Architecture MVC en PHP*

Partie 3 : Passez à une architecture MVC orientée objet

Chapitres 1 et 2

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-poo.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 la base de données MySQL dénommée `blog`, créée lors de la précédente séance.

Vous devrez éventuellement remplacer les identifiants de connexion à la base de données utilisés dans le code par les vôtres dans le fichier `src/model.php`, aux lignes du type :

```
$bdd = new PDO('mysql:host=localhost;dbname=blog;charset=utf8', 'root', 'root');
```

Votre base de données est en principe déjà remplie. Vous pouvez recharger le schéma par défaut (et quelques données), contenu dans le fichier `db.sql`.

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

```
> git add --all
> git commit -m "Base pour passer à une architecture MVC OO"
> git push -u origin main
```

*Mathieu Nebra

2 Structurez vos données

C'est parti pour cette nouvelle partie de cours, dédiée à la **Programmation Orientée Objet**, appliquée au patron de conception MVC !

2.1 Créez et utilisez un nouveau type Comment

Dans votre code, vous aviez représenté vos commentaires en les **composant** de trois variables (auteur, date et commentaire) de types prédéfinis fournis par PHP.

Nous allons créer une classe `Comment` ayant ces 3 propriétés auteur date et commentaire avec le code PHP (version 8 et supérieures) suivant :

```php
<?php

class Comment
{
    public string $author;
    public string $frenchCreationDate;
    public string $comment;
}
```

Code PHP versions antérieures :

```php
<?php

class Comment
{
    public $author;
    public $frenchCreationDate;
    public $comment;
}
```

Pour créez une variable de type `Comment`, vous utiliserez l'instruction :

```php
<?php
$comment = new Comment();
```

Les propriétés d'un objet sont accessibles via une nouvelle syntaxe : l'opérateur flèche `->`, suivi du nom de la propriété à laquelle vous accédez. Ainsi, pour renseigner la date de création d'un commentaire, vous allez faire :

```php
<?php
$comment->frenchCreationDate = '15/10/2023 à 11h48';
```

Bien entendu, l'opérateur flèche fonctionne aussi pour l'accès en lecture à des propriétés.

2.1.1 fichier src/model/comment.php

Travail à faire

Modifiez le fichier `src/model/comment.php` pour que la fonction `getComments()` construise et renvoie un tableau d'objets `Comment`.
Indications :
Tout d'abord, vous devez ajouter la déclaration de la classe `Comment` en haut du fichier. Puis, dans la boucle `while`, pour chaque ligne de résultat SQL, vous instanciez un nouvel objet de type `Comment`. Enfin, vous renseignez la valeur de chacune des propriétés.

2.1.2 fichier templates/post.php

Travail à faire

Modifiez le template `templates/post.php`, qui reçoit maintenant des objets à la place de tableaux indexés.
Indications :
Il n'y a que 2 lignes à changer, à l'intérieur du `foreach`. Vous avez juste à remplacer la syntaxe d'accès aux index d'un tableau (`$array['index']`) par celle d'accès aux propriétés d'un objet (`$object->property`).

2.2 Au tour des billets

Travail à faire

Lancez-vous dans la modification des billets en suivant le même schéma que pour les commentaires. Profitez-en pour déplacer le modèle des billets dans un nouveau fichier nommé `src/model/post.php`.

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

```
> git add --all
> git commit -m "Structurez les données"
> git push -u origin main
```

3 Donnez vie à vos structures

3.1 Encapsulez la connexion à la base de données

Nous allons commencer par régler le problème de connexions répétées à la base de données à chaque requête SQL. En effet, se connecter à une base de données, c'est une opération lourde, qui prend

du temps. Il faut le faire le moins possible.

Il nous faut une structure qui contienne la connexion à la base de données, si elle existe, et qu'on passerait en paramètre à chaque fonction.

Faisons-ça tout de suite dans notre fichier `src/model/post.php`.

Travail à faire

Dans le fichier `src/model/post.php` :

- Insérez le code suivant à la ligne 10 : *PHP version 8 et supérieures*

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

//...

class PostRepository
{
        public ?PDO $database = null;
}

// ...
```

PHP versions antérieures

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

//...

class PostRepository
{
        public $database = null;
}

// ...
```

Explications :
`PostRepository` est une classe possèdant une propriété nullable nommée `$database`, qui représente une potentielle connexion avec une base de données. `PDO` est le type de la connexion SQL sous-jacente.

- Modifiez les signatures des fonctions `getPosts`, `getPost` et `dbConnect` respectivement

lignes 16, 35 et 52 en ajoutant à chacune un paramètre `PostRepository $repository`. Cela permet d'avoir la connexion à la base de données, si elle existe, dans le contexte d'exécution des de ces fonctions.

- Modifiez le corps de la fonction `dbConnect` ligne 53 et suivante pour modifier le paramètre `$repository`. En effet, c'est cette fonction qui va être responsable d'initialiser la connexion à la base de données la première fois :

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

//...

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

A ce stade le code du fichier `src/model/post.php` n'est pas opérationnel, il y a quelques modifications que vous devez apporter pour rendre l'ensemble cohérent.

Le screencast `EncapsulezConnBDD.asf` reprend pas à pas ce travail à faire et passe en revue toutes les modifications pour rendre le fichier `src/model/post.php` cohérent.

Le code n'est toujours pas opérationnel, il y aurait encore quelques modifications à apporter dans les autres fichiers pour rendre l'ensemble cohérent.

3.2 Utilisez des méthodes

Notre code, en l'état actuel, possède au moins deux problèmes :

- Le paramètre `PostRepository $repository` est trop souvent présent. Le code est devenu verbeux, et donc moins agréable à écrire. C'est un signe qui ne trompe pas en programmation : vous avez affaire à un **contexte**.

- Et puis, les `$repository` passés en paramètre devront être initialisés par une autre couche de code : le contrôleur. On préfèrerait que le modèle soit **autonome**.

3.2.1 Travail à faire

Nous allons transformer la fonction `getPost()` en méthode dans la classe `PostRepository` . Et automatiquement, c'est comme si elle recevait en premier paramètre sa propre instance de `PostRepository`, nommée `$this` :

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

// ...
class PostRepository
{
  public ?PDO $database = null;

  public function getPost(/*PostRepository $this,*/string $identifier): Post
        {
            dbConnect($this);
            $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;
        }
}
// ...
```

3.2.2 Travail à faire

Les méthodes d'un objet s'utilisent presque de la même façon que ses propriétés : avec l'opérateur flèche `->`. Éditez le fichier `src/controllers/post.php`, nous allons le modifier ensemble :

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

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

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

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

Explications :

1. Vous commencez par créer une instance de votre objet `PostRepository`, qui contiendra sa propre connexion à la base de données. ☞ ligne 9

2. Puis, vous utilisez `$postRepository->getPost($identifier)` pour demander à votre objet de vous faire parvenir le bon billet de blog. ☞ ligne 10

Le screencast `UtilisezDesMéthodes.asf` reprend pas à pas ces deux derniers travaux.

3.3 Finalisez la refactorisation des billets

3.3.1 Travail à faire

Pour le moment, le code de notre fichier `src/model/post.php` est dans un état un peu brouillon. Une méthode dans la classe et deux fonctions en dehors. Vous allez harmoniser tout ça, en transformant les deux fonctions en méthodes de la classe `PostRepository`.

Il vous faudra aussi modifier le fichier `src/controllers/homepage.php`, pour qu'il utilise à son tour un `PostRepository` :

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

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

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

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

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

```
> git add --all
> git commit -m "Donnez vie à vos structures"
> git push -u origin main
```