Synthèse des apprentissages avant le TP noté

Le thème de ce développement est un blog. La plateforme de test est xampp sur Windows.

Cette application web full responsive mobile first utilise : la POO PHP8, Bootstrap 5, twig 3, MariaDB, Alpinejs, et s'appuie sur les patterns MVC et Singleton pour la connexion à la base de donnée , le logger d'erreur, la gestion des sessions.

1.Environnement et Structure (XAMPP & MVC)

Environnement (XAMPP)

1. **Installation :** Installez XAMPP.
2. **Configuration :** Lancez le panneau de contrôle XAMPP et démarrez Apache et MySQL.
3. **Base de données :** Allez sur `http://localhost/phpmyadmin` et créez votre base de données (par ex. `blog_db`) avec l'interclassement `utf8mb4_general_ci`.
4. **Emplacement du projet :** Placez votre projet dans le dossier `C:\xampp\htdocs\3A2526-blog`.

Structure MVC (Modèle-Vue-Contrôleur)

Voici une structure de dossiers classique pour ce type de projet :

```
/3A2526-blog
|
|-- /public                 <-- Seul dossier accessible depuis le web
|   |-- .htaccess           <-- (Pour la réécriture d'URL)
|   |-- index.php           <-- (Point d'entrée unique - Front Controller)
|   |-- /assets
|       |-- /css
|       |-- /js
|       |-- /images
|
|-- /app                    <-- Cœur de l'application
|   |-- /Controllers        <-- (Gère la logique de requête)
|   |-- /Models             <-- (Gère les données et la logique métier)
|   |-- /Views              <-- (Les templates Twig .twig)
|   |-- /Core               <-- (Classes de base : Router, Singletons...)
|
|-- /vendor                 <-- (Géré par Composer pour Twig, etc.)
|
|-- composer.json           <-- (Fichier de configuration Composer)
```

2.Les Singletons (Connexion BDD, Logger, Session)

Le pattern **Singleton** garantit qu'une classe n'a qu'une seule instance et fournit un point d'accès global à cette instance.

Singleton pour la Connexion BDD (MariaDB avec PDO)

Création une classe Database dans /app/Core.

```php
<?php
// Fichier : /app/Core/Database.php
namespace App\Core;

use PDO;
use PDOException;

class Database {
    // L'instance unique
    private static ?self $instance = null;

    // L'objet PDO de connexion
    private PDO $connection;

    // Informations de connexion
    private string $host = 'localhost';
    private string $db_name = 'blog_db';
    private string $username = 'root'; // Par défaut sur XAMPP
    private string $password = '';     // Par défaut sur XAMPP

    // Le constructeur est privé pour empêcher l'instanciation directe
    private function __construct() {
        try {
            $dsn = "mysql:host={$this->host};dbname={$this->db_name};charset=utf8mb4";
            $this->connection = new PDO($dsn, $this->username, $this->password);
            $this->connection->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
            $this->connection->setAttribute(PDO::ATTR_DEFAULT_FETCH_MODE, PDO::FETCH_OBJ);
        } catch (PDOException $e) {
            // En production, logguer l'erreur plutôt que de l'afficher
            die('Erreur de connexion : ' . $e->getMessage());
        }
    }

    // La méthode statique qui crée ou retourne l'instance
    public static function getInstance(): self {
        if (self::$instance === null) {
            self::$instance = new self();
        }
        return self::$instance;
    }

    // Méthode pour récupérer l'objet PDO
    public function getConnection(): PDO {
        return $this->connection;
    }

    // Empêcher le clonage
    private function __clone() {}
```

```
    // Empêcher la désérialisation
    public function __wakeup() {
        throw new \Exception("Cannot unserialize a singleton.");
    }
}
```

Utilisation dans un Modèle :

Les modèles (ex: PostModel) utiliseront cette instance pour communiquer avec la BDD.

```
// Fichier : /app/Models/PostModel.php
namespace App\Models;

use App.Core\Database;
use PDO;

class PostModel {
    private PDO $db;

    public function __construct() {
        // On récupère l'instance unique de la connexion BDD
        $this->db = Database::getInstance()->getConnection();
    }

    public function findAll(): array {
        $stmt = $this->db->query("SELECT * FROM posts ORDER BY created_at DESC");
        return $stmt->fetchAll();
    }
}
```

Ce même pattern (constructeur privé, méthode `getInstance()`) sera appliquer pour le **Logger** (qui écrira dans un fichier `error.log`) et le **Gestionnaire de Session** (qui encapsulera les fonctions `session_start()`, `$_SESSION[]`, etc.).

3.Intégration (Composer, Twig, Routing)

1. Composer et Autoloading (PSR-4)

Pour que vos classes (Contrôleurs, Modèles, Singletons) soient chargées automatiquement, vous avez besoin de Composer.

1. Créez un fichier `composer.json` à la racine de `mon_blog` :

```
{
    "name": "epul/3a2526-blog",
    "description": "Un blog moderne en PHP 8 POO",
    "require": {
        "twig/twig": "^3.0"     },
    "autoload": {
        "psr-4": {
            "App\\": "app/"         }
    }
}
```

2. Ouvrez un terminal dans ce dossier et lancez `composer install`.
3. Incluez l'autoloader dans votre point d'entrée `public/index.php` : `require_once '../vendor/autoload.php';`

2. Le Routeur (Le Cœur du MVC)

Dans public/index.php, vous avez besoin d'un routeur simple pour diriger les requêtes (/, /post/123, /admin) vers le bon Contrôleur et la bonne méthode. C'est le "Front Controller".

```php
<?php
// Fichier : /public/index.php

// 1. Charger l'autoloader de Composer
require_once '../vendor/autoload.php';

// 2. Utiliser vos classes (Singletons, Contrôleurs...)
use App\Controllers\HomeController;
use App\Controllers\PostController;
// ...et votre Singleton de Session
// App\Core\SessionManager::getInstance()->start();


// 3. Routage simple (à améliorer)
$request_uri = $_SERVER['REQUEST_URI'];
$base_path = '/mon_blog'; // Le sous-dossier dans htdocs

// Enlever le chemin de base de l'URI
$route = str_replace($base_path, '', $request_uri);
$route = parse_url($route, PHP_URL_PATH); // Enlever les query params

switch ($route) {
    case '/':
        (new HomeController())->index();
        break;

    case (preg_match('/^\/post\/(\d+)$/', $route, $matches) ? true : false):
        // $matches[1] contient l'ID du post
        (new PostController())->show($matches[1]);
        break;
```

```
    default:
        http_response_code(404);
        (new HomeController())->error404();
        break;
}
```

3. Twig (Les Vues)

Les contrôleurs n'afficheront pas de HTML. Ils chargeront et rendront un template Twig.

```
<?php
// Fichier : /app/Controllers/HomeController.php
namespace App\Controllers;

use App\Models\PostModel;
use Twig\Environment;
use Twig\Loader\FilesystemLoader;

class HomeController {

    private Environment $twig;
    private PostModel $postModel;

    public function __construct() {
        // Initialiser Twig
        $loader = new FilesystemLoader('../app/Views');
        $this->twig = new Environment($loader, [
             // 'cache' => '../cache/twig', // Activer pour la production
        ]);

        $this->postModel = new PostModel();
    }

    public function index(): void {
        // 1. Récupérer les données (via le Modèle)
        $posts = $this->postModel->findAll();

        // 2. Rendre la Vue (Twig)
        echo $this->twig->render('home.twig', [
            'page_title' => 'Accueil du Blog',
            'posts' => $posts
        ]);
    }
}
```

4.Frontend (Bootstrap 5, Alpine.js, Mobile-First)

C'est ici que Twig, Bootstrap et Alpine.js se rencontrent.

1. Template de Base (Twig)

Créez un fichier base.twig que les autres templates étendront.

```
<!doctype html>
<html lang="fr">
<head>
    <meta charset="utf-C-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>{{ page_title|default('Mon Blog') }}</title>

    <link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css"
rel="stylesheet">

    <script defer
src="https://cdn.jsdelivr.net/npm/alpinejs@3.x.x/dist/cdn.min.js"></script>
</head>
<body>

    <nav class="navbar navbar-expand-lg navbar-dark bg-dark">
        </nav>

    <div class="container mt-4">
        {% block content %}{% endblock %}
    </div>

    <footer class="text-center mt-5 py-3 bg-light">
        © {{ "now"|date("Y") }} Mon Blog
    </footer>

    <script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/js/bootstrap.bundle.min.js"
></script>
</body>
</html>
```

2. Page d'Accueil (Twig)

La vue home.twig hérite de base.twig.

```
{% extends "base.twig" %}

{% block content %}
    <h1 class="mb-4">{{ page_title }}</h1>

    <div x-data="{ open: false }">
        <button @click="open = !open" class="btn btn-primary mb-3">
            Afficher/Masquer le message
        </button>
        <div x-show="open" x-transition class="alert alert-info">
            Ceci est un blog génial !
        </div>
```

```
    </div>

    <div class="row">
        {% for post in posts %}
            <div class="col-12 col-md-6 col-lg-4 mb-4">
                <div class="card h-100">
                    <div class="card-body">
                        <h5 class="card-title">{{ post.title }}</h5>
                        <p class="card-text">{{ post.content|slice(0, 100)
}}...</p>
                        <a href="/mon_blog/post/{{ post.id }}" class="btn btn-
secondary">Lire la suite</a>
                    </div>
                </div>
            </div>
        {% else %}
            <p>Aucun article à afficher.</p>
        {% endfor %}
    </div>
{% endblock %}
```

Cette structure donne un contrôle total sur l'application, sépare clairement les responsabilités (PHP/logique vs HTML/présentation) et utilise les patterns.

1.Configuration du `.htaccess` (pour le Front Controller)

Ce fichier est crucial pour le pattern MVC. Son rôle est d'intercepter **toutes** les requêtes HTTP (comme `/post/123`) et de les rediriger vers votre point d'entrée unique (`index.php`), qui se chargera ensuite d'appeler le bon routeur et le bon contrôleur.

Ce fichier doit être placé dans le dossier public : `/3A2526-Blog/public/.htaccess`

```
# Activer le moteur de réécriture d'URL
RewriteEngine On

# Définir le chemin de base pour la réécriture
# Ceci est essentiel car le projet est dans un sous-dossier de XAMPP
RewriteBase /mon_blog/public/

# Condition 1 : Ne pas réécrire si la requête est pour un fichier qui existe
RewriteCond %{REQUEST_FILENAME} !-f

# Condition 2 : Ne pas réécrire si la requête est pour un dossier qui existe
RewriteCond %{REQUEST_FILENAME} !-d

# Règle : Rediriger tout le reste vers index.php
# L'URL demandée (ex: "post/123") sera passée en paramètre GET "url"
# [L] = Dernière règle / [QSA] = Query String Append (conserve les ?page=2 etc.)
RewriteRule ^(.*)$ index.php?url=$1 [L,QSA]
```

Adaptation du code contenu dans `public/index.php`

Le routeur dans `public/index.php` doit maintenant lire ce paramètre `url` au lieu de `$_SERVER['REQUEST_URI']` :

```php
<?php
// Fichier : /public/index.php

require_once '../vendor/autoload.php';

// Définir le chemin pour les logs (voir section 3)
define('LOG_PATH', dirname(__DIR__) . '/logs');

use App\Controllers\HomeController;
use App\Controllers\PostController;

// Récupérer l'URL "propre" depuis le .htaccess
// On nettoie le slash de fin s'il existe
$url = $_GET['url'] ?? '/';
$url = rtrim($url, '/');

// Routage simple basé sur le paramètre 'url'
switch ($url) {
    case '':
    case '/':
        (new HomeController())->index();
        break;

    // preg_match vérifie si l'URL correspond au format "post/un_nombre"
    case (preg_match('/^post\/(\d+)$/', $url, $matches) ? true : false):
        // $matches[1] contient l'ID (le \d+)
        (new PostController())->show($matches[1]);
        break;

    default:
        http_response_code(404);
        (new HomeController())->error404();
        break;
}
```

2.Singleton pour la Gestion des Sessions (`SessionManager`)

Ce Singleton s'assure que `session_start()` n'est appelé qu'une seule fois et fournit une interface propre pour manipuler `$_SESSION`.

Il doit être placer dans `/app/Core/SessionManager.php`

```php
<?php
// Fichier : /app/Core/SessionManager.php
namespace App\Core;

class SessionManager {

    private static ?self $instance = null;

    // Le constructeur est privé
    // Il démarre la session uniquement si elle n'est pas déjà active
```

```php
    private function __construct() {
        if (session_status() === PHP_SESSION_NONE) {
            session_start();
        }
    }

    /**
     * Récupère l'instance unique du gestionnaire de session.
     */
    public static function getInstance(): self {
        if (self::$instance === null) {
            self::$instance = new self();
        }
        return self::$instance;
    }

    /**
     * Définit une valeur dans la session.
     *
     * @param string $key La clé
     * @param mixed $value La valeur
     */
    public function set(string $key, mixed $value): void {
        $_SESSION[$key] = $value;
    }

    /**
     * Récupère une valeur de la session.
     *
     * @param string $key La clé
     * @param mixed $default La valeur par défaut si la clé n'existe pas
     * @return mixed
     */
    public function get(string $key, mixed $default = null): mixed {
        return $_SESSION[$key] ?? $default;
    }

    /**
     * Vérifie si une clé existe dans la session.
     */
    public function has(string $key): bool {
        return isset($_SESSION[$key]);
    }

    /**
     * Supprime une clé de la session.
     */
    public function remove(string $key): void {
        if ($this->has($key)) {
            unset($_SESSION[$key]);
        }
    }

    /**
     * Détruit complètement la session (ex: déconnexion).
     */
    public function destroy(): void {
        // Détruit toutes les variables de session
        $_SESSION = [];
```

```
        // Si vous souhaitez détruire le cookie de session aussi
        if (ini_get("session.use_cookies")) {
            $params = session_get_cookie_params();
            setcookie(session_name(), '', time() - 42000,
                $params["path"], $params["domain"],
                $params["secure"], $params["httponly"]
            );
        }

        session_destroy();
        // Recrée une instance pour pouvoir continuer à utiliser le site
        // (ex: définir un message flash "Vous êtes déconnecté")
        self::$instance = new self();
    }

    // Empêcher le clonage
    private function __clone() {}

    // Empêcher la désérialisation
    public function __wakeup() {
        throw new \Exception("Cannot unserialize a singleton.");
    }
}
```

Exemple d'utilisation dans un contrôleur :

```
use App\Core\SessionManager;

// Démarrer ou récupérer la session
$session = SessionManager::getInstance();

// Définir un message flash
$session->set('flash_message', 'Article créé avec succès !');

// Récupérer l'ID de l'utilisateur connecté
$userId = $session->get('user_id');

// Déconnexion
// $session->destroy();
```

Singleton pour le Logger d'Erreurs

Ce Singleton écrira les erreurs dans un fichier `app.log` (créer par nos soins dans le dossier `/logs` à la racine de du projet).

Placez-le dans `/app/Core/Logger.php`

```
<?php
// Fichier : /app/Core/Logger.php
namespace App\Core;

class Logger {
```

```php
    private static ?self $instance = null;
    private $logFile; // Ressource de fichier

    // Le constructeur est privé
    private function __construct() {
        // Utilise la constante définie dans index.php
        $logDir = LOG_PATH;

        if (!is_dir($logDir)) {
            mkdir($logDir, 0755, true);
        }

        $logFilePath = $logDir . '/app.log';

        // Ouvre le fichier en mode "append" (ajout)
        // @ supprime l'erreur PHP si le fichier n'est pas inscriptible,
        // nous le gérons manuellement
        $this->logFile = @fopen($logFilePath, 'a');

        if (!$this->logFile) {
            // En cas d'échec d'ouverture, on lève une exception
            // (le dossier /logs n'est peut-être pas inscriptible)
            throw new \Exception("Impossible d'ouvrir le fichier de log :
$logFilePath");
        }
    }

    /**
     * Récupère l'instance unique du Logger.
     */
    public static function getInstance(): self {
        if (self::$instance === null) {
            self::$instance = new self();
        }
        return self::$instance;
    }

    /**
     * Écrit un message dans le fichier de log.
     *
     * @param string $level (ex: INFO, WARNING, ERROR)
     * @param string $message Le message à logger
     */
    public function log(string $level, string $message): void {
        if (!$this->logFile) {
            return; // N'essaie pas d'écrire si le fichier n'a pas pu être ouvert
        }

        $date = (new \DateTime())->format('Y-m-d H:i:s');
        $formattedMessage = "[$date] [$level] $message" . PHP_EOL; // PHP_EOL =
saut de ligne

        // Écriture atomique (verrouillage exclusif)
        fwrite($this->logFile, $formattedMessage);
    }

    // --- Méthodes de convenance ---

    public function info(string $message): void {
```

```
        $this->log('INFO', $message);
    }

    public function warning(string $message): void {
        $this->log('WARNING', $message);
    }

    public function error(string $message, \Throwable $exception = null): void {
        if ($exception) {
             $message .= " | Exception: " . $exception->getMessage() . " in " .
$exception->getFile() . " on line " . $exception->getLine();
        }
        $this->log('ERROR', $message);
    }


    // Le destructeur ferme le fichier lorsque le script se termine
    public function __destruct() {
        if ($this->logFile) {
            fclose($this->logFile);
        }
    }

    // Empêcher le clonage et la désérialisation
    private function __clone() {}
    public function __wakeup() {
        throw new \Exception("Cannot unserialize a singleton.");
    }
}
```

Exemple d'utilisation (dans un `try...catch` de votre `Database.php` par ex.) :

```
// Dans la classe Database (ou ailleurs)
use App\Core\Logger;

// ...
try {
    $this->connection = new PDO($dsn, $this->username, $this->password);
    // ...
} catch (PDOException $e) {
    // Utiliser le singleton Logger au lieu de 'die()'
    $logger = Logger::getInstance();
    $logger->error("Échec de la connexion à la BDD", $e);

    // Afficher un message générique à l'utilisateur
    die("Une erreur critique est survenue. Veuillez contacter l'administrateur.");
}
```

Éviter la répétition de code dans tous vos contrôleurs et modèles ?

L'utilisation de classes de base (l'héritage) va nous permettre de la logique commune et respecter le principe DRY (Don't Repeat Yourself).

Voici comment implémenter un `BaseController` et un `BaseModel`.

Le BaseController (pour Twig et les Services)

Pourquoi ?

Au lieu d'initialiser Twig, le `SessionManager` et le `Logger` dans *chaque* constructeur de *chaque* contrôleur, nous le faisons une seule fois dans le `BaseController`. Tous les autres contrôleurs hériteront de celui-ci.

Implémentation (`/app/Core/BaseController.php`)

Créez ce nouveau fichier. Ce sera une classe `abstract` car elle n'est pas destinée à être instanciée directement.

```php
<?php
// Fichier : /app/Core/BaseController.php
namespace App\Core;

use Twig\Environment;
use Twig\Loader\FilesystemLoader;

abstract class BaseController {

    protected Environment $twig;
    protected SessionManager $session;
    protected Logger $logger;

    public function __construct() {
        // 1. Initialiser Twig
        $loader = new FilesystemLoader(dirname(__DIR__) . '/Views');
        $this->twig = new Environment($loader, [
            // 'cache' => dirname(__DIR__) . '/cache/twig', // Pour la prod
        ]);

        // 2. Récupérer les instances des Singletons
        $this->session = SessionManager::getInstance();
        $this->logger = Logger::getInstance();

        // Bonus : Rendre la session accessible dans tous les templates Twig
        $this->twig->addGlobal('session', $this->session);
    }

    /**
     * Méthode d'aide pour rendre une vue Twig.
     *
     * @param string $template Le nom du fichier .twig
     * @param array $context Les données à passer au template
     */
    protected function render(string $template, array $context = []): void {
        try {
```

```
            echo $this->twig->render($template, $context);
        } catch (\Exception $e) {
            // En cas d'erreur de Twig (ex: template non trouvé)
            $this->logger->error("Erreur de rendu Twig", $e);
            // Afficher une page d'erreur générique
            http_response_code(500);
            echo "Une erreur est survenue lors du rendu de la page.";
        }
    }
}
```

Exemple d'utilisation (`/app/Controllers/HomeController.php`)

Le `HomeController` devient beaucoup plus simple.

```
<?php
// Fichier : /app/Controllers/HomeController.php
namespace App\Controllers;

use App\Core\BaseController; // <-- Importer la classe de base
use App\Models\PostModel;

// HomeController HÉRITE de BaseController
class HomeController extends BaseController {

    private PostModel $postModel;

    public function __construct() {
        // TRÈS IMPORTANT : Appeler le constructeur parent
        parent::__construct();

        // Initialiser uniquement ce qui est spécifique à ce contrôleur
        $this->postModel = new PostModel();
    }

    public function index(): void {
        // 1. Récupérer les données
        $posts = $this->postModel->findAll();

        // 2. Utiliser la méthode render() héritée du parent
        $this->render('home.twig', [
            'page_title' => 'Accueil du Blog',
            'posts' => $posts
        ]);
    }

    public function error404(): void {
        // $this->logger est disponible !
        $this->logger->warning("Page non trouvée (404) : " .
$_SERVER['REQUEST_URI']);

        http_response_code(404);
        $this->render('errors/404.twig', [
            'page_title' => 'Page non trouvée'
        ]);
    }
```

```
}
```

À noter : L'appel à `parent::__construct();` est crucial. Il exécute le constructeur du `BaseController`, ce qui initialise `$this->twig`, `$this->session`, et `$this->logger`.

Le `BaseModel` (pour la connexion BDD)

Pourquoi ?

De la même manière, au lieu de récupérer l'instance de la BDD et le Logger dans chaque modèle, nous le faisons dans un `BaseModel`.

Implémentation (`/app/Core/BaseModel.php`)

Créez ce fichier. Lui aussi sera `abstract`.

```php
<?php
// Fichier : /app/Core/BaseModel.php
namespace App\Core;

use PDO;

abstract class BaseModel {

    protected PDO $db;
    protected Logger $logger;

    public function __construct() {
        // Récupérer les instances des Singletons
        $this->db = Database::getInstance()->getConnection();
        $this->logger = Logger::getInstance();
    }

    // Vous pourriez même ajouter des méthodes communes ici,
    // par exemple une méthode find($id) générique,
    // mais pour l'instant, gardons simple.
}
```

Exemple d'utilisation (`/app/Models/PostModel.php`)

Votre `PostModel` est maintenant incroyablement épuré.

```php
<?php
// Fichier : /app/Models/PostModel.php
namespace App\Models;

use App\Core\BaseModel; // <-- Importer la classe de base
use PDOException;

// PostModel HÉRITE de BaseModel
class PostModel extends BaseModel {
```

```php
    // Pas besoin de __construct() ici !
    // Le constructeur de BaseModel est appelé automatiquement
    // s'il n'y a pas de constructeur défini dans la classe enfant.
    // $this->db et $this->logger sont donc immédiatement disponibles.

    /**
     * Récupère tous les articles de blog.
     */
    public function findAll(): array {
        try {
            // Utiliser $this->db (qui vient de BaseModel)
            $stmt = $this->db->query("SELECT * FROM posts ORDER BY created_at
DESC");
            return $stmt->fetchAll();

        } catch (PDOException $e) {
            // Utiliser $this->logger (qui vient de BaseModel)
            $this->logger->error("Erreur lors de la récupération de tous les
posts", $e);
            return []; // Toujours retourner un type de données cohérent
        }
    }

    /**
     * Récupère un article par son ID.
     */
    public function findById(int $id): object|false {
        try {
            $stmt = $this->db->prepare("SELECT * FROM posts WHERE id = ?");
            $stmt->execute([$id]);
            return $stmt->fetch(); // Retourne un objet ou false s'il n'est pas
trouvé

        } catch (PDOException $e) {
            $this->logger->error("Erreur lors de la récupération du post ID $id",
$e);
            return false;
        }
    }
}
```

Le résultat :

- Los **Contrôleurs** se concentrent sur la gestion des requêtes et le rendu des vues.
- Los **Modèles** se concentrent sur la logique métier et les requêtes SQL.
- Le dossier `/app/Core` contient toute la logique de "plomberie" (Singletons, classes de base) qui fait fonctionner l'application.

Ce découplage rend l'application beaucoup plus facile à maintenir et à tester.