

Création d'un mini blog en PHP

MVC et POO

Table des matières

1. Configuration.....	3
Serveur Web.....	3
Navigateurs.....	3
Éditeur de texte ou IDE.....	3
Configurer PHP	3
Afficher les erreurs.....	3
Composer.....	4
2. La classe Database.....	5
La base de données.....	5
La classe Database.....	5
Améliorons notre classe Database.....	9
Modification de notre fichier Database.....	9
Versionning.....	10
Pour ceux qui utilisent PHPStorm.....	10
Quelques révisions.....	10
Bilan.....	10
3. La classe Article.....	11
La base de données.....	11
La classe Article.....	12
Récupération de la connexion à la base de données.....	12
Afficher un article.....	14
Mise en place d'une navigation.....	16
Quelques révisions.....	17
Bilan.....	17
4. Un peu de refactorisation.....	18
Petit topo.....	18
L'existant.....	18
La classe Database.....	19
La classe Article.....	23
Travailler avec des objets.....	24
Quelques révisions.....	29
Bilan.....	29
5. L'architecture et les namespaces.....	30
Refactorisons notre application : étape 1.....	30
Refactorisons notre application : étape 2.....	30
Utilisons les namespaces.....	31
Quelques révisions.....	34
Bilan.....	34
6. L'autoloader.....	35
Mettons en place un autoloader.....	35


Quelques révisions	36
Bilan	36
7. L'autoload de Composer	37
Créons notre composer.json	37
Utiliser l'autoload de Composer	37
Versionning	38
Bilan	38
8. Le contrôleur frontal	39
Quelques explications	39
index.php	39
Améliorons notre architecture	40
Bilan	42
9. Le Router	42
Mise en place du router	42
Bilan	43
10. Controller	44
Notre premier contrôleur	44
Un peu de refactorisation	47
ErrorController, pour gérer les erreurs	49
Bilan	51
11. Model	52
Notre premier model : Article.php	52
Bilan	58
12. View	59
Notre classe View	59
Un template de base	60
Bilan	61

1. Configuration

Serveur Web

Dans le cadre de ce projet, nous utiliserons Laragon

Éditeur de texte ou IDE

-  [PHPSTORM \(IDE\)](#)
-  [VISUAL STUDIO CODE](#)

Configurer PHP

Dernière chose avant de rentrer dans le vif du sujet, la configuration de PHP est à vérifier !

Afficher les erreurs

Pour afficher les erreurs dans votre navigateur, je vous invite à ouvrir votre terminal et à lancer la commande suivante :

```
php -v
```

S'il vous affiche la version de PHP et des informations concernant xdebug, c'est que ce dernier est activé, vous pouvez donc passer cette étape, sinon, je vous invite à localiser votre version de php en exécutant la commande :

```
php --ini
```

Ouvrez ce fichier php.ini dans votre éditeur de texte et recherchez une ligne (et non un groupe de lignes) qui contient :

```
display_errors = Off
```

Changer la valeur de ce dernier :

```
display_errors = On
```

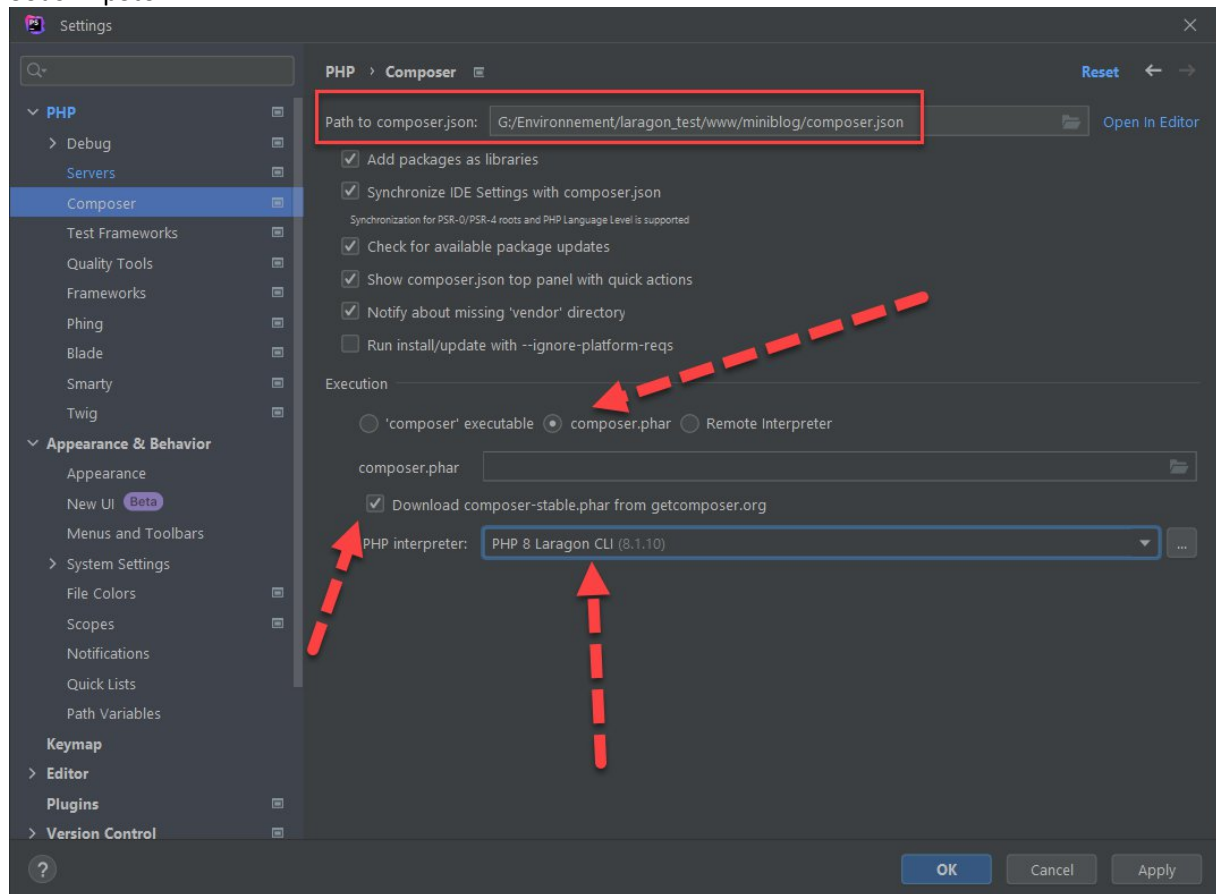
S'il y a un point-virgule en début de ligne, supprimez-le pour activer la configuration.

Enregistrez votre fichier et... passez à la suite, ne fermez pas votre fichier pour le moment 😊

Composer

Voici le dernier outil à installer dans notre liste : Composer, qui est un gestionnaire de dépendances pour PHP.

Sous PhpStorm

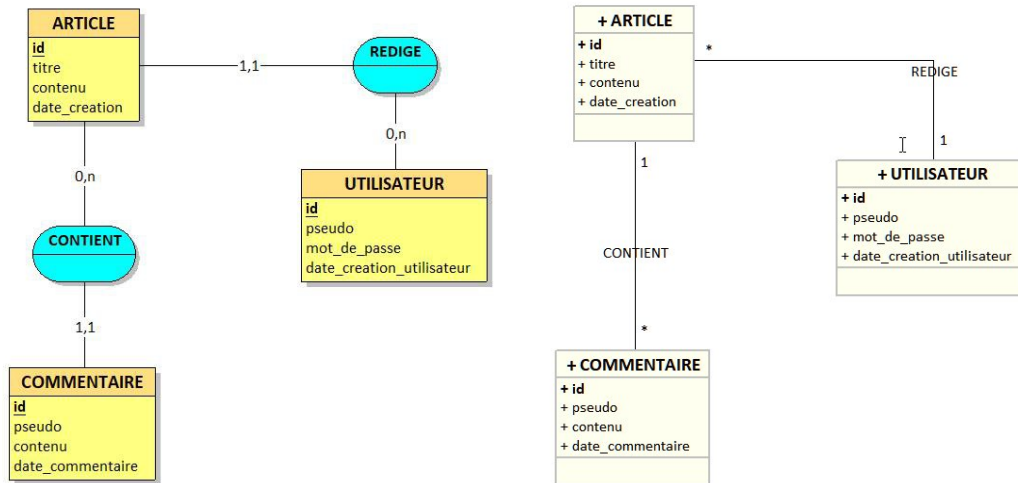


Maintenant, lisez chaque chapitre en entier avant de commencer à coder !
Ne vous contentez pas d'un simple copié/collé ...
Il s'agit de comprendre le code

2. La classe Database

La base de données

Nous allons commencer par travailler avec notre base de données.



Formalise Entité/Association

Formalisme UML

Voici le schéma SQL actuel de notre base de données **blogg** :

```
CREATE DATABASE IF NOT EXISTS `blogg` DEFAULT CHARACTER SET utf8 COLLATE utf8_general_ci;
```

Tous les scripts seront conservés dans le dossier **sql** à la racine du projet

Créez donc ce dossier **sql** et créez le fichier **db.sql** à l'intérieur avec le contenu de la requête sql ci-dessus.

La classe Database

Passons maintenant côté PHP où nous allons créer une classe concernant notre base de données dans un fichier appelé **Database.php** (à la racine de notre projet):

```
<?php
class Database
{
}
```

Attention au nommage de la classe : la première lettre de la classe commence par une Majuscule !

La première chose dont nous devons nous occuper, c'est de se connecter à notre base.

Pour ceci, nous allons créer une méthode (fonction dépendante d'un objet) que nous allons appeler **getConnection()**

```
<?php
class Database
{
    public function getConnection()
    {
    }
}
```

Euh oui, mais ta méthode, elle ne fait rien là ?


Tout juste, nous allons ajouter le code à l'intérieur

```

<?php
class Database
{
    //Méthode de connexion à notre base de données
    public function getConnection()
    {
        //Tentative de connexion à la base de données
        try{
            $connection = new
PDO('mysql:host=localhost;dbname=blogg;charset=utf8', 'root', '');
            $connection->setAttribute(PDO::ATTR_ERRMODE,
PDO::ERRMODE_EXCEPTION);
        }
        //On lève une erreur si la connexion échoue
        catch(Exception $errorConnection)
        {
            die ('Erreur de connexion :'. $errorConnection->getMessage());
        }
    }
}

```

Pensez à changer les valeurs de `host`, `dbname`, `user` (ici root), et `pass` (ici vide '') pour que cela fonctionne avec votre application web

Créons maintenant un fichier  `home.php` (à la racine de notre projet) avec le contenu suivant :

```

<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="utf-8">
    <title>Mon blog</title>
</head>

<body>
    <div>
        <h1>Mon blog</h1>
        <p>En construction</p>
    </div>
</body>
</html>

```

Si vous essayez d'accéder à la page web, ici <http://localhost/blog/home.php>, vous devriez avoir l'affichage correct.

Pour le moment, nous avons juste créé une méthode pour nous connecter à la base de données, nous ne l'avons jamais utilisé dans notre application.

Modifions maintenant notre fichier `home.php` dans lequel nous allons faire appel à cette méthode

```
<?php
//On inclut le fichier dont on a besoin (ici à la racine de notre site)
require 'Database.php';
?>
<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="utf-8">
    <title>Mon blog</title>
</head>

<body>
<div>
    <h1>Mon blog</h1>
    <p>En construction</p>
    <?php
        //On crée un nouvel objet $db, qui est une instance de la classe
        Database
        $db = new Database();
        //On fait appel à notre méthode getConnection()
        $db->getConnection();
    ?>
</div>
</body>
</html>
```

C'est tout ? Ça ne marche toujours pas ! Rien ne s'affiche !

Pour le moment, rien ne s'affiche, mais c'est tout à fait normal. Nous ne lui avons pas demandé d'afficher quoi que ce soit à l'écran.

Modifions légèrement notre code pour vérifier ce qui se passe à l'écran :

Le fichier `Database.php` :

```
<?php

class Database
{
    //Méthode de connexion à notre base de données
    public function getConnection()
    {
        //Tentative de connexion à la base de données
        try{
            $connection = new
PDO('mysql:host=localhost;dbname=blogg;charset=utf8', 'root', '');
            $connection->setAttribute(PDO::ATTR_ERRMODE,
PDO::ERRMODE_EXCEPTION);
            //On renvoie un message avec le mot-clé return
            return 'Connexion OK';
        }
        //On lève une erreur si la connexion échoue
        catch(Exception $errorConnection)
        {
            die ('Erreur de connexion :'.$errorConnection->getMessage());
        }
    }
}
```

Le fichier home.php :

```
<?php
//On inclut le fichier dont on a besoin (ici à la racine de notre site)
require 'Database.php';
?>
<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="utf-8">
    <title>Mon blog</title>
</head>

<body>
    <div>
        <h1>Mon blog</h1>
        <p>En construction</p>
        <?php

            $db = new Database();
            //On ajoute un echo pour vérifier qu'un message s'affiche à l'écran
            echo $db->getConnection();
        ?>
    </div>
</body>
</html>
```

Ici, on a appliqué deux modifications :

- dans notre classe **Database**, on renvoie un message avec le mot-clé **return** ;
- dans notre fichier **home.php**, ce message est affiché grâce au **echo** ajouté.

Si vous actualisez la fenêtre de votre navigateur, en retournant sur <http://localhost/blog/home.php>, tout fonctionne.

Si vous essayez de changer un paramètre de connexion (hôte, user ou pass), le navigateur web devrait vous afficher une erreur (le message contenu dans le catch écrit précédemment).

ça marche !!! On fait quoi maintenant ? Quelle est l'étape suivante ?

Améliorons notre classe Database

Notre classe fonctionne maintenant, pourtant nous ne nous connecterons pas à celle-ci de la sorte. Ici, la connexion se fait, mais aucune interaction n'est faite dans celle-ci.

Oui mais c'est ce qu'on va faire par la suite ?

Vous avez raison. Mais rien n'empêche d'améliorer notre application régulièrement.

Nous allons régulièrement réécrire notre code pour améliorer notre application : on appelle ça la **refactorisation**.

Nous faisons actuellement face à un problème : les paramètres de connexion sont noyés dans le code.

On va y remédier immédiatement 😊

Modification de notre fichier Database

Nous allons sortir les paramètres de connexion et les stocker dans des ... **CONSTANTES**

Pourquoi pas dans des variables ?

Les paramètres de connexion ne vont pas changer, les constantes sont donc parfaitement adaptées à notre besoin.

Voici donc notre classe mise à jour :

```
<?php

class Database
{
    //Nos constantes
    const DB_HOST = 'mysql:host=localhost;dbname=blogg;charset=utf8';
    const DB_USER = 'root';
    const DB_PASS = '';

    //Méthode de connexion à notre base de données
    public function getConnection()
    {
        //Tentative de connexion à la base de données
        try{
            $connection = new PDO(self::DB_HOST, self::DB_USER,
self::DB_PASS);
            $connection->setAttribute(PDO::ATTR_ERRMODE,
PDO::ERRMODE_EXCEPTION);
            //On renvoie un message avec le mot-clé return
            return 'Connexion OK';
        }
        //On lève une erreur si la connexion échoue
        catch(Exception $errorConnection)
        {
            die ('Erreur de connection :'. $errorConnection->getMessage());
        }
    }
}
```

self ? self ? Pourquoi self et pas \$this ?

self fait référence à la classe et **\$this** à l'objet, ici les paramètres ne vont pas changer, ils appartiennent à la classe : on utilise donc **self**.

On peut de la même façon remplacer **self** par le nom de la classe, ici **Database**

```
<?php

class Database
{
    //Nos constantes
    const DB_HOST = 'mysql:host=localhost;dbname=blogg;charset=utf8';
    const DB_USER = 'root';
    const DB_PASS = '';

    //Méthode de connexion à notre base de données
    public function getConnection()
    {
        //Tentative de connexion à la base de données
        try{
            $connection = new PDO(Database::DB_HOST, Database::DB_USER,
Database::DB_PASS);
            $connection->setAttribute(PDO::ATTR_ERRMODE,
PDO::ERRMODE_EXCEPTION);
            //On renvoie un message avec le mot-clé return
            return 'Connexion OK';
        }
        //On lève une erreur si la connexion échoue
        catch(Exception $errorConnection)
        {
            die ('Erreur de connexion :'.$errorConnection->getMessage());
        }
    }
}
```

Par la suite, je garderai **self** et non **Database**, pour une raison simple : si le nom de la classe doit changer, il n'y aura pas de modification supplémentaire à faire que de changer le nom de la classe.

Versionning

Nous allons versionner notre code avec **git**, vous pourrez ainsi l'héberger sur une plateforme de votre choix (GitHub, Gitlab, Bitbucket...).

Pour ceux qui utilisent PhpStorm

Créons maintenant un fichier **.gitignore** à la racine de notre projet et ajoutons le contenu suivant :
.idea/

Cela permettra de ne pas versionner le dossier **idea** qui est utilisé par PhpStorm pour configurer votre projet.

Quelques révisions

Si vous avez besoin de revoir certains points, voici les liens en conséquence :

- [!\[\]\(33006de4dd11f8c729ca8ca0fde0352f_img.jpg\) DOCUMENTATION MySQL](#)
- [!\[\]\(d5f9ffa97ddb414b7e96feb8ad710c8e_img.jpg\) CLASSE EN PHP](#)
- [!\[\]\(d900cae4f5a7d73d67b6a98ff3e7bb9a_img.jpg\) PDO](#)
- [!\[\]\(ca15981e1cca5aa5bb6984590487f7b9_img.jpg\) REQUIRE](#)
- [!\[\]\(bbe2f84e2181f98756cb54246365881c_img.jpg\) CONSTANTES](#)
- [!\[\]\(4758ad81fc3759288da80cbff59390b5_img.jpg\) GIT](#)

Bilan

Dans ce chapitre, nous avons travaillé avec notre classe **Database** qui permettra d'interagir avec notre base de données.

3. La classe Article

La base de données

Il est temps de passer à la gestion de nos articles...

Mais nous n'avons pas encore d'articles ?

Tout juste, nous allons modifier la structure de notre base de données en ajoutant une table article 😊

Voici le schéma de notre base de données (j'ai ajouté 3 articles).

```
CREATE TABLE `article` (  
  `id` int(11) NOT NULL,  
  `title` varchar(100) NOT NULL,  
  `content` text NOT NULL,  
  `author` varchar(100) NOT NULL,  
  `createdAt` datetime NOT NULL  
) ENGINE=InnoDB DEFAULT CHARSET=utf8;  
  
INSERT INTO `article` (`id`, `title`, `content`, `author`, `createdAt`) VALUES  
(1, 'Voici mon premier article', 'Mon super blog est en construction.', Lia, '2019-03-15 08:10:24'),  
(2, 'Un deuxième article', 'Je continue à ajouter du contenu sur mon blog.', Lia, '2019-03-16 13:27:38'),  
(3, 'Mon troisième article', 'Mon blog est génial !!!', Lia, '2019-03-16 14:45:57');  
  
ALTER TABLE `article`  
ADD PRIMARY KEY (`id`);  
  
ALTER TABLE `article`  
MODIFY `id` int(11) NOT NULL AUTO INCREMENT, AUTO INCREMENT=4;
```

Ici on retrouve une table article classique avec les champs suivants :

- **id** qui est la **PRIMARY KEY** en **AUTO INCREMENT**
- **title** qui est de type **VARCHAR** limité à 100 caractères
- **content** qui est de type **TEXT**
- **author** qui est de type **VARCHAR** limité à 100 caractères
- **createdAt** qui est de type **DATETIME**

Tous les champs sont indiqués comme **NOT NULL**.

J'ai créé le fichier `articles.sql` dans le dossier `sql`.

Dans la suite de ce cours, nous ajouterons de nouveaux articles depuis notre application.

La classe Article

Nous allons commencer par créer une classe **Article** dans un fichier `Article.php` à la racine de notre projet :

```
<?php

class Article
{
}

```

Nous allons commencer par récupérer la connexion à la base de données et récupérer nos articles dans une méthode `getArticles`

Récupération de la connexion à la base de données

On va modifier notre fichier `Database.php` pour renvoyer la connexion :

```
<?php

class Database
{
    //Nos constantes
    const DB_HOST = 'mysql:host=localhost;dbname=blogg;charset=utf8';
    const DB_USER = 'root';
    const DB_PASS = '';

    //Méthode de connexion à notre base de données
    public function getConnection()
    {
        //Tentative de connexion à la base de données
        try{
            $connection = new PDO(self::DB_HOST, self::DB_USER,
self::DB_PASS);
            $connection->setAttribute(PDO::ATTR_ERRMODE,
PDO::ERRMODE_EXCEPTION);
            //On renvoie la connexion
            return $connection;
        }
        //On lève une erreur si la connexion échoue
        catch(Exception $errorConnection)
        {
            die ('Erreur de connexion :'. $errorConnection->getMessage());
        }
    }
}

```

Ici, la seule modification a été faite pour renvoyer la connexion à la base de données (qui sera utilisée par la classe **Article**).

Nous allons modifier notre fichier `Article.php` ainsi que le fichier `home.php`

Commençons par créer notre méthode `getArticles()` qui va nous permettre de récupérer tous les articles :

```
<?php

class Article
{
    public function getArticles()
    {
        $db = new Database();
        $connection = $db->getConnection();
        $result = $connection->query('SELECT id, title, content, author,
createdAt FROM article ORDER BY id DESC');
        return $result;
    }
}

```

Et le fichier `home.php` en conséquence :

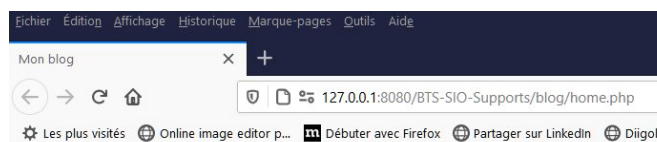
```
<?php
//On inclut le fichier dont on a besoin (ici à la racine de notre site)
require 'Database.php';
//Ne pas oublier d'ajouter le fichier Article.php
require 'Article.php';
?>
<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="utf-8">
    <title>Mon blog</title>
</head>

<body>
<div>
    <h1>Mon blog</h1>
    <p>En construction</p>
    <?php
        $article = new Article();
        $articles = $article->getArticles();
        while($article = $articles->fetch())
        {
            ?>
            <div>
                <h2><?= htmlspecialchars($article['title']);?></h2>
                <p><?= htmlspecialchars($article['content']);?></p>
                <p><?= htmlspecialchars($article['author']);?></p>
                <p>Créé le : <?= htmlspecialchars($article['createdAt']);?></p>
            </div>
            <br>
            <?php
        }
        $articles->closeCursor();
    ?>
</div>
</body>
</html>
```

Quelques explications complémentaires :

- Dans le corps de la page, une instance de la classe `Article` est créée avec `$article = new Article();`. Cette classe doit être définie dans le fichier `Article.php` que nous avons inclus précédemment.
- La méthode `getArticles()` de l'objet `$article` est ensuite appelée pour récupérer tous les articles de la base de données. Cette méthode renvoie un objet `PDOStatement` qui peut être parcouru à l'aide d'une boucle `while`.
- Dans la boucle `while`, chaque article est affiché dans une `div` en utilisant les données récupérées depuis la base de données. Les données sont affichées à l'aide de la fonction `htmlspecialchars()` qui permet d'éviter les attaques XSS en encodant les caractères spéciaux.
- Enfin, la méthode `closeCursor()` est appelée pour libérer les ressources utilisées par l'objet `PDOStatement`.

Voici le résultat dans notre navigateur web :



Notre page affiche notre liste d'articles, classé par leurs identifiants, en ordre décroissant.

Il est temps de travailler sur l'affichage d'un article en particulier.

Mon blog

En construction

[Mon troisième article](#)

Mon blog est génial !!!

Karim

Créé le : 2019-03-16 14:45:57

[Un deuxième article](#)

Je continue à ajouter du contenu sur mon blog.

Karim

Créé le : 2019-03-16 13:27:38

[Voici mon premier article](#)

Mon super blog est en construction.

Karim

Créé le : 2019-03-15 08:10:24

Afficher un article

Notre page liste maintenant nos articles, nous allons travailler sur la page qui affiche un article en particulier. On va commencer par ajouter une méthode `getArticle()` pour récupérer un article unique :

```
<?php

class Article
{
    public function getArticles()
    {
        $db = new Database();
        $connection = $db->getConnection();
        $result = $connection->query('SELECT id, title, content, author,
        createdAt FROM article ORDER BY id DESC');
        return $result;
    }

    public function getArticle($articleId)
    {
        $db = new Database();
        $connection = $db->getConnection();
        $result = $connection->prepare('SELECT id, title, content, author,
        createdAt FROM article WHERE id = ?');
        $result->execute([
            $articleId
        ]);
        return $result;
    }
}
```

Ici, on va passer un paramètre à notre méthode `getArticle` pour récupérer un article en particulier (symbolisé par `$idArt`).

Créons maintenant un fichier `single.php` (à la racine de notre projet) qui va afficher un article en particulier :

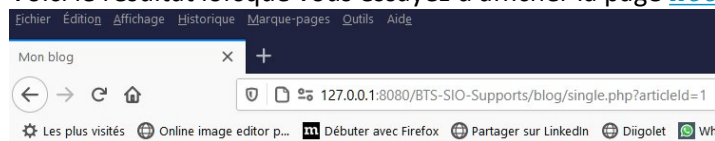
```
<?php
//On inclut le fichier dont on a besoin (ici à la racine de notre site)
require 'Database.php';
//Ne pas oublier d'ajouter le fichier Article.php
require 'Article.php';
?>

<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="utf-8">
    <title>Mon blog</title>
</head>

<body>
<div>
    <h1>Mon blog</h1>
    <p>En construction</p>
    <?php
        $article = new Article();
        $articles = $article->getArticle(1);
        $article = $articles->fetch();
    ?>
    <div>
        <h2><?= htmlspecialchars($article['title']);?></h2>
        <p><?= htmlspecialchars($article['content']);?></p>
        <p><?= htmlspecialchars($article['author']);?></p>
        <p>Créé le : <?= htmlspecialchars($article['createdAt']);?></p>
    </div>
    <br>
    <?php
        $articles->closeCursor();
    ?>
</div>
</body>
</html>
```

Ici, j'ai passé le paramètre 1 à la méthode `getArticle` pour afficher notre article avec l'identifiant 1 en base de données.

Voici le résultat lorsque vous essayez d'afficher la page <http://localhost/blog/single.php> :



Mon blog

En construction

Voici mon premier article

Mon super blog est en construction.

Karim

Créé le : 2019-03-15 08:10:24

Si je veux un autre article, je n'ai qu'à changer le paramètre dans la méthode ?

Surtout pas ! On ne doit jamais faire ceci de cette manière ! ☹️

On va passer le paramètre dynamiquement, en passant par l'URL.

Mise en place d'une navigation

On va modifier le fichier `home.php` pour ajouter l'identifiant de notre article et en dirigeant vers la page `single.php`

On modifie d'abord notre fichier `home.php` :

```
<?php
//On inclut le fichier dont on a besoin (ici à la racine de notre site)
require 'Database.php';
//Ne pas oublier d'ajouter le fichier Article.php
require 'Article.php';
?>
<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="utf-8">
    <title>Mon blog</title>
</head>
<body>
<div>
    <h1>Mon blog</h1>
    <p>En construction</p>
    <?php
        $article = new Article();
        $articles = $article->getArticles();
        while($article = $articles->fetch())
        {
            ?>
            <div>
                <h2><a href="single.php?articleId=?=
htmlspecialchars($article['id']);?>">?=
htmlspecialchars($article['title']);?></a></h2>
                <p><?= htmlspecialchars($article['content']);?></p>
                <p><?= htmlspecialchars($article['author']);?></p>
                <p>Créé le : <?= htmlspecialchars($article['createdAt']);?></p>
            </div>
            <br>
        <?php
        }
        $articles->closeCursor();
    ?>
</div>
</body>
</html>
```

Sans oublier notre fichier `single.php` :

```
<?php
//On inclut le fichier dont on a besoin (ici à la racine de notre site)
require 'Database.php';
//Ne pas oublier d'ajouter le fichier Article.php
require 'Article.php';
?>

<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="utf-8">
    <title>Mon blog</title>
</head>
<body>
<div>
    <h1>Mon blog</h1>
    <p>En construction</p>
    <?php
```

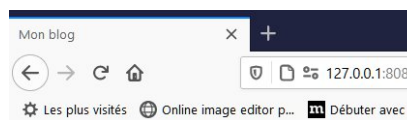


```

    $article = new Article();
    $articles = $article->getArticle($_GET['articleId']);
    $article = $articles->fetch()
    ?>
    <div>
        <h2><?= htmlspecialchars($article['title']);?></h2>
        <p><?= htmlspecialchars($article['content']);?></p>
        <p><?= htmlspecialchars($article['author']);?></p>
        <p>Créé le : <?= htmlspecialchars($article['createdAt']);?></p>
    </div>
    <br>
    <?php
    $articles->closeCursor();
    ?>
    <a href="home.php">Retour à l'accueil</a>
</div>
</body>
</html>

```

Voici le rendu de la page `home.php`



Et celui de la page `single.php` :

Mon blog

En construction

Mon troisième article

Mon blog est génial !!!

Karim

Créé le : 2019-03-16 14:45:57

Un deuxième article

Je continue à ajouter du contenu sur mon blog.

Karim

Créé le : 2019-03-16 13:27:38

Voici mon premier article

Mon super blog est en construction.

Karim

Créé le : 2019-03-15 08:10:24



Mon blog

En construction

Mon troisième article

Mon blog est génial !!!

Karim

Créé le : 2019-03-16 14:45:57

[Retour à l'accueil](#)

Il est temps de refactoriser son application

Quelques révisions

Si vous avez besoin de revoir certains points, voici les liens en conséquence :

- [HTMLSPECIALCHARS](#)
- [VARIABLES SUPERGLOBALES](#)

Bilan

Dans ce chapitre, nous avons ajouté une classe `Article` connectée à notre base de données qui nous permet d'afficher la liste des articles sur la page `home.php` et un article en particulier sur la page `single.php`.

4. Un peu de refactorisation

Petit topo

Il serait maintenant temps de refactoriser le code de notre classe `Article` qui se répète légèrement. La refactorisation consiste à réécrire son code d'une manière différente pour qu'il soit plus facile à lire, à maintenir et à faire évoluer. Vous devez éviter au maximum de répéter votre code.

L'existant

Nous avons actuellement la classe `Database` qui a la structure suivante

```
<?php

class Database
{
    //Nos constantes
    const DB_HOST = 'mysql:host=localhost;dbname=blogg;charset=utf8';
    const DB_USER = 'root';
    const DB_PASS = '';

    //Méthode de connexion à notre base de données
    public function getConnection()
    {
        //Tentative de connexion à la base de données
        try{
            $connection = new PDO(self::DB_HOST, self::DB_USER,
self::DB_PASS);
            $connection->setAttribute(PDO::ATTR_ERRMODE,
PDO::ERRMODE_EXCEPTION);
            //On renvoie la connexion
            return $connection;
        }
        //On lève une erreur si la connexion échoue
        catch(Exception $errorConnection)
        {
            die ('Erreur de connection :'. $errorConnection->getMessage());
        }
    }
}
```

La classe `Article` a la structure suivante

```
<?php

class Article
{
    public function getArticles()
    {
        $db = new Database();
        $connection = $db->getConnection();
        $result = $connection->query('SELECT id, title, content, author,
createdAt FROM article ORDER BY id DESC');
        return $result;
    }

    public function getArticle($articleId)
    {
        $db = new Database();
        $connection = $db->getConnection();
        $result = $connection->prepare('SELECT id, title, content, author,
createdAt FROM article WHERE id = ?');
        $result->execute([
```

```

        $articleId
    });
    return $result;
}
}

```

Il y a ici quelques problèmes :

- dans la classe `Database`, la méthode `getConnection()` est en public et peut être appelée depuis n'importe où.

- dans la classe `Article`, un objet `$db` est instancié à chaque méthode, et la méthode `getConnection()` est répétée.

Nous allons corriger ces incohérences dès maintenant

La classe Database

On va effectuer quelques modifications :

- modifier la méthode `getConnection` en private : pour récupérer la connexion à la base de données uniquement depuis notre classe

- créer une méthode `createQuery`, qui fera appel à notre méthode `getConnection` et va gérer nos requêtes

On commence par créer une méthode `createQuery`, qui va gérer nos requêtes :

```

<?php
class Database
{
    //Nos constantes
    const DB_HOST = 'mysql:host=localhost;dbname=blogg;charset=utf8';
    const DB_USER = 'root';
    const DB_PASS = '';

    //Méthode de connexion à notre base de données
    public function getConnection()
    {
        //Tentative de connexion à la base de données
        try{
            $connection = new PDO(self::DB_HOST, self::DB_USER,
self::DB_PASS);
            $connection->setAttribute(PDO::ATTR_ERRMODE,
PDO::ERRMODE_EXCEPTION);
            //On renvoie la connexion
            return $connection;
        }
        //On lève une erreur si la connexion échoue
        catch(Exception $errorConnection)
        {
            die ('Erreur de connection :'. $errorConnection->getMessage());
        }
    }
    protected function createQuery($sql, $parameters = null)
    {
        if($parameters)
        {
            $result = $this->getConnection()->prepare($sql);
            $result->execute($parameters);
            return $result;
        }
        $result = $this->getConnection()->query($sql);
        return $result;
    }
}

```

Je n'ai pas compris, tu peux nous expliquer ce que tu veux faire ici ?

On vient de créer une nouvelle méthode, appelée `createQuery`, qui prend deux paramètres, une requête sql et des paramètres.

Ces derniers par défaut sont null, étant donné qu'une requête `query` n'a pas besoin de paramètres particuliers.

En revanche, pour une requête `prepare`, on a besoin de lui préciser les paramètres en question. On modifiera en conséquence notre classe `Article`.

On va maintenant s'occuper d'un problème de répétition de la connexion à la base de données.

En quoi cela pose problème alors ?

Lorsqu'une de vos pages fait appel à une seule méthode, pas de problème. Mais si votre page doit faire appel à plusieurs méthodes, et que ces dernières contiennent chacune au moins une requête, on risque d'ouvrir une connexion à chaque requête. Ce serait quand même mieux d'en faire une seule non ?

Oui mais à l'heure actuelle aucune de nos pages ne fait appel à plusieurs requêtes ?

Tout juste, mais un bon développeur doit être en mesure d'anticiper d'éventuels problèmes. Trêve de bavardages, nous avons du travail, nous allons :

- ajouter une propriété `$connection` qui va stocker la connexion s'il y en a une
- ajouter une méthode `checkConnection()` qui va vérifier si une connexion est présente ou non

Voici notre classe Database actualisée

```
<?php
class Database
{
    //Nos constantes
    const DB_HOST = 'mysql:host=localhost;dbname=blogg;charset=utf8';
    const DB_USER = 'root';
    const DB_PASS = '';

    private $connection;

    private function checkConnection()
    {
        //Vérifie si la connexion est nulle et fait appel à getConnection()
        if($this->connection === null) {
            return $this->getConnection();
        }
        //Si la connexion existe, elle est renvoyée, inutile de refaire une
        connexion
        return $this->connection;
    }

    //Méthode de connexion à notre base de données
    public function getConnection()
    {
        //Tentative de connexion à la base de données
        try{
            $this->connection = new PDO(self::DB_HOST, self::DB_USER,
            self::DB_PASS);
            $this->connection->setAttribute(PDO::ATTR_ERRMODE,
            PDO::ERRMODE_EXCEPTION);
            //On renvoie la connexion
            return $this->connection;
        }
        //On lève une erreur si la connexion échoue
        catch(Exception $errorConnection)
        {
            die ('Erreur de connexion :'. $errorConnection->getMessage());
        }
    }
}
```

```
protected function createQuery($sql, $parameters = null)
{
    if($parameters)
    {
        $result = $this->checkConnection()->prepare($sql);
        $result->execute($parameters);
        return $result;
    }
    $result = $this->checkConnection()->query($sql);
    return $result;
}
}
```

Quelques explications :

- l'attribut `$connection` stocke la connexion si celle-ci existe, sinon renvoie `null`
- la méthode `checkConnection()` teste si `$connection` est `null`, et appelle `getConnection()` pour créer une nouvelle connexion. Si `$connection` a une connexion existante, la méthode renvoie celle-ci ;
- la méthode `getConnection()` fait la même chose que précédemment, mais renvoie la connexion dans la propriété `$connection`
- la méthode `createQuery()` a été modifiée, pour vérifier si la connexion existe avant d'en faire une nouvelle au besoin.

Comment je sais si la méthode `checkConnection()` fonctionne bien ?

Il vous suffit de modifier la méthode `checkConnection()` en `public` et d'ajouter deux fois la fonction `var_dump()` comme ici :

```
<?php

public function checkConnection()
{
    //Vérifie si la connexion est nulle et fait appel à getConnection
    if($this->connection == null){
        var_dump('connexion inconnue');
        return $this->getConnection();
    }
    //Si la connexion existe, elle est renvoyée, inutile de refaire une
    connexion
    var_dump('connexion déjà existante');
    return $this->connection;
}
```

Et de faire appel à celle-ci plusieurs fois dans `home.php`, comme ici :

```
<?php
//On inclut le fichier dont on a besoin (ici à la racine de notre site)
require 'Database.php';
//Ne pas oublier d'ajouter le fichier Article.php
require 'Article.php';
?>
<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="utf-8">
    <title>Mon blog</title>
</head>

<body>
<div>
    <h1>Mon blog</h1>
    <p>En construction</p>
    <?php
        $db = new Database();
        $db->checkConnection();
        $article = new Article();
        $articles = $article->getArticles();
```

```

while($article = $articles->fetch())
{
    ?>
    <div>
        <h2><a href="single.php?articleId=<?=
htmlspecialchars($article['id']);?>"><?=
htmlspecialchars($article['title']);?></a></h2>
        <p><?= htmlspecialchars($article['content']);?></p>
        <p><?= htmlspecialchars($article['author']);?></p>
        <p>Créé le : <?= htmlspecialchars($article['createdAt']);?></p>
    </div>
    <br>
    <?php
}
$articles->closeCursor();
$db->checkConnection();
?>
</div>
</body>
</html>

```

⚠ Pensez à repasser la méthode `checkConnection()` en `private` et retirer les `var_dump()` dans la classe `Database`, ainsi que les deux appels à la méthode `checkConnection()` dans la page `home.php` ainsi que l'instanciation de l'objet `$db`.

On peut maintenant passer notre méthode `getConnection()` en `private`, pour qu'elle ne soit appelée que depuis la classe `Database`.

On va aussi passer la classe `Database` en classe abstraite, pour qu'on ne puisse plus l'instancier.

Voici le résultat

```

<?php

abstract class Database
{
    //Nos constantes
    const DB_HOST = 'mysql:host=localhost;dbname=blogg;charset=utf8';
    const DB_USER = 'root';
    const DB_PASS = '';

    private $connection;

    private function checkConnection()
    {
        //Vérifie si la connexion est nulle et fait appel à getConnection()
        if($this->connection === null) {
            return $this->getConnection();
        }
        //Si la connexion existe, elle est renvoyée, inutile de refaire une
        connexion
        return $this->connection;
    }

    //Méthode de connexion à notre base de données
    private function getConnection()
    {
        //Tentative de connexion à la base de données
        try{
            $this->connection = new PDO(self::DB_HOST, self::DB_USER,
self::DB_PASS);
            $this->connection->setAttribute(PDO::ATTR_ERRMODE,
PDO::ERRMODE_EXCEPTION);
            //On renvoie la connexion
            return $this->connection;
        }
    }
}

```

```

    }
    //On lève une erreur si la connexion échoue
    catch(Exception $errorConnection)
    {
        die ('Erreur de connection :'. $errorConnection->getMessage());
    }
}

protected function createQuery($sql, $parameters = null)
{
    if($parameters)
    {
        $result = $this->checkConnection()->prepare($sql);
        $result->execute($parameters);
        return $result;
    }
    $result = $this->checkConnection()->query($sql);
    return $result;
}
}

```

Plus rien ne marche quand j'actualise mon navigateur web, c'est la catastrophe !

C'est normal, il faut modifier la classe `Article` en conséquence.

La classe Article

On ne peut plus utiliser la classe en l'état, parce que les méthodes actuelles ne fonctionnent plus. Voici la classe `Article`, pour rappel :

```

<?php

class Article
{
    public function getArticles()
    {
        $db = new Database();
        $connection = $db->getConnection();
        $result = $connection->query('SELECT id, title, content, author,
createdAt FROM article ORDER BY id DESC');
        return $result;
    }

    public function getArticle($articleId)
    {
        $db = new Database();
        $connection = $db->getConnection();
        $result = $connection->prepare('SELECT id, title, content, author,
createdAt FROM article WHERE id = ?');
        $result->execute([
            $articleId
        ]);
        return $result;
    }
}

```

Nous devons donc modifier notre classe `Article`

```
<?php

class Article extends Database
{
    public function getArticles()
    {
        $sql = 'SELECT id, title, content, author, createdAt FROM article
ORDER BY id DESC';
        return $this->createQuery($sql);
    }

    public function getArticle($articleId)
    {
        $sql = 'SELECT id, title, content, author, createdAt FROM article
WHERE id = ?';
        return $this->createQuery($sql, [$articleId]);
    }
}
```

Quelques explications :

- on vient d'étendre la classe `Article` avec le mot clé `extends`.
- les méthodes `getArticles` et `getArticle` ont été simplifiées, en faisant appel à la méthode `createQuery` créée précédemment.

Si vous actualisez votre page web, le résultat est le même.

Pourquoi nous avoir fait faire tout ça ? grrr

Parce que les futures classes que nous allons créer vont suivre le même fonctionnement.

On va pouvoir gérer notre future classe Comment de la même façon ?

Exactement

Finissons de refactoriser notre application en travaillant avec des... objets

Non mais oh, tu te fiches de nous là ? Ce n'est pas ce qu'on fait depuis le début ?

Suivez le guide...

Travailler avec des objets

On vient tout juste de refactoriser nos classes, mais au niveau de nos vues, ce serait bien de travailler avec des objets pour afficher nos articles, plutôt qu'avec des tableaux. Voyez-vous même ici le fichier `home.php` :

```
<?php
//On inclut le fichier dont on a besoin (ici à la racine de notre site)
require 'Database.php';
//Ne pas oublier d'ajouter le fichier Article.php
require 'Article.php';
?>

<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="utf-8">
    <title>Mon blog</title>
</head>

<body>
<div>
    <h1>Mon blog</h1>
    <p>En construction</p>
    <?php
        $article = new Article();
        $articles = $article->getArticles();
        while($article = $articles->fetch())
        {
            ?>
        }
    </div>
```



```

protected function createQuery($sql, $parameters = null)
{
    if($parameters)
    {
        $result = $this->checkConnection()->prepare($sql);
        $result->setFetchMode(PDO::FETCH_CLASS, Article::class);
        $result->execute($parameters);
        return $result;
    }
    $result = $this->checkConnection()->query($sql);
    $result->setFetchMode(PDO::FETCH_CLASS, Article::class);
    return $result;
}
}

```

Ici, on a ajouté la méthode [SetFetchMode](#) de **PDO**, en lui passant en premier paramètre le type (**PDO::FETCH_CLASS**), et en deuxième paramètre le nom de la classe.

J'ai ici indiqué la classe **Article**, mais on peut faire encore mieux pour rendre cela réutilisable.

On va lui passer le nom de la classe dynamiquement, comme ça nos futures classes pourront aussi utiliser cette même méthode 😊

Pour passer le nom de la classe qui a appelé la méthode dynamiquement, on peut utiliser la fonction [get_called_class](#) de PHP.

Si vous regardez dans la documentation, on peut même remplacer cette fonction par **static::class**.

Pour ceux qui utilisent PhpStorm, c'est aussi ce que vous conseille de faire l'IDE.

Voici notre classe **Database** actualisée

```

<?php

abstract class Database
{
    //Nos constantes
    const DB_HOST = 'mysql:host=localhost;dbname=blogg;charset=utf8';
    const DB_USER = 'root';
    const DB_PASS = '';

    private $connection;

    private function checkConnection()
    {
        //Vérifie si la connexion est nulle et fait appel à getConnection()
        if($this->connection === null) {
            return $this->getConnection();
        }
        //Si la connexion existe, elle est renvoyée, inutile de refaire une
        connexion
        return $this->connection;
    }
    //Méthode de connexion à notre base de données
    private function getConnection()
    {
        //Tentative de connexion à la base de données
        try{
            $this->connection = new PDO(self::DB_HOST, self::DB_USER,
self::DB_PASS);
            $this->connection->setAttribute(PDO::ATTR_ERRMODE,
PDO::ERRMODE_EXCEPTION);
            //On renvoie la connexion
            return $this->connection;
        }
        //On lève une erreur si la connexion échoue
        catch(Exception $errorConnection)
        {

```


```

        die ('Erreur de connection :'. $errorConnection->getMessage());
    }
}

protected function createQuery($sql, $parameters = null)
{
    if($parameters)
    {
        $result = $this->checkConnection()->prepare($sql);
        $result->setFetchMode(PDO::FETCH_CLASS, static::class);
        $result->execute($parameters);
        return $result;
    }
    $result = $this->checkConnection()->query($sql);
    $result->setFetchMode(PDO::FETCH_CLASS, static::class);
    return $result;
}
}

```

Actualisez votre page <http://localhost/blog/home.php>

Une erreur d'affiche 

Je le savais que depuis le début, tu te moquais de nous

Attendez, on n'a pas fini de refactoriser notre code encore.

Mettez à jour votre fichier **home.php**

```

<?php
//On inclut le fichier dont on a besoin (ici à la racine de notre site)
require 'Database.php';
//Ne pas oublier d'ajouter le fichier Article.php
require 'Article.php';
?>
<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="utf-8">
    <title>Mon blog</title>
</head>

<body>
<div>
    <h1>Mon blog</h1>
    <p>En construction</p>
    <?php
        $article = new Article();
        $articles = $article->getArticles();
        while($article = $articles->fetch())
        {
            var_dump($article);
            ?>
            <div>
                <h2><a href="single.php?articleId=<?=  
htmlspecialchars($article['id']);?>">?>=   
htmlspecialchars($article['title']);?></a></h2>
                <p><?=  
htmlspecialchars($article['content']);?></p>
                <p><?=  
htmlspecialchars($article['author']);?></p>
                <p>Créé le : <?=  
htmlspecialchars($article['createdAt']);?></p>
            </div>
            <br>
        <?php
    }
    $articles->closeCursor();
    ?>
</div>

```

```
</body>
</html>
```

J'ai ici ajouté un `var_dump` pour que vous puissiez voir ce que nous avons comme données... des objets

Oui, mais pourquoi une erreur s'affiche alors ?

Parce que notre syntaxe dans notre fichier est **incorrecte**.

Mettez à jour votre fichier **home.php** (pensez à supprimer le **var_dump**)

```
<?php
//On inclut le fichier dont on a besoin (ici à la racine de notre site)
require 'Database.php';
//Ne pas oublier d'ajouter le fichier Article.php
require 'Article.php';
?>
<!DOCTYPE html>
<html lang="fr">
<head>
  <meta charset="utf-8">
  <title>Mon blog</title>
</head>

<body>
<div>
  <h1>Mon blog</h1>
  <p>En construction</p>
  <?php
    $article = new Article();
    $articles = $article->getArticles();
    while($article = $articles->fetch())
    {
      ?>
      <div>
        <h2><a href="single.php?articleId=<?=  
htmlspecialchars($article->id);?>"><?=  
htmlspecialchars($article->title);?></a></h2>
        <p><?=  
htmlspecialchars($article->content);?></p>
        <p><?=  
htmlspecialchars($article->author);?></p>
        <p>Créé le : <?=  
htmlspecialchars($article->createdAt);?></p>
      </div>
      <br>
    <?php
  }
  $articles->closeCursor();
  ?>
</div>
</body>
</html>
```

Actualisez votre page, cela doit fonctionner maintenant.

Pensez à en faire de même pour notre fichier **single.php** :

```
<?php
//On inclut le fichier dont on a besoin (ici à la racine de notre site)
require 'Database.php';
//Ne pas oublier d'ajouter le fichier Article.php
require 'Article.php';
?>

<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="utf-8">
    <title>Mon blog</title>
</head>

<body>
<div>
    <h1>Mon blog</h1>
    <p>En construction</p>
    <?php
        $article = new Article();
        $articles = $article->getArticle($_GET['articleId']);
        $article = $articles->fetch();
    ?>
    <div>
        <h2><?= htmlspecialchars($article->title);?></h2>
        <p><?= htmlspecialchars($article->content);?></p>
        <p><?= htmlspecialchars($article->author);?></p>
        <p>Créé le : <?= htmlspecialchars($article->createdAt);?></p>
    </div>
    <br>
    <?php
        $articles->closeCursor();
    ?>
    <a href="home.php">Retour à l'accueil</a>
</div>
</body>
</html>
```

Il est grand temps de s'occuper de notre classe **Comment**.

Quelques révisions

Si vous avez besoin de revoir certains points, voici les liens en conséquence :

- [!\[\]\(8c4dca64662d21542001ca0ed7eeb688_img.jpg\) PDO](#)
- [!\[\]\(3de35c640e7147a3fb61ee393128d2ae_img.jpg\) VAR_DUMP](#)
- [!\[\]\(d1438aeefda19c86ae7477bf1fb30796_img.jpg\) CLASSE ABSTRAITE](#)

Bilan

Dans ce chapitre, nous avons refactorisé notre classe **Database** et adapté notre classe **Article**, ainsi que nos fichiers **home.php** et **single.php** en conséquence.

5. L'architecture et les namespaces

Il est temps de mettre en place une architecture pour être mieux organisé, avec un code plus facile à maintenir et à faire évoluer.

Mettons en place le modèle MVC !

Refactorisons notre application : étape 1

Nous allons commencer par mettre en place une architecture plus adaptée.

Mais on en a déjà une en place, elle n'est pas bien ?

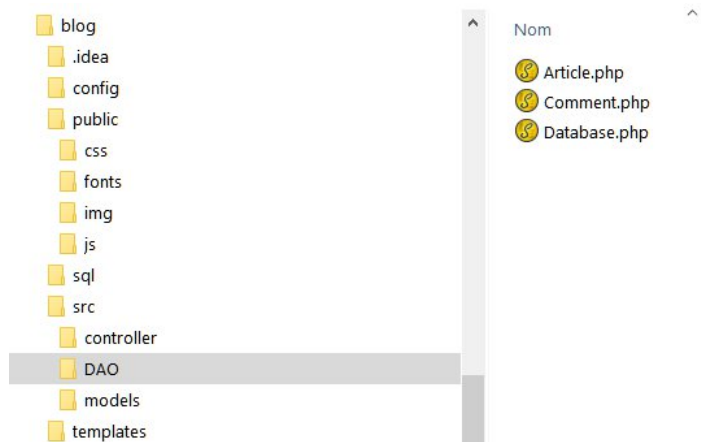
Nous devons mettre en place une architecture professionnelle, et l'actuelle risque d'être très limite.

Nous allons créer l'architecture suivante :

- un dossier `config` à la racine de notre projet, qui contiendra la configuration de notre application;
- un dossier `public`, qui sera le seul dossier accessible à l'utilisateur final, avec un dossier `css` à l'intérieur, ainsi qu'un dossier `js`, `fonts` et `img`
- un dossier `src` qui contiendra toute notre logique, où on va retrouver les dossiers `model` (contenant les classes), un dossier `controller` (pour les contrôleurs) et un dossier `DAO` (pour les managers).
- un dossier `templates` qui contiendra nos vues.

Nous allons mettre les fichiers `Article.php` et `Database.php` dans le dossier `DAO`.
On place `home.php` et `single.php` dans le dossier `templates`

Voici la nouvelle architecture :



Plus rien ne marche !

On va remédier à ceci dès maintenant.

Refactorisons notre application : étape 2

Nous allons commencer par **renommer nos classes** `Database` et `Article` :

- on renomme `Database` en `DAO`
- on renomme `Article` en `ArticleDAO`

Pensez à changer le nom de la classe et le nom du fichier

On va aussi modifier l'appel à nos fichiers sur `home.php` et `single.php` :

```
<?php
//Pour nos deux fichiers
require '../src/DAO/DAO.php';
require '../src/DAO/ArticleDAO.php';
```

Vous voyez que la refactorisation ne fait pas de mal 😊

Si vous essayez d'accéder à la page `home.php` , dans le dossier `templates` , l'affichage fonctionne correctement. Le lien vers le fichier `single.php` fonctionne très bien aussi.

Pensez à bien accéder à l'URL suivante : <http://localhost/blog/templates/home.php> (nos fichiers sont dans le dossier **templates** maintenant).

Utilisons les namespaces

Mais pourquoi ?

Dans le cas où votre projet évoluerait et que l'on pourrait être susceptible d'ajouter des librairies externes, il faut s'assurer que nos classes ne rentrent pas en conflit avec d'autres.

Il y a ici peu de choses à modifier

Ajouter dans vos fichiers `DAO.php` et `ArticleDAO.php` la ligne suivante (avant la déclaration de la classe).

Pour la classe **DAO** en particulier, ajoutez aussi la ligne indiquée :

```
<?php

//Pour toutes les classes dans DAO
namespace App\src\DAO;

//Uniquement pour la classe DAO
use PDO;
use Exception;
```

Du côté des templates, vous devez modifier légèrement vos appels aux classes comme ici

pour `home.php` :

```
<?php
require '../src/DAO/DAO.php';
require '../src/DAO/ArticleDAO.php';
?>
<!DOCTYPE html>
<html lang="fr">
<head>
  <meta charset="utf-8">
  <title>Mon blog</title>
</head>

<body>
<div>
  <h1>Mon blog</h1>
  <p>En construction</p>
  <?php
    $article = new \App\src\DAO\ArticleDAO();
    $articles = $article->getArticles();
    while($article = $articles->fetch())
    {
      ?>
      <div>
        <h2><a href="single.php?articleId=?=
htmlspecialchars($article->id);?>">?> <?>
htmlspecialchars($article->title);?></a></h2>
        <p><?> htmlspecialchars($article->content);?></p>
        <p><?> htmlspecialchars($article->author);?></p>
        <p>Créé le : <?> htmlspecialchars($article->createdAt);?></p>
```

```

        </div>
        <br>
        <?php
    }
    $articles->closeCursor();
    ?>
</div>
</body>
</html>

```

Et pour `single.php` :

```

<?php
require '../src/DAO/DAO.php';
require '../src/DAO/ArticleDAO.php';
require '../src/DAO/CommentDAO.php';
?>
<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="utf-8">
    <title>Mon blog</title>
</head>

<body>
<div>
    <h1>Mon blog</h1>
    <p>En construction</p>
    <?php
        $article = new \App\src\DAO\ArticleDAO();
        $articles = $article->getArticle($_GET['articleId']);
        $article = $articles->fetch();
    ?>
    <div>
        <h2><?= htmlspecialchars($article->title);?></h2>
        <p><?= htmlspecialchars($article->content);?></p>
        <p><?= htmlspecialchars($article->author);?></p>
        <p>Cr   le : <?= htmlspecialchars($article->createdAt);?></p>
    </div>
    <br>
    <?php
        $articles->closeCursor();
    ?>
    <a href="home.php">Retour   l'accueil</a>
</div>
</body>
</html>

```

Si vous actualisez votre navigateur, cela fonctionne toujours tr s bien

Maintenant que nous avons mis en place les **namespaces**, essayons de les utiliser correctement en simplifiant le nom de la classe lors de l'instanciation et en utilisant la directive **use** au besoin.

Voici le fichier **home.php** actualis 

```

<?php
require '../src/DAO/DAO.php';
require '../src/DAO/ArticleDAO.php';
use App\src\DAO\ArticleDAO;
?>
<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="utf-8">
    <title>Mon blog</title>
</head>

```


*Oui mais, c'est bien de faire des **require**, mais c'est répétitif, et si on en oublie un ?*

Bon d'accord, je vous l'accorde, ce n'est pas idéal, nous allons mettre en place un **autoloader** pour y remédier.

Quelques révisions

Si vous avez besoin de revoir certains points, voici les liens en conséquence :

 [NAMESPACES](#)

Bilan

Dans ce chapitre, nous avons mis en place une architecture MVC qui va nous servir pour la suite pour mieux organiser notre code. Nous en avons profité pour ajouter les namespaces à nos DAO pour faciliter l'intégration d'un **Autoloader**.

6. L'autoloader

Mettons en place un autoloader

La répétition des `require` commence à se faire sentir, et une fois que notre application va évoluer, on risque d'en oublier...

On va donc créer la classe **Autoloader** dans le fichier `Autoloader.php` (qu'on place dans le dossier `config`)

Voici notre **Autoloader**

```
<?php

namespace App\config;

class Autoloader
{
    public static function register()
    {
        spl_autoload_register([__CLASS__, 'autoload']);
    }

    public static function autoload($class)
    {
        $class = str_replace('App', '', $class);
        $class = str_replace('\\', '/', $class);
        require '../'.$class.'.php';
    }
}
```

On peut avoir quelques explications ?

Bien sûr, on a créé deux méthodes, `register()` et `autoload()`

Mais encore ? Que fait `spl_autoload_register()` ? et `str_replace()` ?

Je veux tout savoir moi ! Tout

Ok très bien : [RTFM](#) 🙄

Non mais ? Tu nous expliques ?

Je ne peux pas et ne veux pas tout vous expliquer, vous devez apprendre à chercher par vous-même et ne pas attendre qu'on vous donne toutes les explications : un bon développeur doit être curieux et autonome.

Tant que j'y pense : lien vers la documentation de PHP : <http://php.net/manual/en/>

Revenons à notre **Autoloader**, il faut maintenant l'appeler depuis nos fichiers `home.php` et `single.php`

```
<?php
require '../config/Autoloader.php';
use \App\config\Autoloader;
Autoloader::register();
```

⚠ Pensez à retirer les autres `require` sur les pages `home.php` et `single.php`.
Attention à ne pas supprimer les `use`, on en a besoin ici !
Si vous actualisez notre navigateur tout fonctionne toujours très bien

*Mais on utilise encore **require** !*

Oui, effectivement, mais on n'a plus qu'un seul **require** à faire et l'**Autoloader** charge les classes pour nous maintenant.

*Mais quand on va créer un nouveau fichier pour poster un commentaire, on va devoir encore faire un **require** de l'**autoloader** ? Pas pratique quand même ?*

Oui, je vous l'accorde. On va maintenant s'occuper de mettre en place un contrôleur frontal pour éviter ce type de répétition dans les fichiers.

Juste avant, une petite précision : dans le développement en PHP moderne, plus personne ne travaille à mettre en place un **autoloader** maison.

Pourquoi tu nous fais faire des trucs qu'on utilise plus alors ?

La réponse est simple... pour que vous puissiez comprendre ce qu'est un **autoloader** et comment cela fonctionne. Si un jour vous tombez sur un projet ancien en PHP, vous pourrez aussi comprendre le fonctionnement et le mettre à jour.

Oui, ok, mais on fait comment maintenant ? On le modifie de nouveau ?

Effectivement, on s'en occupe dans le prochain chapitre en utilisant l'**autoloader** de Composer.

Quelques révisions

Si vous avez besoin de revoir certains points, voici les liens en conséquence :

- 🔗 [Autoloader](#)
- 🔗 [spl_autoload_register](#)
- 🔗 [str_replace](#)
- 🔗 [Composer](#)

Bilan

Dans ce chapitre, nous avons ajouté un **Autoloader** pour limiter l'utilisation des **require** multiple dans notre application et faciliter l'inclusion automatique de nos classes PHP.

7. L'autoload de Composer

À ce stade du cours, Composer doit déjà être installé.

Créons notre composer.json

Pour pouvoir utiliser au mieux Composer, il faut commencer par créer un fichier **composer.json**. Dans notre cas, ce fichier sera très simple pour l'utilisation que l'on en fait, on va uniquement se servir de l'autoload de Composer.

Créez ce fichier à la racine de votre projet.

Voici notre **composer.json** :

```
{
  "autoload": {
    "psr-4": {"App\\src\\": "src/"}
  }
}
```

Ici, rien de bien complexe, on indique à Composer que l'on veut que l'autoload qui soit fait respecte le PSR-4 et l'endroit où se trouve les fichiers qui vont être appelés par l'autoloader.

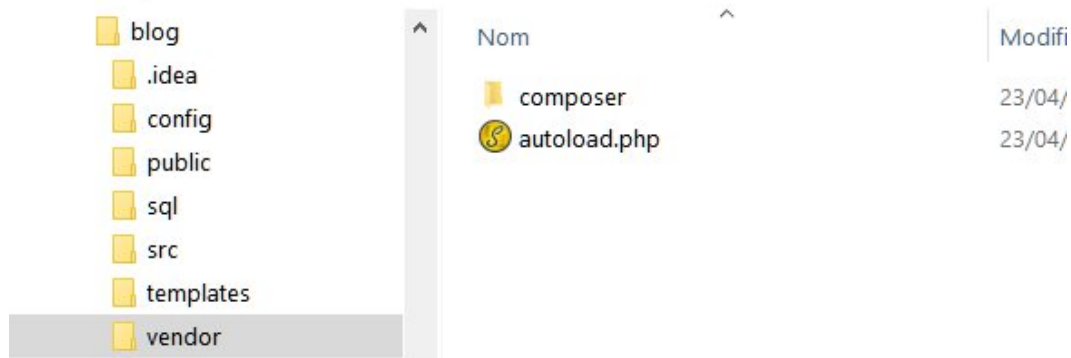
Si vous avez besoin d'en savoir plus sur le PSR-4, je vous invite suivre ce [lien](#).

Ouvrez votre terminal, placez-vous dans le répertoire de votre projet; et lancez la commande suivante :

```
composer dump-autoload
```

```
G:\Environnement\EasyPHP-Devserver-17\eds-www\BTS-SIO-Supports\blog>composer dump-autoload
Generating autoload files
Generated autoload files
G:\Environnement\EasyPHP-Devserver-17\eds-www\BTS-SIO-Supports\blog>
```

Cette commande permet aussi de générer les fichiers **d'autoload** par Composer, nous n'avons plus à nous en occuper



Il est temps de faire le ménage

Utiliser l'autoload de Composer

Ouvrez votre fichier **home.php** et mettez à jour votre fichier :

```
<?php
require '../vendor/autoload.php';

use App\src\DAO\ArticleDAO;
?>
```

Supprimez l'appel à votre fichier **Autoloader** ainsi que l'appel à la méthode **register**, et les **require** aussi.

Supprimez aussi votre fichier **Autoloader.php** dans le dossier **config**

Non mais attends, tu nous l'as fait créer pour le supprimer ?

Oui, le chapitre précédent était pour vous faire comprendre comment utiliser un **autoloader**, mais cette méthode est dépassée. Dans les applications modernes, on utilise Composer.

Faites-en de même pour **single.php** :

```
<?php
require '../vendor/autoload.php';

use App\src\DAO\ArticleDAO;
?>
```

Gardez pour le moment les **use**, on les retirera d'ici peu mais on en a encore besoin pour le moment.

Actualisez votre page web, notre blog fonctionne toujours.

Versionning

Pensez à mettre à jour votre fichier **.gitignore** :

```
.idea/
/vendor/
```

Cela permet de ne pas versionner le dossier **vendor**, qui est créé par Composer lors de l'exécution :

```
composer dump-autoload
```

```
G:\Environnement\EasyPHP-Devserver-17\eds-www\BTS-SIO-Supports\blog>composer dump-autoload
Generating autoload files
Generated autoload files
```

```
G:\Environnement\EasyPHP-Devserver-17\eds-www\BTS-SIO-Supports\blog>
```

Bilan

Dans ce chapitre, nous avons utilisé Composer pour avoir un autoloader plus performant.
(blogV1.rar)

8. Le contrôleur frontal

Quelques explications

Le contrôleur frontal va nous permettre d'intercepter toutes les requêtes et de renvoyer une réponse.

On va donc faire appel à `index.php` qui va jouer ce rôle et nous donner la page que l'on souhaite en fonction de notre besoin.

On va donc passer des paramètres à notre page `index.php` (via **GET**) et en fonction de ces paramètres, on affichera une vue associée :

- si notre `index.php` n'a pas de paramètre par défaut, on affichera la **home**
- si notre `index.php` à un paramètre article, on affichera la page **single**
- si un autre paramètre est spécifié, on affichera une erreur

index.php

On va donc créer notre fichier `index.php` (que l'on met dans le dossier `public`)

Attention à ne pas le mettre à la racine de son projet ☹☹☹

Je reviendrais plus tard sur l'intérêt de ne pas mettre ce fichier à la racine de son projet

```
<?php

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

try{
    if(isset($_GET['route']))
    {
        if($_GET['route'] === 'article'){
            require '../templates/single.php';
        }
        else{
            echo 'page inconnue';
        }
    }
    else{
        require '../templates/home.php';
    }
}
catch (Exception $e)
{
    echo 'Erreur';
}
```

On vient de centraliser l'appel à l'Autoloader dans le fichier `index.php`

Vous pouvez retirer les appels à ce dernier dans les fichiers `home.php` et `single.php`

Dans le fichier `home.php`, pensez à changer le lien

```
<?php
use App\src\DAO\ArticleDAO;
?>
<!--
Code inchangé
-->
<h2><a href="../public/index.php?route=article&articleId=<?=
htmlspecialchars($article->id);?>"><?=
htmlspecialchars($article->title);?></a></h2>
<!--
Code inchangé
-->
```

Idem dans le fichier `single.php`

```
<?php
use App\src\DAO\ArticleDAO;
?>
<!--
Code inchangé
-->
<a href="../public/index.php">Retour à l'accueil</a>
<!--
Code inchangé
-->
```

Si vous essayez de naviguer depuis l'URL `http://localhost/blog/public/index.php`, on est toujours bon.

Si cela ne fonctionne pas, assurez-vous bien d'être dans le dossier `public`.

Améliorons notre architecture

On va aussi s'occuper de créer un fichier de configuration pour notre application.

Si vous vous souvenez du fichier `DAO.php`, les identifiants de connexion sont dans le fichier, déplaçons les dans un fichier `dev.php` que l'on va créer dans le dossier `config`

```
<?php
const DB_HOST = 'mysql:host=localhost;dbname=blogg;charset=utf8';
const DB_USER = 'root';
const DB_PASS = '';
```

On peut maintenant modifier les paramètres de `PDO` dans la méthode `getConnection()` en retirant les constantes et en supprimant l'appel à `self` comme ici :

DAO.php

```
<?php

namespace App\src\DAO;

use PDO;
use Exception;

abstract class DAO
{
    private $connection;

    private function checkConnection()
    {
        //Vérifie si la connexion est nulle et fait appel à getConnection()
        if($this->connection === null) {
            return $this->getConnection();
        }
        //Si la connexion existe, elle est renvoyée, inutile de refaire une
        connexion
        return $this->connection;
    }

    //Méthode de connexion à notre base de données
    private function getConnection()
    {
        //Tentative de connexion à la base de données
        try{
            $this->connection = new PDO(DB_HOST, DB_USER, DB_PASS);
            $this->connection->setAttribute(PDO::ATTR_ERRMODE,
PDO::ERRMODE_EXCEPTION);
            //On renvoie la connexion
            return $this->connection;
        }
        //On lève une erreur si la connexion échoue
        catch(Exception $errorConnection)
```



```

        {
            die ('Erreur de connection :'. $errorConnection->getMessage());
        }
    }

    protected function createQuery($sql, $parameters = null)
    {
        if($parameters)
        {
            $result = $this->checkConnection()->prepare($sql);
            $result->setFetchMode(PDO::FETCH_CLASS, static::class);
            $result->execute($parameters);
            return $result;
        }
        $result = $this->checkConnection()->query($sql);
        $result->setFetchMode(PDO::FETCH_CLASS, static::class);
        return $result;
    }
}

```

J'en profite pour découper un peu plus ce fichier :

```

<?php

const HOST = 'localhost';
const DB_NAME = 'blogg';
const CHARSET = 'utf8';
const DB_HOST = 'mysql:host='.HOST.';dbname='.DB_NAME.';charset='.CHARSET;
const DB_USER = 'root';
const DB_PASS = '';

```

Dernière chose, on ajoute l'appel au fichier `dev.php` dans le fichier dans `index.php`

```

<?php

require '../config/dev.php';
require '../vendor/autoload.php';

try{
    if(isset($_GET['route']))
    {
        if($_GET['route'] === 'article'){
            require '../templates/single.php';
        }
        else{
            echo 'page inconnue';
        }
    }
    else{
        require '../templates/home.php';
    }
}
catch (Exception $e)
{
    echo 'Erreur';
}

```

L'autoloader ne fait pas déjà le chargement du fichier `dev.php` ?

Eh non, il s'occupe de charger les classes, `dev.php` est un fichier de configuration qui n'en contient pas.

Toujours dans le but d'avoir une architecture plus adaptée, on va maintenant travailler sur un système de routing, avec une classe associée.

Dans ce chapitre, nous avons ajouté notre contrôleur frontal, représenté par le fichier `index.php`, qui intercepte toutes les requêtes et renvoie vers la vue en conséquence. Nous en avons aussi profité pour centraliser l'appel à notre fichier de configuration, ainsi que l'appel à notre **Autoloader**.

9. Le Router

Mise en place du router

Commençons par créer une classe `Router.php` dans le dossier `config`

```
<?php

namespace App\config;
use Exception;

class Router
{
    public function run()
    {
        try{
            if(isset($_GET['route']))
            {
                if($_GET['route'] === 'article'){
                    require '../templates/single.php';
                }
                else{
                    echo 'page inconnue';
                }
            }
            else{
                require '../templates/home.php';
            }
        }
        catch (Exception $e)
        {
            echo 'Erreur';
        }
    }
}
```

On a simplifié le fichier `index.php` qui se limite au code suivant :

```
<?php

require '../config/dev.php';
require '../vendor/autoload.php';

$routeur = new \App\config\Router();
$routeur->run();
```

L'avantage ici est d'avoir une configuration pour la phase de développement.

Vous pouvez aussi créer un fichier `prod.php` dans le dossier `config` avec les mêmes constantes contenant vos paramètres de `prod`.

Quand vous passerez du mode `dev` au mode `prod`, il vous suffira de modifier la ligne dans le fichier `index.php`

On a maintenant un fichier `index.php` simplifié, et une classe `Router` qui va s'occuper de gérer toute nos routes.

Nous allons créer de nouvelles routes dans la partie suivante de ce cours

Accédez à l'URL <http://localhost/blog/public/index.php> et ... on a tout cassé...

(!) Fatal error: Uncaught Error: Class 'App\config\Router' not found in G:\Environnement\EasyPHP-Devserver-17\eds-www\BTS-SIO-Supports\blog\public\index.php on line 11			
(!) Error: Class 'App\config\Router' not found in G:\Environnement\EasyPHP-Devserver-17\eds-www\BTS-SIO-Supports\blog\public\index.php on line 11			
Call Stack			
#	Time	Memory	Function
1	0.0004	358504	{main}()
Location			
...index.php:0			
Dump \$ _SERVER			
\$ _SERVER['REMOTE_ADDR'] = G:\Environnement\EasyPHP-Devserver-17\eds-www\BTS-SIO-Supports\blog\public\index.php:11:string '127.0.0.1' (Length=9)			
\$ _SERVER['REQUEST_METHOD'] = G:\Environnement\EasyPHP-Devserver-17\eds-www\BTS-SIO-Supports\blog\public\index.php:11:string 'GET' (Length=3)			
\$ _SERVER['REQUEST_URI'] = G:\Environnement\EasyPHP-Devserver-17\eds-www\BTS-SIO-Supports\blog\public\index.php:11:string '/BTS-SIO-Supports/blog/public/' (Length=30)			
Variables in local scope (#1)			
\$router = G:\Environnement\EasyPHP-Devserver-17\eds-www\BTS-SIO-Supports\blog\public\index.php:11:*uninitialized*			

Cela vient simplement de notre configuration.

Étant donné que notre classe **Router** est dans le fichier **config**, il faut le déclarer dans le fichier **composer.json**

```
{
  "autoload": {
    "psr-4": {
      "App\\config\\": "config/",
      "App\\src\\": "src/"
    }
  }
}
```

On devra le faire à chaque nouvelle classe créée ?

Non, uniquement si les classes ne sont pas dans le dossier **src** ou **config** (si la classe est dans un sous-dossier de ceux-ci, il n'y a rien à faire).

N'oubliez pas de mettre à jour l'autoload de composer en lançant de nouveau la commande :
`composer dump-autoload`

```
G:\Environnement\EasyPHP-Devserver-17\eds-www\BTS-SIO-Supports\blog>composer dump-autoload
Generating autoload files
Generated autoload files
```

Actualisez votre page web, tout est bon

Eh mais j'y pense, tu nous a parlé de mettre en place une architecture en MVC, mais ce n'est pas encore ça n'est-ce pas ?

Effectivement, on s'y met tout de suite au prochain chapitre.

Bilan

Dans ce chapitre, nous avons ajouté un **Router** pour centraliser dans un même fichier l'appel à toutes nos routes. Ce fichier est appelé par le contrôleur frontal, **index.php** qui intercepte toutes les requêtes.

10. Controller

Notre premier contrôleur

Nous allons avoir besoin de deux contrôleurs pour notre blog :

- un **FrontController** qui va gérer ce qui est accessible à tout le monde
- un **BackController** qui permettra de gérer les fonctionnalités de l'espace d'administration, mais qui ne fait pas l'objet de ce TD.

Commençons par créer une classe **FrontController** dans le dossier **controller** dans **src** et ajoutons la méthode **home** :

```
<?php

namespace App\src\controller;

class FrontController
{
    public function home()
    {

    }
}
```

La méthode **home** va gérer l'affichage de la page d'accueil de notre site.

Oui c'est bien beau tout ça mais on en fait quoi de cette méthode ?

Nous allons faire appel à notre contrôleur depuis notre... Routeur

Et c'est justement l'appel à cette action qui va gérer le rendu de notre page d'accueil.

Mais qu'est-ce que tu racontes ? Je n'ai rien compris

Ne vous en faites pas, voici la mise en pratique associée

Router.php

```
<?php

namespace App\config;
use App\src\controller\FrontController;
use Exception;

class Router
{
    public function run()
    {
        try{
            if(isset($_GET['route']))
            {
                if($_GET['route'] === 'article'){
                    require '../templates/single.php';
                }
                else{
                    echo 'page inconnue';
                }
            }
            else{
                $frontController = new FrontController();
                $frontController->home();
            }
        }
        catch (Exception $e)
        {
            echo 'Erreur';
        }
    }
}
```

Ici, on instancie un objet **\$frontController** et on fait appel à la méthode **home**.

*Où est passé le require du fichier **home.php** ?*

Rassurez-vous, on l'a déplacé dans la méthode **home** du contrôleur que l'on vient de créer

Voyez-par vous-mêmes

FrontController.php

```
<?php

namespace App\src\controller;

class FrontController
{
    public function home()
    {
        require '../templates/home.php';
    }
}
```

Si vous actualisez votre page d'accueil, ça fonctionne toujours...

On va aller un petit peu plus loin, en limitant au maximum le code php dans nos vues.

On va retirer les deux lignes suivantes dans **home.php** :

```
<?php
$article = new ArticleDAO();
$articles = $article->getArticles();
```

et les déplacer dans la méthode **home** du **FrontController**.

Pensez à récupérer le **use** en haut du fichier **home.php** et déplacez le dans le **FrontController.php**, comme ici :

FrontController.php

```
<?php

namespace App\src\controller;
use App\src\DAO\ArticleDAO;

class FrontController
{
    public function home()
    {
        $article = new ArticleDAO();
        $articles = $article->getArticles();
        require '../templates/home.php';
    }
}
```

Si vous actualisez votre page, on est toujours bon ☺

Profitons-en pour en faire de même avec le rendu de la page single

🔗 Voici nos fichiers actualisés :

Router.php

```
<?php

namespace App\config;
use App\src\controller\FrontController;
use Exception;

class Router
{
    public function run()
    {
        try{
            if(isset($_GET['route']))
            {
```

```

        if($_GET['route'] === 'article'){
            $frontController = new FrontController();
            $frontController->article($_GET['articleId']);
        }
        else{
            echo 'page inconnue';
        }
    }
    else{
        $frontController = new FrontController();
        $frontController->home();
    }
}
catch (Exception $e)
{
    echo 'Erreur';
}
}
}

```

FrontController.php

```

<?php

namespace App\src\controller;
use App\src\DAO\ArticleDAO;

class FrontController
{
    public function home()
    {
        $article = new ArticleDAO();
        $articles = $article->getArticles();
        require '../templates/home.php';
    }

    public function article($articleId)
    {
        $article = new ArticleDAO();
        $articles = $article->getArticle($articleId);
        require '../templates/single.php';
    }
}

```

single.php

```

<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="utf-8">
    <title>Mon blog</title>
</head>

<body>
<div>
    <h1>Mon blog</h1>
    <p>En construction</p>
    <?php
        $article = $articles->fetch()
    ?>
    <div>
        <h2><?= htmlspecialchars($article->title);?></h2>
        <p><?= htmlspecialchars($article->content);?></p>
        <p><?= htmlspecialchars($article->author);?></p>
        <p>Cr   le : <?= htmlspecialchars($article->createdAt);?></p>
    </div>
    <br>
    <?php

```

```
$articles->closeCursor();  
?>  
<a href="../public/index.php">Retour à l'accueil</a>  
</div>  
</body>  
</html>
```

Dans le router, on a utilisé le même principe que tout à l'heure, mais on a fait appel à une autre méthode, `article`, qui a besoin de l'identifiant passé en paramètre. Le contrôleur instancie un objet `ArticleDAO` et fait appel aux méthodes dont on a besoin, la vue associée se contente de faire une boucle et d'afficher les informations. On a aussi remplacé dans notre Contrôleur `$_GET['articleId']` par `$articleId`.

Un peu de refactorisation

Encore ? Tu te fiches de nous ?

Je vous l'avais dit, cela arrivera régulièrement dans le développement de vos projets.

Je vais ici refactoriser le fichier `Router.php` et `FrontController.php`

Je vous invite à essayer par vous-même (vous avez dû remarquer qu'un peu de code était répété), et voir la solution si vous bloquez....

Router.php

```
<?php
namespace App\config;
use App\src\controller\FrontController;
use Exception;

class Router
{
    private $frontController;

    public function __construct()
    {
        $this->frontController = new FrontController();
    }

    public function run()
    {
        try{
            if(isset($_GET['route']))
            {
                if($_GET['route'] === 'article'){
                    $this->frontController->article($_GET['articleId']);
                }
                else{
                    echo 'page inconnue';
                }
            }
            else{
                $this->frontController->home();
            }
        }
        catch (Exception $e)
        {
            echo 'Erreur';
        }
    }
}
```

FrontController.php

```
<?php
namespace App\src\controller;
use App\src\DAO\ArticleDAO;

class FrontController
{
    private $articleDAO;

    public function __construct()
    {
        $this->articleDAO = new ArticleDAO();
    }

    public function home()
    {
        $articles = $this->articleDAO->getArticles();
        require '../templates/home.php';
    }

    public function article($articleId)
    {
        $articles = $this->articleDAO->getArticle($articleId);
        require '../templates/single.php';
    }
}
```

On commence maintenant à respecter le modèle MVC, même si nous ne travaillons pas encore tout à fait avec des objets. Je reviens là-dessus bientôt.

ErrorController, pour gérer les erreurs

Vous avez dû remarquer (ou pas ☹️) que dans notre **Router** les erreurs n'étaient pas gérées par des contrôleurs.

Pour une application optimale, il faut que tout soit géré de la même manière pour éviter de s'y perdre.

Je vous laisse essayer de mettre en place cette fonctionnalité, sachant que :

- le **Router** doit faire appel à une méthode du **ErrorController**
- chaque méthode de ce contrôleur doit renvoyer une vue
- la vue peut contenir uniquement un echo en php, ou en HTML, peu importe

À vous de jouer.

Vous avez réussi ? Vous avez bloqué sur un point ?

Pas de panique, je vous donne une solution ici :

Notre fichier **Router.php** actualisé

```
<?php

namespace App\config;
use App\src\controller\FrontController;
use App\src\controller\ErrorController;
use Exception;

class Router
{
    private $frontController;
    private $errorController;

    public function __construct()
    {
        $this->frontController = new FrontController();
        $this->errorController = new ErrorController();
    }

    public function run()
    {
        try{
            if(isset($_GET['route']))
            {
                if($_GET['route'] === 'article'){
                    $this->frontController->article($_GET['articleId']);
                }
                else{
                    $this->errorController->errorNotFound();
                }
            }
            else{
                $this->frontController->home();
            }
        }
        catch (Exception $e)
        {
            $this->errorController->errorServer();
        }
    }
}
```

Voici donc le **ErrorController.php**

```
<?php

namespace App\src\controller;

class ErrorController
{
    public function errorNotFound()
    {
        require '../templates/error_404.php';
    }

    public function errorServer()
    {
        require '../templates/error_500.php';
    }
}
```

Je ne mets pas le code des fichiers **error_404.php** et **error_500.php**, ce sont de simples messages affichant 'page non trouvée' et 'problème serveur' (mais pensez à coder ces deux pages !)

Vous vous rappelez ce que je vous ai dit tout à l'heure ? Qu'on ne travaillait pas totalement avec des objets ?

On va s'en occuper maintenant

Bilan

Dans ce chapitre, nous avons mis en place nos contrôleurs pour respecter le modèle MVC.
Nous allons maintenant nous occuper de la partie model de notre MVC.
(blogV2.rar)

11. Model

Notre premier model : Article.php

Occupons-nous maintenant de créer un model pour nos articles : une classe Article.php

Très bien, mais ça va nous servir à quoi exactement ? À travailler avec des... objets

Mais c'est déjà ce qu'on fait non ?

Pas tout à fait, on travaille effectivement avec des "objets", mais ce ne sont pas des "objets" conformes.

Vous n'avez rien compris ? Ok, je vous explique...

Ouvrez votre fichier **home.php** et faites un **var_dump** de **\$article** ...

```
<?php
while($article = $articles->fetch())
{
    var_dump($article);
    ?>
    <div>
        <h2><a href=" ../public/index.php?route=article&articleId=<?=
htmlspecialchars($article->id);?>"><?=
htmlspecialchars($article->title);?></a></h2>
        <p><?= htmlspecialchars($article->content);?></p>
        <p><?= htmlspecialchars($article->author);?></p>
        <p>Créé le : <?= htmlspecialchars($article->createdAt);?></p>
    </div>
    <br>
<?php
}
$articles->closeCursor();
?>
```

Actualisez la page <http://localhost/blog/public/index.php>

Rien ne vous "choque" ?

Mon blog

In construction

```
:\Environnement\EasyPHP-Devserver-17\eds-www\BTS-SIO-Supports\blog\templates\home.php:33:
bjeect (App\src\DAO\ArticleDAO) [9]
  private 'connection' (App\src\DAO\DAO) => null
  public 'id' => string '3' (length=1)
  public 'title' => string 'Mon troisième article' (length=22)
  public 'content' => string 'Mon blog est génial !!!' (length=24)
  public 'author' => string 'Karim' (length=5)
  public 'createdAt' => string '2019-03-16 14:45:57' (length=19)
```

Mon troisième article

Mon blog est génial !!!

Karim

Créé le : 2019-03-16 14:45:57

```
:\Environnement\EasyPHP-Devserver-17\eds-www\BTS-SIO-Supports\blog\templates\home.php:33:
bjeect (App\src\DAO\ArticleDAO) [10]
  private 'connection' (App\src\DAO\DAO) => null
  public 'id' => string '2' (length=1)
  public 'title' => string 'Un deuxième article' (length=20)
  public 'content' => string 'Je continue à ajouter du contenu sur mon blog.' (length=47)
  public 'author' => string 'Karim' (length=5)
  public 'createdAt' => string '2019-03-16 13:27:38' (length=19)
```

On a bien ici des objets, de la classe ... **ArticleDAO**

Oui et alors ? Ça fonctionne c'est le principal non ?

Oui je vous l'accorde, mais ce n'est pas suffisant.

Dans ce qu'on a mis en place dans notre application, les classes DAO ne doivent gérer **QUE** les requêtes SQL entre notre application et la base de données.

Si vous regardez de plus près, on a aussi un attribut **\$connection** qui appartient à la classe parents, la classe **DAO**.

Le mieux c'est d'avoir une classe **Article**, qui ne contienne que les propriétés et méthodes lui appartenant, qui nous crée ces objets, et non pas **ArticleDAO**.

Mais comment va-t-on faire pour transformer ça en objet ? C'est justement l'objet de ce chapitre

⚠ Vous pouvez retirer le **var_dump**.

Commençons par créer une classe **Article.php** (dans **src/model**) :

```
<?php
```

```
namespace App\src\model;
```

```
class Article
{
}
}
```

On y met quoi à l'intérieur ?

Eh bien, ses propriétés et ses méthodes :

```
<?php
```

```
namespace App\src\model;
```

```
class Article
{
    /**
     * @var int
     */
    private $id;
    /**
     * @var string
     */
    private $title;
    /**
     * @var string
     */
    private $content;
    /**
     * @var string
     */
    private $author;
    /**
     * @var \DateTime
     */
    private $createdAt;
    /**
     * @return int
     */
    public function getId()
    {
        return $this->id;
    }
}
```

```

    }
    /**
     * @param int $id
     */
    public function setId($id)
    {
        $this->id = $id;
    }
    /**
     * @return string
     */
    public function getTitle()
    {
        return $this->title;
    }
    /**
     * @param string $title
     */
    public function setTitle($title)
    {
        $this->title = $title;
    }
    /**
     * @return string
     */
    public function getContent()
    {
        return $this->content;
    }
    /**
     * @param string $content
     */
    public function setContent($content)
    {
        $this->content = $content;
    }
    /**
     * @return string
     */
    public function getAuthor()
    {
        return $this->author;
    }
    /**
     * @param string $author
     */
    public function setAuthor($author)
    {
        $this->author = $author;
    }
    /**
     * @return \DateTime
     */
    public function getCreatedAt()
    {
        return $this->createdAt;
    }
    /**
     * @param \DateTime $createdAt
     */
    public function setCreatedAt($createdAt)
    {
        $this->createdAt = $createdAt;
    }
}

```

Faisons maintenant le lien entre notre classe **Article** et **ArticleDAO**.

Pour cela, nous allons avoir besoin de plusieurs fichiers :

- **ArticleDAO.php** en récupérant le résultat de la requête et en "transformant" celle-ci en objets, basé sur la classe **Article**
- **DAO.php**, en supprimant l'appel à **SetFetchMode**, qui ne va plus nous servir
- en adaptant le fichier **home.php**, notre vue

```
<?php

namespace App\src\DAO;

use App\src\model\Article;

class ArticleDAO extends DAO
{
    private function buildObject($row)
    {
        $article = new Article();
        $article->setId($row['id']);
        $article->setTitle($row['title']);
        $article->setContent($row['content']);
        $article->setAuthor($row['author']);
        $article->setCreatedAt($row['createdAt']);
        return $article;
    }

    public function getArticles()
    {
        $sql = 'SELECT id, title, content, author, createdAt FROM article
ORDER BY id DESC';
        $result = $this->createQuery($sql);
        $articles = [];
        foreach ($result as $row) {
            $articleId = $row['id'];
            $articles[$articleId] = $this->buildObject($row);
        }
        $result->closeCursor();
        return $articles;
    }

    public function getArticle($articleId)
    {
        $sql = 'SELECT id, title, content, author, createdAt FROM article
WHERE id = ?';
        return $this->createQuery($sql, [$articleId]);
    }
}
```

On a créé une méthode privée **buildObject** qui nous permet de convertir chaque champ de la table en propriété de notre objet **Article**.

Cette méthode est appelée dans notre méthode **getArticles** pour renvoyer maintenant des objets de la classe **Article**

Dans le fichier **DAO.php**, pensez à supprimer la ligne avec **SetFetchMode** en ce qui concerne l'absence de paramètre.

On peut maintenant modifier notre vue **home.php** en conséquence

home.php

```
<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="utf-8">
    <title>Mon blog</title>
</head>
```

```

<body>
<div>
  <h1>Mon blog</h1>
  <p>En construction</p>
  <?php
    foreach ($articles as $article)
    {
      ?>
      <div>
        <h2><a href='../public/index.php?route=article&articleId=?=
htmlspecialchars($article->getId());?>">?>=
htmlspecialchars($article->getTitle());?></a></h2>
        <p>?>= htmlspecialchars($article->getContent());?></p>
        <p>?>= htmlspecialchars($article->getAuthor());?></p>
        <p>Créé le : <?>=
htmlspecialchars($article->getCreatedAt());?></p>
      </div>
      <br>
    }
  <?php
  ?>
</div>
</body>
</html>

```

Dans notre vue **home.php**, on a ici :

- remplacé notre boucle **while** par une boucle **foreach**
- fais appel à nos **getters** de la classe **Article**
- supprimer l'appel à **closeCursor** (qui est maintenant fait dans ArticleDAO.php)

À vous de jouer pour faire la même chose pour ce qui est de la méthode **getArticle** (attention, vous aurez plusieurs fichiers à mettre à jour).

Je vous mets à jour les fichiers en conséquence juste en-dessous

ArticleDAO.php

```
<?php

namespace App\src\DAO;

use App\src\model\Article;

class ArticleDAO extends DAO
{
    private function buildObject($row)
    {
        $article = new Article();
        $article->setId($row['id']);
        $article->setTitle($row['title']);
        $article->setContent($row['content']);
        $article->setAuthor($row['author']);
        $article->setCreatedAt($row['createdAt']);
        return $article;
    }

    public function getArticles()
    {
        $sql = 'SELECT id, title, content, author, createdAt FROM article
ORDER BY id DESC';
        $result = $this->createQuery($sql);
        $articles = [];
        foreach ($result as $row) {
            $articleId = $row['id'];
            $articles[$articleId] = $this->buildObject($row);
        }
        $result->closeCursor();
        return $articles;
    }

    public function getArticle($articleId)
    {
        $sql = 'SELECT id, title, content, author, createdAt FROM article
WHERE id = ?';
        $result = $this->createQuery($sql, [$articleId]);
        $article = $result->fetch();
        $result->closeCursor();
        return $this->buildObject($article);
    }
}
```

Dans le fichier DAO.php, pensez à supprimer l'autre ligne avec **setFetchMode**.

J'ai aussi mis à jour FrontController.php en renommant **\$articles** en **\$article** dans la méthode article étant donné qu'on ne renvoie qu'un seul article, cet objet étant utilisé dans la vue **single.php**.

FrontController.php :

```
<?php

namespace App\src\controller;
use App\src\DAO\ArticleDAO;

class FrontController
{
    private $articleDAO;

    public function __construct()
    {
        $this->articleDAO = new ArticleDAO();
    }
}
```

```

    }

    public function home()
    {
        $articles = $this->articleDAO->getArticles();
        require '../templates/home.php';
    }

    public function article($articleId)
    {
        $article = $this->articleDAO->getArticle($articleId);
        require '../templates/single.php';
    }
}

```

single.php :

```

<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="utf-8">
    <title>Mon blog</title>
</head>

<body>
<div>
    <h1>Mon blog</h1>
    <p>En construction</p>
    <div>
        <h2><?= htmlspecialchars($article->getTitle());?></h2>
        <p><?= htmlspecialchars($article->getContent());?></p>
        <p><?= htmlspecialchars($article->getAuthor());?></p>
        <p>Créé le : <?= htmlspecialchars($article->getCreatedAt());?></p>
    </div>
    <br>
    <a href="../public/index.php">Retour à l'accueil</a>
</div>
</body>
</html>

```

À ce stade, l'affichage des articles doit fonctionner correctement

Bilan

Dans ce chapitre, nous avons mis en place nos objets pour utiliser un maximum la programmation orientée objet.
(blogV3.rar)

12. View

Notre classe View

Nous allons créer une classe `view.php` (dans `src/model`) qui va s'occuper de la gestion de nos vues.

Voici notre classe `view`

```
<?php

namespace App\src\model;

class View
{
    private $file;
    private $title;

    public function render($template, $data = [])
    {
        $this->file = '../templates/'.$template.'.php';
        $content = $this->renderFile($this->file, $data);
        $view = $this->renderFile('../templates/base.php', [
            'title' => $this->title,
            'content' => $content
        ]);
        echo $view;
    }

    private function renderFile($file, $data)
    {
        if(file_exists($file)){
            extract($data);
            ob_start();
            require $file;
            return ob_get_clean();
        }
        header('Location: index.php?route=notFound');
    }
}
```

À partir de maintenant, je donnerais de moins en moins d'explications pour que vous puissiez aussi chercher à comprendre comment ça fonctionne

Tu nous expliques ce que fait ce fichier ?

Qu'est-ce que je viens de dire plus haut ? ☹️

Je vous laisse essayer de comprendre ce que font chacune de ces méthodesⁱ.

On va adapter nos contrôleurs en conséquence

`FrontController.php`

```
<?php

namespace App\src\controller;
use App\src\DAO\ArticleDAO;
use App\src\model\View;

class FrontController
{
    private $articleDAO;
    private $view;

    public function construct()
    {
        $this->articleDAO = new ArticleDAO();
        $this->view = new View();
    }
}
```

```

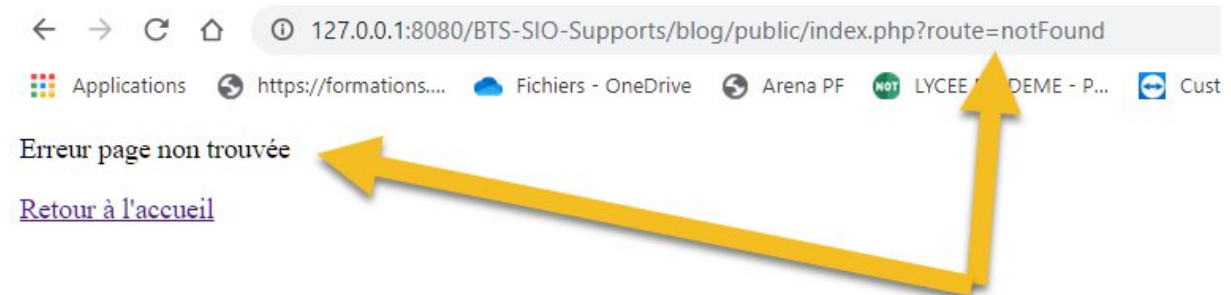
public function home()
{
    $articles = $this->articleDAO->getArticles();
    return $this->view->render('home', [
        'articles' => $articles
    ]);
}

public function article($articleId)
{
    $article = $this->articleDAO->getArticle($articleId);
    return $this->view->render('single', [
        'article' => $article,
    ]);
}
}

```

Actualisez votre page web

Ça ne marche plus !



Eh oui, à votre avis, pour quelle raison est-ce que ça ne fonctionne plus ?

Pour ceux qui ont bien regardé la classe `view`, on indique un fichier qui s'appelle `base.php`, mais celui-ci n'a jamais été créé.

Eh bien, créons-le...

Un template de base

Je pense que vous l'avez remarqué, on répète plusieurs fois dans nos vues la structure html de base.

On va donc la centraliser dans le fichier `base.php` :

```

<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="UTF-8" />
    <title><?= $title ?></title>
</head>
<body>
    <div id="content">
        <?= $content ?>
    </div>
</body>
</html>

```

On adapte nos deux fichiers `home.php` et `single.php` :

home.php

```
<?php $this->title = "Accueil"; ?>



<h1>Mon blog</h1>
<p>En construction</p>
<?php
foreach ($articles as $article)
{
    ?>
    <div>
        <h2><a href='../public/index.php?route=article&articleId=?=
htmlspecialchars($article->getId());?>"<?=
htmlspecialchars($article->getTitle());?></a></h2>
        <p><?= htmlspecialchars($article->getContent());?></p>
        <p><?= htmlspecialchars($article->getAuthor());?></p>
        <p>Créé le : <?= htmlspecialchars($article->getCreatedAt());?></p>
    </div>
    <br>
    <?php
}
?>
```

single.php

```
<?php $this->title = "Article"; ?>
<h1>Mon blog</h1>
<p>En construction</p>
<div>
    <h2><?= htmlspecialchars($article->getTitle());?></h2>
    <p><?= htmlspecialchars($article->getContent());?></p>
    <p><?= htmlspecialchars($article->getAuthor());?></p>
    <p>Créé le : <?= htmlspecialchars($article->getCreatedAt());?></p>
</div>
<br>
<a href='../public/index.php'>Retour à l'accueil</a>
```

/** * Created by PhpStorm. * User: Mircille * Date: 29/04/2021 * Time: 09:10 */

Et voilà le travail

 Je vous laisse adapter le contrôleur **ErrorController** ainsi que les vues éventuelles créés (dans le dossier  template).

Bilan

Dans ce chapitre, nous avons mis en place une classe **View** qui s'occupe de gérer nos fichiers de template.
(blogV4.rar)

Mon blog

En construction

[Mon troisième article](#)

Mon blog est génial !!!

Karim

Créé le : 2019-03-16 14:45:57

[Un deuxième article](#)

Je continue à ajouter du contenu sur mon blog.

Karim

Créé le : 2019-03-16 13:27:38

[Voici mon premier article](#)

Mon super blog est en construction.

Karim

Créé le : 2019-03-15 08:10:24

ⁱ <https://nouvelle-techno.fr/actualites/live-coding-introduction-au-mvc-en-php>