Architecture MVC en PHP*

Partie 3 : Passez à une architecture MVC orientée objet

Chapitres 3 à fin

1 Préparation de l'environnement de travail

1.1 Dépôt Git

Vous allez continuer à travailler avec votre dépôt local Git créé lors de la séance précédente dans le répertoire `~\UwAmp\www\php-mvc`. Pour ce faire :

1. Placez-vous donc dans le dossier `mvc-php` et ouvrez une invite de commande Git Bash ou Windows PowerShell.
2. Décompressez le contenu du fichier `mvc-php-base-tp5.zip` directement à la racine de votre dossier `mvc-php` en acceptant de remplacer les fichiers existant avec ces nouveaux fichiers.

1.2 Base de données

Nous réutilisons toujours la même base de données MySQL *blog*. Les identifiants de connexion à la base de données utilisés dans le code sont *root root*. Changez-les si les vôtres sont différents.

Créez une nouvelle version de votre projet avec l'enchaînement de commandes ci-après :

```
> git add --all
> git commit -m "Base pour le TP5"
> git push -u origin main
```

2 Tirez parti de la composition

On a réduit le nombre de connexions à la base de données à une connexion par instance de `PostRepository`. C'était nécessaire, mais ce n'est pas encore suffisant. Par exemple, sur la page d'affichage d'un billet, on a une connexion pour les billets et une autre pour les commentaires.

*Mathieu Nebra

Dans un premier temps, nous allons continuer à travailler sur la classe `PostRepository` pour faire en sorte que les deux modèles partagent la même connexion à la base de données.
Une fois qu'on aura la possibilité de créer deux instances de cette classe, et que celles-ci partageront la même connexion PDO, alors vous appliquerez les changements au code modèle des commentaires.

2.1 Initiez-vous à la composition

Commençons par relire le code que nous avions dans `src/model/post.php` :

```php
<?php

class Post
{
        public string $title;
        public string $frenchCreationDate;
        public string $content;
        public string $identifier;
}

class PostRepository
{
        public ?PDO $database = null;

        public function getPost(string $identifier): Post
        {
            $this->dbConnect();
            $statement = $this->database->prepare(
                "SELECT id, title, content, DATE_FORMAT(creation_date,
                '%d/%m/%Y à %Hh%imin%ss') AS french_creation_date
                FROM posts WHERE id = ?"
            );
            $statement->execute([$identifier]);

            $row = $statement->fetch();
            $post = new Post();
            $post->title = $row['title'];
            $post->frenchCreationDate = $row['french_creation_date'];
            $post->content = $row['content'];
            $post->identifier = $row['id'];

            return $post;
        }

        public function getPosts(): array
        {
            $this->dbConnect();
            $statement = $this->database->query(
```

```
                "SELECT id, title, content, DATE_FORMAT(creation_date,
                '%d/%m/%Y à %Hh%imin%ss') AS french_creation_date FROM posts
                ORDER BY creation_date DESC LIMIT 0, 5"
            );
            $posts = [];
            while (($row = $statement->fetch())) {
                $post = new Post();
                $post->title = $row['title'];
                $post->frenchCreationDate = $row['french_creation_date'];
                $post->content = $row['content'];
                $post->identifier = $row['id'];

                $posts[] = $post;
            }

            return $posts;
        }

        public function dbConnect()
        {
            if ($this->database === null) {
                $this->database = new PDO('mysql:host=localhost;dbname=blog;
                charset=utf8', 'blog', 'password');
            }
        }
}
```

Actuellement, si on change la valeur de la propriété `$database` d'une instance de `PostRepository`, ça ne modifiera pas la valeur de la propriété dans les autres instances. En effet, jusqu'à présent, on voulait que chaque objet soit **autonome** et ait son **propre contexte**. C'était nos deux prérequis de la séance précédente.

Techniquement, avec le mot-clé `static`, nos objets pourront utiliser la même connexion et partager le même contexte du coup.
Mais c'est une pratique plutôt déconseillée. En plus, dès qu'on passera au modèle des commentaires, la problématique se posera à nouveau. Et à ce moment-là, avec deux classes différentes, ça sera réellement impossible que leurs instances partagent le même contexte.

Nous allons plutôt créer une nouvelle classe `DatabaseConnection` qui représentera une connexion avec la base de données.

2.2 Créez la classe DatabaseConnection

Travail à faire

Notre classe `DatabaseConnection` est une nouvelle catégorie de code PHP qu'on pourrait qualifier d'**outillage**. Il sert à résoudre des problèmes purement techniques : ici, maintenir une connexion à une base de données SQL. Pour ce projet, nous allons la mettre dans `src/lib/database.php` :

```php
<?php

class DatabaseConnection
{
        public ?PDO $database = null;

        public function getConnection(): PDO
        {
            if ($this->database === null) {
                $this->database = new PDO('mysql:host=localhost;dbname=blog;
                charset=utf8', 'root', 'root');
            }

            return $this->database;
        }
}
```

Explications :

- On crée la classe `DatabaseConnection`, qui encapsule la connexion `PDO` dans la propriété `$database`.
- Ensuite, on définit une méthode `getConnection()`, qui renvoie forcément une instance de `PDO`.
- À l'intérieur de cette méthode, on initialise la connexion si elle ne l'est pas déjà.

Avec cette classe, il reste possible d'établir une nouvelle connexion SQL, en parallèle à l'ancienne, simplement en créant un nouvel objet de type `DatabaseConnection`.

2.3 Utilisez PostRepository avec DatabaseConnection

Travail à faire

① Nous devons tout d'abord utiliser cette nouvelle classe `DatabaseConnection` à l'intérieur de la classe `PostRepository` :

```php
<?php

require_once('src/lib/database.php');

// ...

class PostRepository
{
        public DatabaseConnection $connection;

        public function getPost(string $identifier): Post
        {
            $statement = $this->connection->getConnection()->prepare(
                "SELECT id, title, content, DATE_FORMAT(creation_date,
                '%d/%m/%Y à %Hh%imin%ss') AS french_creation_date FROM posts
                WHERE id = ?"
            );

            // ...
        }

        public function getPosts(): array
        {
            $statement = $this->connection->getConnection()->query(
                "SELECT id, title, content, DATE_FORMAT(creation_date,
                '%d/%m/%Y à %Hh%imin%ss') AS french_creation_date FROM posts
                ORDER BY creation_date DESC LIMIT 0, 5"
            );

            // ...
        }
}
```

Récapitulatif des changements :

- Ligne 3 : on inclut le nouveau fichier `src/lib/database.php`.
- Ligne 9 : on remplace la propriété `$database` par `$connection` de type `DatabaseConnection`.
- Lignes 13 et 24 : on remplace l'accès à la propriété `$database` par l'appel de la méthode `getConnection()` de la propriété `$connection`.

❣ N'oubliez pas de supprimer la méthode `dbConnect()`.

② Vous trouverez ci-après le contenu modifié de nos deux contrôleurs `src/controllers/homepage.php` et `src/controllers/post.php` lorsqu'ils utilisent la nouvelle forme de la classe `PostRepository` :

```php
<?php
// src/controllers/homepage.php

require_once('src/lib/database.php');
require_once('src/model/post.php');

function homepage()
{
        $postRepository = new PostRepository();
        $postRepository->connection = new DatabaseConnection();
        $posts = $postRepository->getPosts();

        require('templates/homepage.php');
}
```

```php
<?php
// src/controllers/post.php

require_once('src/lib/database.php');
require_once('src/model/comment.php');
require_once('src/model/post.php');

function post(string $identifier)
{
        $postRepository = new PostRepository();
        $postRepository->connection = new DatabaseConnection();
        $post = $postRepository->getPost($identifier);
        $comments = getComments($identifier);

        require('templates/post.php');
}
```

③ Enfin, c'est à votre tour de refactoriser le modèle des commentaires, pour qu'il utilise la nouvelle classe `DatabaseConnection`. Une fois que vous aurez réussi, il n'y aura plus qu'une seule connexion au serveur SQL dans votre contrôleur `src/controllers/post.php`.
N'oubliez pas la fonctionnalité d'ajout de commentaire.

Créez une nouvelle version de votre projet avec l'enchaînement de commandes ci-après :

```
> git add --all
> git commit -m "Tirez parti de la composition"
> git push -u origin main
```

3 Utilisez les namespaces

3.1 Découvrez le rôle des namespaces

Imaginez que vous travaillez sur un gros programme. Vous réutilisez plusieurs bibliothèques. Vous pouvez être sûr qu'à un moment donné, deux classes porteront le même nom. À ce moment-là, c'est le plantage garanti : on n'a pas le droit d'appeler deux classes par le même nom, sauf si on utilise les namespaces. Ils agissent un peu comme des dossiers. Ils vous permettent d'avoir 2 classes du même nom dans votre programme, du moment qu'elles sont dans des namespaces différents.

3.2 Mettez en place votre premier namespace

Concrètement, les namespaces ont cette forme :

Entreprise\Projet\Section

Ce sont vraiment comme des dossiers. D'ailleurs, il est recommandé d'utiliser l'arborescence des dossiers pour construire celle de l'espace de nom. Vous pouvez en imbriquer autant que vous voulez :

Entreprise\Projet\Section\SousSection\SousSousSection

Dans la pratique, il y a deux cas principaux pour choisir la racine de son espace de nom :

- Soit vous codez une bibliothèque à réutiliser, et vous commencerez par le nom de l'entreprise qui est responsable du projet, suivi du nom du projet. Utilisez votre nom ou votre pseudonyme, si vous n'avez pas d'entreprise.
- Ou vous codez une application finale, dont le code ne sera pas partagé. Dans ce cas, il est plutôt recommandé d'utiliser le nom générique 'Application' (ou 'App') comme racine de vos espaces de nom. Avec notre blog, nous sommes dans cette seconde catégorie.

Pour définir un namespace, rien de plus simple. On va déclarer un `namespace` juste avant la définition de la classe.
Exemple :

```php
<?php
// src/model/post.php

namespace Application\Model\Post;

require_once('src/lib/database.php');

// ...

class PostRepository
{
        public \DatabaseConnection $connection;
```

```
13
14          // ...
15  }
```

Ligne 12 : en plaçant le contenu du fichier `src/model/post.php` dans notre namespace, nous avons un problème pour appeler `DatabaseConnection`. En effet, `DatabaseConnection` est une classe qui se trouve à la racine (dans le namespace global). Pour régler le problème, il a fallu écrire `\DatabaseConnection`.

Désormais, les noms complets de toutes les classes et fonctions déclarées dans ce fichier sont différents. Ils sont le résultat de la concaténation de l'espace de nom avec leur nom interne.
Tous les fichiers qui font appel aux classes de ce fichier doivent maintenant ajouter le namespace en préfixe. Voilà par exemple à quoi va ressembler votre contrôleur `src/controllers/post.php` :

```php
<?php
// src/controllers/post.php
require_once('src/lib/database.php');
require_once('src/model/comment.php');
require_once('src/model/post.php');

function post(string $identifier)
{
        $connection = new DatabaseConnection();

        $postRepository = new \Application\Model\Post\PostRepository();
        $postRepository->connection = $connection;
        $post = $postRepository->getPost($identifier);

        $commentRepository = new CommentRepository();
        $commentRepository->connection = $connection;
        $comments = $commentRepository->getComments($identifier);

        require('templates/post.php');
}
```

3.3 Évitez la répétition du préfixe

il y a un moyen d'éviter de répéter le namespace en préfixe à chaque fois : il faut utiliser le mot-clé `use` au début d'un fichier qui fait régulièrement appel à des classes d'un même namespace :

```php
<?php
// src/controllers/post.php
require_once('src/lib/database.php');
require_once('src/model/comment.php');
```

```php
require_once('src/model/post.php');

use Application\Model\Post\PostRepository;

function post(string $identifier)
{
        $connection = new DatabaseConnection();

        $postRepository = new PostRepository();
        $postRepository->connection = $connection;
        $post = $postRepository->getPost($identifier);

        $commentRepository = new CommentRepository();
        $commentRepository->connection = $connection;
        $comments = $commentRepository->getComments($identifier);

        require('templates/post.php');
}
```

La plupart du code que vous pourrez croiser dans le milieu professionnel utilise toujours ces `use`. C'est une manière de construire un index des dépendances de votre fichier.

Travail à faire

Mettez tout le code situé dans le dossier `src/` dans des espaces de noms. Ça va faire beaucoup de petites modifications et vous devrez vérifier que toutes les fonctionnalités de votre blog restent fonctionnelles.
Pour le code de vos contrôleurs, vous avez le choix : refactoriser vos contrôleurs en objets ou les laisser en fonctions. La première option est recommandée, mais, si vous prenez la seconde, sachez que le mot-clé `use function` existe.

Créez une nouvelle version de votre projet avec l'enchaînement de commandes ci-après :

```
> git add --all
> git commit -m "Utilisez les namespaces"
> git push -u origin main
```

4 Modifiez un commentaire

Travail à faire

Tout visiteur doit pouvoir modifier à tout moment n'importe quel commentaire. Un lien "modifier" doit être présent à côté de chaque commentaire et ouvrir une page qui permet de modifier le commentaire.
Vous me direz que ce n'est pas raisonnable d'autoriser tout le monde à modifier tous les commentaires. Je vous réponds tout de suite que c'est un exercice dont le but est uniquement pédagogique !
Ce qui est important dans ce travail à faire, c'est de respecter les bonnes pratiques du cours : votre code doit respecter MVC et le modèle doit être en objet. Et n'oubliez pas les namespaces.
Vous vérifiez d'ailleurs que votre code respecte bien les critères suivants :

- Il y a un lien pour modifier chaque commentaire.
- La modification de commentaire fonctionne.
- Chaque sous-élément de la fonctionnalité est à sa place dans le code : Modèle, Vue, Contrôleur.
- Le modèle utilise la programmation orientée objet, telle que montrée dans le cours.
- Chaque classe est dans un namespace.

Créez une nouvelle version de votre projet avec l'enchaînement de commandes ci-après :

```
> git add --all
> git commit -m "Modifiez les commentaires"
> git push -u origin main
```

5 Pour ceux qui pédalent vite

5.1 Autoloadez

Il existe une solution pour ne plus avoir à utiliser des `require_once` en plus des espaces de nom. C'est le but du système **autoload** fourni par PHP.

Concrètement, l'autoload est un mécanisme qui va être appelé à chaque fois que vous pointez sur un nom tiré d'un namespace. Pour chaque nom complet pointé (une classe, par exemple), un fichier PHP va être "automatiquement chargé". Et cette association entre "espace de nom" et "fichier à inclure" est entièrement **personnalisable**.

Vous pouvez donc configurer un autoload qui va automatiquement charger toutes les classes du namespace `Application\Model\Comment\` depuis le fichier `src/model/comment.php`. Il est possible de faire de même pour le reste de votre modèle ainsi que pour vos contrôleurs.

Cette technique est puissante, certes, mais vraiment fastidieuse à la longue. Et c'est pour ça que la plupart des développeurs PHP utilisent une convention pour créer leurs classes et leurs fichiers : la PSR-4. Pour la résumer :

- Une seule et unique classe doit être présente par fichier.
- Le nom du fichier doit correspondre au nom de la classe (en respectant la casse).
- On a le droit à un préfixe personnalisé, qui correspondra à un dossier personnalisé.

Quant à l'activation de l'autoload dans votre code... Il faut savoir que sur la plupart des projets sur lesquels vous travaillerez, les développeurs utilisent un gestionnaire de dépendances, appelé Composer. Il sert à inclure dans votre projet du code qui n'est pas maintenu par vous. C'est extrêmement puissant. Eh bien, ce gestionnaire de dépendances est capable de générer automatiquement un fichier PHP qui configure l'autoload. Il ne vous restera qu'à l'inclure en tête de votre `index.php` :

```php
<?php

require __DIR__ . '/vendor/autoload.php';
```

5.2 Documentez

Les codes professionnels sont souvent bien documentés. Il existe des conventions de documentation. En PHP, on commente dans le formatPHPDoc (inspirée de la Javadoc du langage Java).

À l'aide d'un programme spécial PHPDoc, on peut générer automatiquement une documentation (au format HTML) depuis votre code.