

R3.01 - ProgWeb

TD 3 - POO+PDO+MVC

Avant propos

Application

Dans ce qui suit, nous allons parler fréquemment d'Application.

Une application est un programme ou un ensemble de programmes généralement accompagnés d'une interface utilisateur. MS Word, Filezilla, VSC en sont quelques exemples. Chrome et Firefox sont aussi des applications.

Les technologies Web sont de plus en plus utilisées pour développer des applications.

Même si un site Web n'est pas exactement une application, on va quand même le considérer ainsi car le navigateur est une application qui affiche des sites Web.

Ainsi, par extension, il n'est donc pas abusif de considérer le binôme contenant-contenu comme tel¹.

A partir de maintenant Application et Site Web sont considérés comme similaires et nous utiliserons donc le terme Application pour parler des deux.

En fin de séance, n'oubliez pas de sauvegarder vos fichiers créés ou modifiés avant de supprimer votre environnement de travail Docker !

Une explication à ce sujet est présentée un peu plus loin.

Bibliothèque et Framework

Une bibliothèque est une collection, un regroupement de fonctions utilitaires, qui appartiennent généralement à un domaine donné (bibliothèque mathématique, graphique, de crypto, de traitement du son, etc.). Ces fonctions sont à la disposition du développeur qui doit coder toute la logique de son application en appelant les fonctions des bibliothèques quand il en a besoin.

¹ Pour les plus curieux : VSC est une application utilisant le framework Electron qui tourne dans un Chrome remaillé pour gommer son aspect "Navigateur Web".

Un framework est un ensemble d'outils et de bibliothèques de fonctions qui permettent le développement rapide et structuré d'applications.

Dans le cas d'un framework, une partie de la mécanique de l'application est déjà contrôlée par le code intégré au framework. Celui-ci s'appuie à la fois sur les fonctions des bibliothèques et sur le code produit par le développeur.

Pour prendre une analogie simple : une bibliothèque s'apparente à une palette de briques et un framework serait plutôt une maison préfabriquée qu'on va pouvoir compléter et qui offre déjà un ensemble de fonctionnalités prêtes à l'emploi. Le framework apporte du ciment entre les briques.

Frameworks Web

Il existe de nombreux frameworks dans différents domaines et langages informatiques.

Dans le domaine du Web en particulier, il y a des frameworks pour le front et d'autres pour le back.

Il en existe de nombreux framework back comme Laravel, Symfony, CodeIgniter ou encore Slim. Il existe aussi des frameworks utilisant d'autres langages comme Django et Flask pour Python ou RoR (Ruby on Rails) pour Ruby.

Tous ces frameworks ont une trame commune et en maîtriser un permet d'en apprendre d'autres assez facilement, sous réserve de maîtriser le langage qui va avec !

Quel choix ?

Choisir un framework est souvent une question d'habitude de travail. On a appris et on maîtrise tel ou tel framework, ce qui nous pousse à nous laisser rouler dans le sens de la pente, sans se demander si notre choix est idéal pour le projet qu'on va démarrer.

L'humain est ainsi fait, c'est naturel.

Maîtriser un framework demande aussi beaucoup de temps et d'énergie car les framework sont souvent très fournis en fonctionnalités, dont certaines sont inutiles pour notre projet mais qui peuvent quand même être un passage obligé car la construction nécessite toutes les briques ou au moins les murs porteurs.

Nous vous proposons dans ce TD de ne faire aucun choix, et même de ne pas travailler avec un framework du marché, mais plutôt de mettre en place les éléments de base sur lesquels sont construits tous les frameworks, afin d'en comprendre la mécanique de base qui vous aidera plus tard à mieux appréhender les frameworks eux-mêmes.

Environnement de travail

Initialisation de l'environnement Dockerisé

Vous allez utiliser un environnement Dockerisé contenant un Apache+PHP v8.2, un SGBD MariaDB v11 et une application PHPMyAdmin v5.2.1 qui permet d'administrer votre BDD.

La configuration de cet environnement est très simple. Vous avez déjà expérimenté Docker en SAÉ 1.03 en BUT1.

Vous avez téléchargé un fichier **docker-compose.yml** depuis Moodle. Placez-le dans un dossier de travail (**TD3** par exemple).

Un fichier de paramétrage des variables d'environnement se trouve dans le même dossier. Il se nomme **.env**. Il est conseillé de le laisser intact mais si vous voulez le modifier sachez qu'il faut **IMPÉRATIVEMENT** le faire avant de lancer l'environnement Docker, après ce sera trop tard et tout changement ultérieur vous empêchera certainement de vous connecter à votre BDD !

Ouvrez ensuite un Terminal et déplacez-vous dans ce dossier de travail.

Toutes les commandes **docker** ou **docker-compose** que vous allez lancer doivent se faire dans un Terminal ouvert sur ce dossier **TD3** et nulle part ailleurs !



Si vous êtes chez vous ou sur votre propre ordinateur, vous devez modifier les lignes suivantes dans le **docker-compose.yml** :

- **image: r301-mariadb:11.0** devient **image: bigpapoo/r301-mariadb:11.0**
- **image: r301-php:8.2-apache** devient **image: bigpapoo/r301-php:8.2-apache**

Ensuite, et uniquement dans le cas des ordinateurs de l'IUT, vous devez télécharger les images Docker de cette façon :

```
docker image pull r301-mariadb:11.0
docker image pull r301-php:8.2-apache
docker image pull r301-phpmyadmin:5.2.1
```

Maintenant, pour tout le monde, exécutez ce qui suit. Note : **docker-compose** n'existe peut-être pas chez vous, utilisez alors **docker compose** (sans le tiret) :

```
docker-compose up -d && docker-compose logs -f
```

A ce stade, vous disposez d'un serveur Web Apache intégrant PHP et PDO pour MySQL/MariaDB. Testez dans un navigateur que **http://localhost:8000** affiche bien

quelque chose (il doit s'agir de l'affichage d'un **phpinfo()**) et vous pouvez vérifier la présence de PDO et de son driver pour MySQL).

Vous avez aussi accès à PHPMysqlAdmin pour administrer votre BDD. Cette application a été configurée de telle sorte que vous n'avez pas de login / mot de passe à saisir. C'est pour vous faciliter la tâche mais c'est évidemment un risque et en environnement de production, il ne faudrait pas procéder ainsi.

PHPMyAdmin est accessible sur l'URL **http://localhost:8001**. Testez l'accès.

Alternative locale à PHPMysqlAdmin

PHPMyAdmin présente l'avantage de tourner sur le serveur et d'éviter d'avoir à ouvrir le port de MariaDB vers l'extérieur. Cependant il n'est pas aussi agréable à utiliser que certains outils clients qu'on installe sur un ordinateur.

Puisque le "serveur" est ici votre propre machine, on a ouvert le port de connexion à MariaDB (**3306**) localement, le rendant donc accessible par un client local. Ce client va être installé par vos soins dans le paragraphe suivant (une extension MySQL dans VSC).

Pour les plus curieux, il est possible de faire de même à distance, en ouvrant le port MariaDB mais sans compromettre la sécurité, en utilisant un tunnel SSH. Ceci sort hélas du cadre de ce TD.


Accès au projet dans VSC

C'est le seul moyen de travailler sur votre projet car le conteneur et le volume qui contiendra les fichiers de votre projet ne sont pas accessibles par l'explorateur de fichiers.

Cependant, VSC possède la capacité de se connecter à un conteneur et d'y travailler comme vous le feriez localement ou à distance sur un serveur. Vous verrez que c'est extrêmement pratique. Vous l'avez déjà expérimenté en SAÉ 1.03 en BUT1.

Ouvrez une nouvelle fenêtre dans VSC et, si ce n'est pas déjà fait, installez les extensions :

- **Remote Development** de Microsoft (en fait, ça installe 4 extensions)
- **MySQL** de Weijan Chen

L'extension **Remote Development** ajoute un bouton  en bas-gauche de la fenêtre VSC.

Cliquez sur ce bouton pour établir une connexion avec le conteneur PHP et choisissez **Attacher au conteneur en cours d'exécution**, puis choisissez le conteneur PHP qui doit se nommer quelque chose ressemblant à **/xxx-myapp-1** où **xxx** est le nom du dossier dans lequel vous avez placé votre fichier **docker-compose.yml** issu de Moodle.

Si vous ne vous retrouvez pas ouvert sur le dossier de l'application, choisissez menu **Fichier > Ouvrir le dossier** et déplacez-vous dans l'arborescence (c'est le conteneur que vous voyez là) pour choisir **/var/www/**.

Et voilà ! Vous êtes dans votre conteneur Apache-PHP et vous pouvez manipuler les fichiers comme vous le feriez sur votre propre machine.

Sauf indication contraire, dans tout ce qui va suivre, nous supposons que vous êtes connecté de cette manière à votre conteneur via VSC et que vous voyez et travaillez sur les fichiers dans le conteneur.

En cas de doute, gardez un œil sur le bouton  qui affiche maintenant que vous êtes dans un conteneur. Si vous perdez cet affichage, soit vous êtes dans une autre fenêtre de VSC, soit le conteneur s'est arrêté et vous avez perdu la connexion. Dans ce dernier cas, vous aurez aussi un message très explicite de perte de connexion.

A partir d'ici, comme certaines commandes sont à lancer sur votre machine hôte et d'autres dans le conteneur PHP, vous allez trouver un texte dans la marge pour éviter de vous tromper :

[CONT] = Terminal du VSC attaché au conteneur PHP

[HOTE] = Terminal de votre machine ou Terminal VSC non attaché à un conteneur

Utilisation du module BDD de VSC

[HOTE] Si vous souhaitez l'utiliser, l'extension MySQL installée précédemment dans VSC a fait apparaître un bouton dans la barre latérale. Il mène vers l'espace **Database**.



Dans cet espace **Database**, en utilisant le bouton **+** situé en haut du panneau, vous pouvez créer une nouvelle connexion à votre BDD **r301_td3** qui tourne dans un conteneur, mais qui a un port de connexion ouvert sur **13306**. Renseignez ces informations pour établir une connexion :

- **Host** : 127.0.0.1
- **Port** : 13306
- **Username** : r301
- **Password** : zorgLub!22

et vérifiez que la connexion s'établit bien. A ce stade, la BDD doit être vide de toute table. Vous pouvez en créer une avec quelques enregistrements pour le test de sauvegarde qui va suivre.

Si vous avez décidé d'utiliser PHPMyAdmin, créez aussi cette table de test.

Sauvegarde et nettoyage

Rappel : votre projet se trouve dans un volume Docker qui est préservé quand vous arrêtez votre environnement (avec **docker-compose down**) mais qui n'est ni accessible

avec votre explorateur de fichiers, ni accessible si vous changez de machine car il est local à cette machine.

Voici comment sauvegarder vos données (code PHP et BDD) et nettoyer en fin de séance.

Sauvegarde

[CONT] Dans un Terminal intégré de VSC, ouvert sur votre conteneur, lancez ceci dans le Terminal intégré à VSC :

```
r301-backup
```

Après l’affichage de quelques lignes de logs dans le Terminal, vous trouverez, dans l’explorateur de fichiers de VSC ouvert du votre projet, un dossier **backup-r301/**.

Récupérez (clic droit puis **Télécharger...**) le contenu de ce dossier vers votre ordinateur. Les sauvegardes sont des fichiers nommés **r301-backup-<date_heure>.tgz**

Nettoyage

[HOTE] **Attention, le nettoyage vous fera perdre tous vos travaux non sauvegardés !**

Pour sauvegarder, lisez le paragraphe précédent.

Pour nettoyer vous devrez utiliser la commande : **docker-compose down -v**

Restauration d’un backup

Dans le conteneur PHP, placez à la racine dans **/var/www** une sauvegarde effectuée précédemment et nommez cette sauvegarde : **backup.tgz**.

[CONT] Dans un Terminal intégré de VSC, ouvert sur votre conteneur, après avoir déposé le **backup.tgz**, lancez ceci :

```
r301-restore
```

Votre projet actuel est écrasé par le projet restauré et la BDD est aussi reconstruite et peuplée de ses données. Le fichier **backup.tgz** est supprimé du conteneur en fin de restauration.

Introduction au Concept MVC

Objectif

Comprendre le besoin du modèle MVC dans le développement web, spécifiquement dans le cadre de PHP.

Le **MVC** est une architecture qui sépare une application en trois composants principaux : Modèle, Vue, et Contrôleur. C'est un peu comme une équipe de super-héros : chacun a ses super-pouvoirs, mais ensemble, ils sauvent le projet de l'anarchie totale.

En PHP, le MVC est souvent utilisé pour organiser le code de manière plus structurée et maintenable. L'utilisation de frameworks comme Laravel ou Symfony illustre bien cette séparation, mais il est essentiel de comprendre les bases avant de passer à ce type d'outils. Ici, nous n'utiliserons donc pas de framework existant, mais nous allons en construire un petit fait maison pour bien comprendre le fonctionnement du MVC.

Problématique

Pourquoi ne pas mélanger la logique d'affaires et la présentation ?

Code Spaghetti

En PHP, il est facile de mélanger du code HTML et du code PHP dans un même fichier. Cependant, cela crée du "code spaghetti", difficile à maintenir et à déboguer. Le MVC permet d'éviter ce mélange en séparant clairement les responsabilités. Imaginez essayer de manger des spaghettis avec une seule fourchette géante, c'est tout simplement le chaos !

Séparation des préoccupations

En séparant la logique d'affaires (calculs, interactions avec la base de données) de la présentation, on réduit la complexité du code et on facilite la collaboration entre développeurs backend et frontend. En gros, c'est comme séparer les garnitures sur une pizza : chacun sait ce qu'il doit manger sans embrouiller les autres.

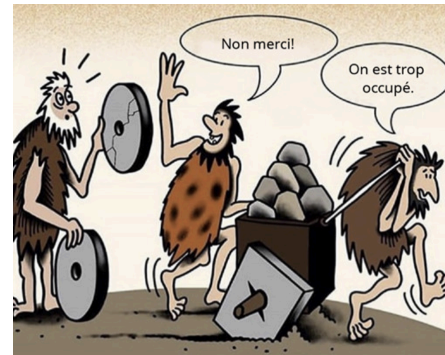
Avantages du MVC

Maintenabilité

Le fait de séparer la logique métier, la logique de contrôle, et la présentation rend le code plus facile à maintenir. Les modifications apportées à l'une des couches ont un impact minimal sur les autres.

Réutilisabilité

Les modèles peuvent être réutilisés dans différents contrôleurs, et les vues peuvent être réutilisées pour différentes actions, ce qui permet de limiter la redondance du code. Pourquoi réinventer la roue quand on peut simplement la réutiliser ?



Facilite les Tests Unitaires

En PHP, il est plus facile de tester chaque composant séparément. Par exemple, on peut tester un modèle indépendamment de la vue, ce qui améliore la qualité globale du code.

Présentation des Composants MVC

Le Modèle (Model)

Rôle : Gérer les données de l'application

Ce composant est responsable de la logique métier et de l'accès aux données. Le modèle interagit avec la base de données, effectue des requêtes SQL, et fournit des données structurées au contrôleur. En d'autres termes, il est la source de vérité de l'application.

Implémentation

Créer des classes PHP qui représentent des entités (ex. **User**, **Product**). Chaque classe est chargée de récupérer, insérer, mettre à jour ou supprimer les données dans la base de données. Le modèle est comme le scénariste d'un film : il s'assure que l'histoire soit bien construite, que tous les éléments soient en place, mais il ne s'occupe pas de la mise en scène ni de la présentation finale.

Exemple (voir code source dans **Mise en œuvre du MVC en PHP**)

La classe **User** est un exemple de modèle qui interagit avec une table **users** dans la base de données. Par exemple, elle peut avoir des méthodes comme **readAll()**, **create()**, **update()**, **delete()**. Ces méthodes sont responsables des opérations **CRUD** (Create, Read, Update, Delete) sur les données utilisateurs.

La Vue (View)

Rôle : Gérer la présentation des données à l'utilisateur

La vue est responsable de l'interface utilisateur, c'est-à-dire de la manière dont les données sont affichées. Elle reçoit des informations du contrôleur et les rend de manière agréable à l'utilisateur. C'est le côté esthétique de l'application.

Implémentation

Utiliser des fichiers PHP principalement pour générer du HTML. La vue ne doit contenir que peu de logique (voire aucune), car son rôle est uniquement de présenter les informations. On peut y inclure des structures conditionnelles basiques (par exemple, vérifier si une liste est vide), mais il est important de garder la vue aussi simple que possible.

Exemple (voir code source dans **Mise en œuvre du MVC en PHP**)

Un fichier de vue **user_list.php** reçoit une liste d'utilisateurs du contrôleur et les affiche dans un tableau HTML. La vue est comme le maquilleur d'un film : elle s'assure que tout soit joli, mais ne décide pas de l'intrigue. Le maquilleur ne décide pas de qui vit ou meurt dans le scénario, il s'assure juste que tout le monde soit présentable !

Le Contrôleur (Controller)

Rôle : Relier le Modèle et la Vue

Le contrôleur est le coordinateur principal : il reçoit les requêtes de l'utilisateur (généralement à travers l'URL), interagit avec le modèle pour obtenir ou manipuler des données, puis transmet ces données à la vue pour l'affichage. En d'autres termes, c'est l'intermédiaire entre la logique métier (le modèle) et la présentation (la vue).

Implémentation

Utiliser des classes pour traiter les requêtes (ex. **UserController** pour gérer les actions sur les utilisateurs). Le contrôleur prend des décisions en fonction des requêtes de l'utilisateur et des données renvoyées par le modèle. Le contrôleur est comme le réalisateur d'un film : il s'assure que chaque scène soit tournée correctement, en suivant le scénario du modèle et en donnant des instructions claires à l'équipe pour que tout fonctionne harmonieusement.

Exemple (voir code source dans **Mise en œuvre du MVC en PHP**)

UserController peut avoir des méthodes telles que **listUsers()**, **createUser()**, **deleteUser()**, etc. Par exemple, lorsqu'un utilisateur accède à une URL comme **/users**,

le contrôleur appelle le modèle pour récupérer la liste des utilisateurs, puis passe ces données à la vue appropriée pour les afficher.

Mise en Relation des Composants

Flux de données dans une application MVC

Requête utilisateur (par ex., URL) → Contrôleur

Le contrôleur reçoit la requête, comme un chef d'orchestre qui reçoit la partition de musique à jouer. Il doit alors interpréter cette partition et donner des instructions précises à chaque section de l'orchestre pour que tout soit parfaitement coordonné. De la même manière, le contrôleur envoie des requêtes au modèle pour récupérer les bonnes données, tout en s'assurant que chaque instrument (ou composant) joue son rôle au bon moment. Le contrôleur est donc le point de départ de toute l'interaction, garantissant que chaque action soit réalisée de manière harmonieuse et fluide.

Contrôleur → Modèle

Le **Contrôleur** appelle le **Modèle** pour obtenir des données. C'est comme si le chef d'orchestre demandait à chaque musicien de jouer sa partition, en sachant exactement quel instrument doit jouer à quel moment. Le modèle, ici, est le musicien qui connaît parfaitement la partition à jouer et sait quand intervenir. Le contrôleur s'assure que tout le monde est prêt et que les données sont disponibles au bon moment. Le modèle exécute les requêtes nécessaires pour fournir les informations correctes, comme un musicien qui joue sa note au moment parfait pour créer l'harmonie. Ensuite, le modèle renvoie les données brutes au contrôleur, un peu comme un musicien qui joue sa note avant de la laisser se fondre dans l'ensemble de la symphonie.

Modèle → Contrôleur

Le **Modèle** renvoie les données au **Contrôleur**. Le musicien joue sa partition avec précision, fournissant ainsi les éléments nécessaires que le chef d'orchestre peut ensuite synchroniser avec le reste de l'orchestre. Le contrôleur prend ces données brutes et les organise de manière à ce qu'elles soient prêtes pour la présentation, un peu comme un chef d'orchestre qui ajuste le timing et le volume de chaque section pour créer une performance harmonieuse. De même, le contrôleur s'assure que chaque partie de l'application est en phase, créant ainsi une expérience utilisateur cohérente et fluide.

Contrôleur → Vue

Le **Contrôleur** envoie ces données à la **Vue** pour l'affichage. C'est comme lorsque le chef d'orchestre fait en sorte que la musique soit bien perçue par le public. La vue est ce que

le public voit et entend, orchestrée de manière harmonieuse pour garantir une expérience agréable.

Mise en œuvre du MVC en PHP

Voici l'exemple pratique qui détaille les différentes parties qu'on vient de voir précédemment.

Note : il vous faudra créer une table **users** et la peupler à l'aide de PHPMYAdmin.

Structure du Projet

Créez une structure de dossier typique pour l'architecture MVC (le code source des fichiers **.php** est donné plus loin) :

```
/var/www/  
  config/  
    database.php  
  controllers/  
    user_controller.php  
  html/  
    index.php  
  models/  
    user.php  
  views/  
    user_list.php
```

Configuration de la Base de Données

Placement du fichier de configuration

Le fichier **database.php** est placé dans un dossier **/config/** qui se trouve ici idéalement **en dehors de la zone accessible par un navigateur web**. Cela signifie qu'il doit être placé en dehors du répertoire public (ici **/var/www/**) qui est directement accessible via l'URL. Si ce fichier est laissé dans une zone accessible, un utilisateur malintentionné pourrait directement y accéder via son navigateur, et obtenir des informations sensibles comme les identifiants de la base de données (nom d'utilisateur et mot de passe). Cela pourrait permettre une attaque où un hacker pourrait utiliser ces informations pour accéder à la base de données, modifier ou voler des données, voire compromettre toute l'application. En plaçant ce fichier hors de la portée publique, on ajoute une couche de sécurité essentielle pour protéger les informations confidentielles et réduire les risques d'attaques.

/config/database.php

```
<?php
class Database {
    // Informations de connexion à la base de données
    private $host = "mariadb";
    private $db_name = "r301_td3";
    private $username = "r301";
    private $password = "zorgLub!22";
    private $conn;

    // Méthode pour établir la connexion à la base de données
    public function getConnection() {
        $this->conn = null;
        try {
            // Création d'une nouvelle instance PDO pour se
            // connecter à la base de données
            $this->conn = new PDO("mysql:host=" . $this->host .
                                ";dbname=" . $this->db_name, $this->username,
                                $this->password);
            // Définir le mode d'erreur sur Exception pour une
            // meilleure gestion des erreurs
            $this->conn->setAttribute(PDO::ATTR_ERRMODE,
                                     PDO::ERRMODE_EXCEPTION);
        } catch (PDOException $exception) {
            // Gestion des erreurs de connexion
            echo "Connection error: " . $exception->getMessage();
        }
        return $this->conn;
    }
}
```

Création du Modèle (/models/user.php)

Représenter un utilisateur

```
<?php
class User {
    private $conn;
    private $table_name = "users";
    private $id;
    private $name;
    private $email;

    public function __construct($db) {
        $this->conn = $db;
    }
}
```

```

    }

    public function readAll() {
        $query = "SELECT id, name, email FROM " . $this->table_name;
        $stmt = $this->conn->prepare($query);
        $stmt->execute();
        $res = [];
        while ($row = $stmt->fetch(PDO::FETCH_ASSOC)) {
            $res[] = $row;
        }
        return $res;
    }
}

?>

```

Création du Contrôleur (/controllers/user_controller.php)

Gérer les actions sur les utilisateurs

```

<?php
require_once __DIR__ . '/../config/database.php';
require_once __DIR__ . '/../models/user.php';

class UserController {
    private $conn;

    public function __construct() {
        $database = new Database();
        $this->conn = $database->getConnection();
    }

    public function listUsers() {
        $user = new User($this->conn);
        $all_users = $user->readAll();
        return $all_users;
    }
}

?>

```

Création de la Vue (/views/user_list.php)

Afficher les utilisateurs

```

<?php if (!empty($_users_)) { ?>
    <table>

```

```

        <tr>
            <th>Code</th>
            <th>Nom</th>
            <th>Email</th>
        </tr>
<?php foreach ($_users_ as $user) { ?>
    <tr>
        <td><?php echo htmlentities($user['id']); ?></td>
        <td><?php echo htmlentities($user['name']); ?></td>
        <td><?php echo htmlentities($user['email']); ?></td>
    </tr>
<?php } ?>
</table>
<?php } else { ?>
    <p>Aucun utilisateur</p>
<?php } ?>

```

Point d'entrée (index.php)

Charger les données via le contrôleur et les passer à la vue

```

<?php
    require_once __DIR__ . '/../controllers/user_controller.php';

    $userController = new UserController();
    $_users_ = $userController->listUsers(); // Les '_' permettent de réduire
    le risque de variables ayant ce nom ailleurs dans le code...

    require_once __DIR__ . '/../views/user_list.php';
?>

```

Vous aurez noté la présence d'un **__DIR__** dans les codes sources précédents. Il s'agit d'une constante magique dont PHP a le secret. Il en existe d'autres. Vous trouverez une liste détaillée de chacune d'elles, en fin de TD.

Exercice : GlobeTales

Contexte

Vous êtes chargé de créer un mini-site où les utilisateurs peuvent poster des récits de leurs voyages et des informations pratiques. Chaque expérience est associée à un utilisateur, et nous souhaitons afficher les informations sur les expériences et leurs auteurs.

Étapes à suivre

Étape 1

Créez une base de données MySQL avec les tables suivantes

- **users** : **id**, **nom**, **pseudo**, **email**
- **trips** : **id**, **titre**, **description** (texte détaillé), **localisation** (texte, ex : ville, pays, lieu), **user_id**, **date_post**

Ici, chaque **trip** est associé à un **user** grâce à **user_id**.

Étape 2

Créez les Modèles **user.php** et **trip.php** :

user.php : ce modèle permet de gérer les utilisateurs. Le modèle doit permettre de :

- Récupérer tous les utilisateurs avec une méthode **getAllUsers(...²) : User[]**³
- Ajouter un utilisateur : **addUser(...) : User**
- Modifier un utilisateur : **alterUser(...) : User**
- Supprimer un utilisateur : **deleteUser(...) : bool**

trip.php : ce modèle permet de gérer les expériences de voyage. Le modèle doit permettre de :

- Récupérer les voyages postés par un utilisateur donné avec une méthode **getByUser(\$user_id) : Trip[]**
- Créer un voyage pour un utilisateur : **addTrip(...) : Trip**
- Modifier un voyage par son identifiant : **alterTrip(...) : Trip**
- Supprimer un voyage par son identifiant : **deleteTrip(...) : bool**

Étape 3

Créez les Contrôleurs **user_controller.php** et **trip_controller.php** :

- **user_controller.php** : ce contrôleur gère les actions sur les utilisateurs, comme afficher tous les utilisateurs, en ajouter un nouveau, etc.
- **trip_controller.php** : ce contrôleur gère les actions sur les voyages, comme lister tous les voyages, ajouter un nouveau voyage, afficher les détails d'un voyage particulier (**getTripsByUser(...)**), etc.

Étape 4

Créez les Vues **user_list.php** et **trip_list.php** :

- **user_list.php** : cette vue affiche une liste des utilisateurs et un lien pour voir les voyages de chaque utilisateur.

² A vous de déterminer les paramètres à passer aux méthodes demandées.

³ **User[]** signifie que ça renvoie un tableau d'objets **User**.

- **trip_list.php** : cette vue affiche la liste des voyages d'un utilisateur, leur titre, leur description, leur localisation, et la date de publication.

Étape 5

Connectez le tout dans un fichier d'index (**index.php**) et donnez la possibilité aux utilisateurs de naviguer entre les utilisateurs et les voyages. Par exemple, un utilisateur peut cliquer sur un voyage pour voir les détails.

Extension : Gestion de Compte

Objectif

Étendre l'application de partage d'expériences de voyage en ajoutant une fonctionnalité de gestion de compte utilisateur avec un système de connexion. Si un utilisateur est connecté, il peut poster et gérer ses propres récits de voyage. S'il n'est pas connecté, il ne peut que consulter les récits existants.

La préservation du statut de connexion se fera par une session PHP.

Étapes à suivre

Étape 1

Ajoutez une gestion des comptes dans la base de données MySQL. Modifiez la table **users** pour ajouter des colonnes **password** et **created_at**.

Étape 2

Créez le Modèle **auth.php** qui permet de gérer l'authentification des utilisateurs, avec des méthodes permettant de :

- Se connecter : **login(...)** : **bool**
- S'inscrire : **register(...)** : **bool**
- Se déconnecter : **logout(...)** : **bool**

Étape 3

Créez le Contrôleur **auth_controller.php** permettant de gérer les actions liées à l'authentification, comme :

- La connexion avec **login()**
- La déconnexion avec **logout()**
- L'inscription par **register()**

Utilisez des sessions PHP.

Étape 4

Créez les Vues **login.php**, **register.php**, et adaptez **trip_list.php** :

- **login.php** : cette vue permet aux utilisateurs de saisir leur email et mot de passe pour se connecter.
- **register.php** : cette vue permet aux nouveaux utilisateurs de créer un compte.

Modifier **trip_list.php** pour afficher des boutons d'édition et de suppression uniquement pour les récits de l'utilisateur connecté.

Étape 5

Adaptez le fichier d'index (**index.php**) pour :

- Le contrôle de session qui détermine si un utilisateur est connecté.
- Permettre aux utilisateurs connectés de créer, modifier, et supprimer leurs récits.

Fonctionnalité supplémentaire (optionnelle, à faire à la fin)

Ajoutez une fonctionnalité de "mot de passe oublié" pour permettre aux utilisateurs de réinitialiser leur mot de passe. Simuler l'envoi d'un email avec un lien vers une URL contenant un tag généré aléatoirement, stocké en BDD dans un champ approprié de la table users. A réception de cette URL, vérifiez que le tag correspond à celui stocké dans la table users et autorisez une remise à zéro du mot de passe dans ce cas.

Extension : Fonctionnalité de "Likes"

Objectif

Étendre l'application de partage d'expériences de voyage pour permettre aux visiteurs (qu'ils soient connectés ou non) de laisser des "likes" sur les récits de voyage. Cette fonctionnalité sera implémentée en utilisant un cookie client pour mémoriser les articles déjà likés, afin d'éviter le spam de likes.

Cookies

Les **cookies PHP** sont créés, à l'initiative du serveur, avec la fonction **setcookie()**, puis envoyés au navigateur, qui les conserve et les renvoie lors de toutes les requêtes suivantes.

À chaque fois que l'utilisateur navigue vers une page du même domaine, le navigateur envoie automatiquement les cookies correspondants dans l'en-tête de la requête HTTP. Cela permet au serveur ou aux scripts côté client de "se souvenir" des informations, telles que les préférences utilisateur ou l'état de connexion, tout au long de la session ou même au-delà, si le cookie a une date d'expiration future.

Exemple de code PHP pour définir un cookie. Dans le cookie on peut stocker du texte (par exemple des données sérialisées en remplaçant le “Alice” de l’exemple ci-après par un appel à **serialize()** avec les variables que vous voulez) :

```
<?php
    // Crée un cookie "user" avec la valeur "Alice" qui expire dans 7 jours
    setcookie("user", "Alice", time() + (86400 * 7), "/");
?>
```

En cas d'utilisation de **serialize()** il faudra **unserialize()** le cookie avant d'en faire usage. L'inconvénient de cette sérialisation est que le cookie est plus difficilement exploitable côté client (en JS) car la sérialisation est un format propre à PHP, même s'il est compréhensible et réversible en JS moyennant un peu d'effort de codage.

Une solution peut aussi être d'y stocker du JSON “stringifié”⁴.

Étapes à suivre

Étape 1

Ajoutez une gestion des likes dans la base de données MySQL en modifiant la table **trips** pour ajouter une colonne **likes** qui, par défaut, doit être initialisée à 0.

Étape 2

Mettez à jour le Modèle **trip.php** en ajoutant une méthode **addLike(...)** qui permet d'incrémenter le nombre de likes pour un voyage spécifique.

Étape 3

Créez le Contrôleur **like_controller.php** permettant de gérer les actions liées aux likes, c'est-à-dire l'ajout d'un like à un voyage et la suppression d'un like (une sorte de **unlike** qui sera vue la partie antispam plus loin)

Étape 4

Adaptez la Vue **trip_list.php** :

- Ajoutez un bouton (icône ?) “Like” à chaque récit de voyage, qui appelle le contrôleur pour ajouter un like lorsque l'utilisateur clique dessus.
- Affichez le nombre de likes pour chaque voyage.

Étape 5 - unlike

Gérez un **cookie côté client** :

⁴ **JSON.stringify()** et **JSON.parse()** en JS.

- Utilisez un cookie pour mémoriser les voyages qui ont été likés par un visiteur spécifique.
- Ajoutez une vérification pour voir si l'utilisateur a déjà liké un voyage en vérifiant le cookie servant d'anti-spam. La suppression d'un like ne doit être acceptée que si ce cookie atteste qu'un like a été placé. Une seule annulation possible ou aucun si pas de cookie. Gérer le cookie en fonction de l'action like / unlike.

Explication des Constantes Magiques en PHP

Les explications données dans ce chapitre pourront vous être très utiles sur des gros projets, tels que les SAÉ par exemple.

PHP propose plusieurs constantes spéciales, souvent appelées **Constantes Magiques**, qui commencent par deux underscores (`__`).

`__DIR__`

Description

`__DIR__` est une constante magique qui retourne le chemin absolu du répertoire dans lequel se trouve le fichier qui l'utilise.

Utilité

Elle est très pratique pour inclure des fichiers de manière fiable, surtout lorsqu'on travaille avec des structures de fichiers complexes. En utilisant `__DIR__`, vous pouvez toujours faire référence à un fichier relatif au script courant, sans vous soucier de la manière dont le script est appelé.

Exemple

```
| require_once __DIR__ . '/../config/database.php';
```

Ici, `__DIR__` est utilisé pour inclure le fichier **database.php**, qui se trouve dans le répertoire parent du fichier courant. Cela rend le code plus robuste et indépendant du chemin d'exécution, car il s'appuie sur le chemin du fichier actuel plutôt que sur l'endroit d'où le script est appelé.

__FILE__

Description

__FILE__ retourne le chemin complet et le nom du fichier actuel où cette constante est utilisée

Utilité

Elle est souvent utilisée pour obtenir le nom du fichier en cours d'exécution, utile pour des tâches de débogage ou de journalisation.

__LINE__

Description

__LINE__ retourne le numéro de la ligne actuelle dans le fichier où cette constante est utilisée.

Utilité

Très utile pour le débogage, car elle permet de savoir exactement à quelle ligne une instruction a été exécutée.

__FUNCTION__

Description

__FUNCTION__ retourne le nom de la fonction dans laquelle on se trouve au moment de son utilisation.

Utilité

Pratique pour la journalisation et le débogage, surtout pour savoir quelle fonction est en cours d'exécution.

__CLASS__

Description

__CLASS__ retourne le nom de la classe courante si elle est utilisée dans une méthode d'une classe.

Utilité

Utile dans les méthodes de classe pour identifier la classe en cours, surtout dans les grandes bases de code où plusieurs classes sont utilisées.

__METHOD__

Description

__METHOD__ retourne le nom de la méthode courante, incluant le nom de la classe.

Utilité

Semblable à **__FUNCTION__**, mais spécifique aux méthodes de classe.

Ces constantes magiques rendent le code plus dynamique et facilitent la maintenance en offrant des informations contextuelles sur le script en cours d'exécution. Utiliser **__DIR__** et d'autres constantes magiques est particulièrement utile lorsqu'il s'agit de référencer des fichiers ou de générer des logs détaillés pour le débogage.