()` : (retrouver tous les éléments du Model)

```
public static function findAll() {
    $data = Db::dbFind(self::TABLE_NAME);
    return $data;
}
```

Méthode `find()` : (retrouver en fonction d'un array `$request`)

```
public static function find(array $request) {
```

```

    $data = Db::dbFind(self::TABLE_NAME, $request);
    return $data;
}

```

Méthode `findOne()` : (retrouver 1 élément en paramètre via son id)

```

public static function findOne(int $id) {
    $request = [
        ['id', '=', $id]
    ];
    $element = Db::dbFind(self::TABLE_NAME, $request);
    if (count($element) > 0) $element = $element[0];
    else return;

    return $element;
}

```

8. Mettez à jour les contrôleurs

1. Vos Model maintenant créés, appelez les données depuis les contrôleurs, notamment dans les méthodes correspondant à l'affichage de listes.

i. Par exemple : `$students = Student::findAll();`

2. Vous pouvez passer une variable `$students` à la vue de cette façon : `view('students.index', compact('students'));`

3. Mettez également à jour les méthodes correspondant à la création d'éléments en utilisant les méthodes des models, par exemple :

```

// Route POST student
public function save() {
    $student = new Student();
    $student->setName($_POST['name']);
    $student->setBirthdate($_POST['birthdate']);
    $student->save();
}

```

4. Si vous êtes sûrs que vos formulaires fonctionnent parfaitement, vous pouvez rediriger vers une autre page à l'issue du traitement du formulaire :

```

public function save() {
    $student = new Student();
    $student->setName($_POST['name']);
    $student->setBirthdate($_POST['birthdate']);
    $student->save();

    // On redirige vers la page d'un étudiant
    Header('Location: ' . url('/students/' . $student->getId()));
}

```

9. Mettez à jour les vues

1. Maintenant que les controllers passent des données aux vues, mettez à jour vos fichiers de vues afin d'afficher les données. Par exemple avec un foreach pour les listes.

10. Pages Update

1. Créez une route `edit` qui prendra aussi en paramètres l'ID de l'élément à modifier
2. Créez la méthode dans le controller, qui va appeler la vue de la page d'update et qui va passer à la page l'élément à modifier
3. Dans la vue, dans les champs `value`, mettez la valeur actuelle de l'élément à modifier.
4. Faites pointer le formulaire vers une méthode de contrôleur dédiée à l'update

Routes :

```

// Formulaire d'update
$router->get('student/{id}/edit', 'StudentsController@edit');

// Traitement de l'update
$router->post('student/{id}/edit', 'StudentsController@update');

```

Méthode affichant le formulaire:

```

class StudentController {
    public function edit($id) {
        $student = Student::findOne($id);
    }
}

```

```
        view('students.edit', compact('student'));
    }
}
```

Vue :

```
<form action=""
      <input type="text" name="firstname" value="= $student-&gt;getName() ?"
</form>
```

Méthode traitant le formulaire:

```
class StudentController {
    public function update($id) {
        $student = Student::findOne($id);
        $student->setFirstname($_POST['firstname']);
        $student->update();

        // On redirige vers la page de l'étudiant
        Header('Location: ' . url('students/' . $student->getId()));
    }
}
```

11. Suppression d'un enregistrement

Pour la suppression de l'enregistrement, il va falloir créer un lien ou un bouton qui redirige vers une méthode de contrôleur dédiée à cela :

Route vers laquelle pointe le bouton ou le lien de suppression :

```
$routes->get('students/{id}/delete', 'StudentsController@delete');
```

Controller :

```
class StudentsController {
    public function delete($id) {
        $student = Db::findOne($id);
        $student->delete();

        // On redirige vers la liste des étudiants
        Header('Location: ' . url('students'));
    }
}
```

12. Contrôles de saisie (validations)

Maintenant que les formulaires fonctionnent, tant pour créer que pour éditer, vous pouvez maintenant ajouter des validations dans les Model. Pour rappel, il faut ajouter les validations dans les `setters` : c'est en effet leur rôle ! Par exemple :

```
public function setFirstname($name) {

    // Liste des validations
    if ( strlen($name) < 3 ) {
        throw new Exception ('Le nom est trop court.');
    }

    // Si pas d'erreur :
    $this->name = $name;
    return $this;
}
```

13. Améliorez les models pour afficher les relations

L'idée est de pouvoir faire des choses comme ceci dans les vues :

```
Nom : <?= $student->firstname ?>
Liste des cours :

<?php foreach ($student->courses() as $course) : ?>

    Nom du cours : <?= $course->getTitle() ?>

<?php endforeach ?>
```

Ou bien l'inverse :

```

titre du cours : <?= $course->getTitre() ?>
Liste des étudiants :

<?php foreach ($course->students() as $student) : ?>

    Nom de l'élève : <?= $student->getFirstname() ?>

<?php endforeach ?>

```

C'est à dire que, depuis une vue concernant par exemple un étudiant, on aurait accès à ses cours ! C'est tout l'intérêt des relations, 1-1, 1-N ou N-N.

Voici comment procéder : on va utiliser une requête `INNER JOIN` écrite à la main dans le Model.

Fichier `Student.php` :

```

class Student extends Db {

    // ...

    public function courses() {

        // J'utilise getById de la classe Db qui me donne un pointeur PDO,
        $bdd = Db::getById();

        // Définition de la requête
        $req = "SELECT *
                FROM `student_course`
                INNER JOIN course ON course.id = student_course.id_course
                WHERE student_course.student_id = " . $this->getId();

        $res = $bdd->query($req);
        $courses = $res->fetchAll(PDO::FETCH_ASSOC);

        return $courses;
    }
}

```

Pensez bien à tester dans PHPMyAdmin vos requêtes !

14. Développements complémentaires

Alert en JS avant de supprimer

1. Ajoutez en Javascript une validation de suppression : lorsque l'on clique sur le lien ou le bouton de suppression, une `alert()` s'ouvre et nous demande de confirmer si on veut supprimer l'élément.

Requêtes SQL

Vous rédigerez les requêtes SQL suivantes, qui peuvent être utiles dans les développements futurs de l'application :

- Afficher le nombre d'étudiants.
- Afficher le nombre de cours.
- Afficher le nombre d'inscriptions.
- Afficher les cours n'ayant pas d'étudiants
- Afficher les étudiants n'ayant pas de cours
- Afficher les cours suivis par l'étudiant « Jean Dupont » (adaptez à vos données)
- Afficher tous les étudiants (même ceux qui n'ont pas de correspondance) ainsi que les cours
- Afficher les étudiants et tous les cours (même ceux qui n'ont pas de correspondance)
- Afficher tous les étudiants et tous les cours, peut importe les correspondances.

[02-faq.md](#)

Raw

PHP et MVC: FAQ

- 1. Rappels sur les models, views, controllers
- 2. Comparaison MVC et classique : comment créer des pages ?
- 3. Ajout des pages principales: liste, ajout, affichage
- 4. Ajouter un CSS ou un JS
- 5. Récupérer des données en jointures
- 6. Uploader et afficher les photos
- 7. Page show() pour afficher un élément

1. Rappels sur les models, views, controllers

En MVC, on a 3 groupes de fichiers : le but est de séparer notre code de façon prévisible par tous les développeurs, et prévisible par le développeur lui même qui se posera bien moins de question sur où se trouvent quels fichiers !

On a :

1. Les *Models* sont les fichiers qui gèrent la base de données: c'est une classe qui contient des setters, getters, et des fonctions d'enregistrement ou de lecture de la base de données.

i. Son nommage est le suivant: `Exemple.php` et `class Example extends Db { /* ... */ }`

2. Les *Views*, ce sont les **seuls** fichiers qui contiennent du HTML.

i. Leur nommage est le suivant :

a. Un dossier `examples` situé dans `public/views`

a. Des fichiers dedans, nommés toujours de la même manière quel que soit la table :

a. `examples/index.php`

b. `examples/add.php`

c. `examples/show.php`

3. Les *Controllers*, ce sont les "chefs d'orchestre" du MVC : lorsqu'un client demande une URL, le routeur pointe vers un controller. Le rôle du controller est de récupérer les données (en demandant au *Model*) et d'afficher les données (en demandant à la *View*).

i. Son nommage est le suivant: `ExamplesController.php` et `class ExamplesController { /* */ }`

Model : fichier en PascalCase, SINGULIER (`Example.php`)

Vues : dossier kebab-case, PLURIEL (`examples/`)

Controller : fichier en PascalCase, PLURIEL (`ExamplesController.php`)

2. Comparaison MVC et classique : comment créer des pages ?

Voici un équivalent en développement "classique" vs "MVC" :

Développement classique : `http://localhost/project/ajout-produit.php`

```
<?php

// On teste si on vient du formulaire en POST
if (!empty($_POST)) {

    // On teste les champs
    if ( strlen($_POST['title']) < 5) {
        throw new Exception ('titre trop court');
    }
    // Si oui, on enregistre en bdd les données du formulaire
    $bdd = new PDO();

    $request = "INSERT INTO products VALUES ...";
    $bdd->prepare($request);
    $bdd->execute([
        'title' => $_POST['title'],
        ...
    ]);
}

// On affiche le formulaire quand on arrive sur la page en GET
<html>
<body>
    <form action="ajout-produit.php" method="post">
        <input type="text" name="title" placeholder="nom produit">
    </form>
</body>
</html>
```

Dans cet exemple, on a beaucoup de choses de mélangées :

1. Le nom du fichier correspond à l'URL : si jamais le déplace le fichier, je dois changer tous les liens de mon site
2. On a du PHP qui vérifie si je viens sur la page en GET ou en POST
3. J'ai des validations de données
4. Si je suis en POST, j'ai du PHP qui enregistre en base de données
5. J'ai du HTML ensuite

Dans cet exemple, tout est mélangé ! C'est peut être très pratique pour des petits projets mais pour des projets plus gros (plus de 2 ou 3 tables) ce fonctionnement est impossible à maintenir car le code se répète de partout.

Développement MVC : `http://localhost/project/produits/add`

Router :

```
$routes->get('produits/add', 'ProduitsController@add');
$route->post('produits/add', 'ProduitsController@save');
```

Controller : `ProduitsController.php`

```
class ProduitsController {
    public function add() {
        view('produits.add');
    }

    public function save() {
        $produit = new Produit;
        $produit->setTitle($_POST['title']);
        $produit->save();
    }
}
```

Model : `Produit.php`

```
class Produit {
    private $id;
    private $title;

    public function setTitle($title) {
        if (strlen($title) > 5) {
            throw new Exception('titre trop long');
        }
        $this->title = $title;
        return $this;
    }

    public function save() {
        Db::dbSave($this);
    }
}
```

View : `produits/add.php`

```
<?php ob_start(); ?>

<form action="= url('produits/add') ?" method="post">
    <input type="text" name="title" placeholder="nom produit">
</form>

<?php $content = ob_get_clean(); view('template', compact('content')) ; ?>
```

Dans le projet en MVC, tout est séparé !

1. Je déclare mes URL : j'aurai une route en GET pour voir mon formulaire et une route en POST pour traiter mon formulaire.

Avantage par rapport au classique : je n'ai pas besoin de me souvenir des noms de fichiers, d'avoir des tas de fichiers éparpillés qui affichent quelque chose : tous les liens existants pour mon site sont répertoriés dans mes routes.

2. Je crée un contrôleur **PAR TABLE** : le rôle des contrôleurs est d'avoir des petites fonctions qui vont chapeauter chaque URL : cette url a besoin de données, cette url a besoin d'afficher ceci, cette url a besoin d'enregistrer ça...

Avantage par rapport au classique : toute la logique (traitement de données, récupération de données...) est déportée dans le controller : si j'ai un souci de PHP et que je suis dans le module "Produits" (listing, ajouts, page produit...), je sais que ça sera dans `ProduitsController.php`.

3. Je crée un Model **PAR TABLE** : le rôle du Model est de me permettre de faire des enregistrements en base de données et de récupérer des données. Ils sont un peu longs à écrire mais une fois écrits une bonne fois pour toute (et ils se ressemblent tous de toute façon), je peux en une ligne enregistrer ou lire quelque chose en bdd. De plus, c'est aussi son rôle de faire les validations avant d'enregistrer les données, grâce aux setters.

Avantage par rapport au classique : le code propre aux données reste dans des fichiers dédiés : les Models. Ainsi si j'ai une erreur de base de données ou de récupération de données, je sais que le problème vient de `Produit.php` !

1. Je crée mes vues: **UN DOSSIER PAR TABLE** : le rôle des vues est simplement d'avoir le HTML isolé dans un fichier dédié. Attention, ce n'est pas parce que ce sont des fichiers qui contiennent du HTML que ce sont des fichiers accessibles directement via un lien vers le fichier !! Ce sont juste des morceaux de fichiers, appellés par le contrôleur.

Avantage par rapport au classique : mon HTML est isolé de tout le reste, n'importe quel développeur ne connaissant pas PHP ou le MVC peut travailler dessus après quelques instructions.

3. Ajout des pages principales: liste, ajout, affichage

1. Créer les routes

Voici les routes principales à créer **par table** :

- une liste (index ou list)
- une page d'ajout (add)
- une route pour traiter l'ajout (save)
- une page pour afficher un élément (show)

Une route est construite ainsi : l'url attendue, le contrôleur et la fonction appelés.

routes.php

```
/**  
 * Pour plus de lisibilité sur les routes, on peut créer des routes nommées avec des slash :  
  
 * localhost/project/produits pour les produits  
 * localhost/project/produits/add pour ajouter un produit  
 * localhost/project/produits/23 pour afficher le produit 23  
 */  
  
$routes->get('produits', 'ProduitsController@list'); // Liste des éléments (on peut pointer sur la fonction ind  
$routes->get('produits/add', 'ProduitsController@add'); // Formulaire  
$routes->post('produits/add', 'ProduitsController@save'); // Traitement formulaire  
$routes->get('produits/{id}', 'ProduitsController@show'); // Afficher un élément, trouvé par son {id}
```

2. Créer le controller

Il s'agit du contrôleur et des fonctions appelés par les routes ci-dessus.

ProduitsController.php

```
class ProduitsController {  
  
    // Ça peut être index ou list, peu importe  
    public function list() {  
        $produits = Produit::findAll();  
  
        /**  
         * 1. On appelle le fichier `public/views/produits/list.php`  
         * 2. On passe à ce fichier la variable $produits  
  
         * Donc : view('produits.list') (dossier produits, fichier list.php)  
         * et compact('produits') (on passe la variable $produits)  
         */  
        view('produits.list', compact('produits'));  
    }  
  
    public function add() {  
  
        // Ici, on a juste à appeler la vue pour le moment :  
        view('produits.add');  
    }  
  
    public function save() {  
  
        // Ici, pas de vues à appeler ! Mais un produit à enregistrer.  
  
        $produit = new Produit;  
        $produit->setTitle($_POST['title']);  
        $produit->save();  
  
        // À l'issue de l'enregistrement, on redirige vers la liste des produits en appelant la fonction dédiée à l'affichage  
        ProduitsController::list();  
    }  
  
    // Cette fonction prend un id en paramètre: celui passé dans la route!  
    public function show($id) {  
  
        // Ici, on va chercher le produit puis l'envoyer à la vue qui affiche un produit.  
  
        $produit = Produit::findOne($id);  
  
        view('produits.show', compact('produit'));  
    }  
}
```

3. Créer le Model

Il s'agit du Model appelé par le contrôleur ci-dessus. Pour composer le model, inspirez vous du fichier `src/models/Example.php`.

Le Model contient :

- des Setters (insérer des données dans un objet Produit)
- des Getters (récupérer des données d'un Produit)
- des fonctions CRUD (save(), findAll(), findOne(\$id)…)

4. Créer les vues

Il s'agit des vues appelées par le contrôleur ci-dessus. Pour créer les vues, inspirez vous du dossier `public/views/examples`.

4. Ajouter un CSS ou un JS

5. Récupérer des données en jointures

6. Uploader et afficher les photos

7. Page show() pour afficher un élément



NaouiNassera commented on 2 Feb 2021

...

Thank you so much your course help me a lot



paintbox00 commented on 31 Mar 2021

...

Bonjour, serait-il possible d'avoir le correctif de l'exercice "Student" donné en exemple. J'ai du mal à voir à quoi sert la constante "TABLE_NAME" dans les classes.

Cela me permettrait également d'avoir une vue d'ensemble du projet.

Merci et bravo pour votre travail



tomsihap commented on 31 Mar 2021

Author ...

@paintbox00 les models, simplifiés ici, utilisent en guise de "manager" (<https://stackoverflow.com/questions/4737636/what-is-the-purpose-of-a-manager-class/13505275>) des méthodes statiques de la classe `Db` qui opère en base de données (pour réaliser les SELECT/INSERT/UPDATE/DELETE), à laquelle sont fournis le `TABLE_NAME`. Par exemple la méthode `save()` des models :

```
$nouvelId = Db::dbCreate(self::$TABLE_NAME, $data);
```

[Sign up for free](#)

to join this conversation on GitHub. Already have an account? [Sign in to comment](#)



© 2022 GitHub, Inc.

[Terms](#)

[Privacy](#)

[Security](#)

[Status](#)

[Docs](#)

[Contact GitHub](#)

[Pricing](#)

[API](#)

[Training](#)

[Blog](#)

[About](#)