Architecture MVC en PHP*

Partie 2 (suite) : Factorisez votre code dans une architecture MVC

R3.01 : Développement web

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-creez-template.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`, à la ligne 65 :

```
$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
```

*https://openclassrooms.com/fr/courses/4670706-adoptez-une-architecture-mvc-en-php/7848047-affichez-des-commentaires

```
> git commit -m "Base pour créer un template"
> git push -u origin main
```

2 Créez un template de page

Essayons maintenant d'améliorer nos vues. Il y a du code qui se répète et, comme vous le savez, le code qui se répète... on n'aime pas ça !

2.1 Incluez des blocs de page

La première approche, que vous connaissez sûrement, consiste à créer des blocs de page dans des fichiers PHP différents. Par exemple :

- `header.php`
- `footer.php`

Avec ça, nos vues n'auraient plus qu'à inclure (avec un `include` ou un `require`) le header et le footer :

```php
<?php require('header.php'); ?>
<h1>Le super blog de l'AVBN !</h1>
<p>Contenu de la page</p>
<?php require('footer.php'); ?>
```

Alors oui, ça marche, mais il y a moyen de faire mieux et plus flexible. En effet, imaginez que le menu change un peu en fonction des pages par exemple. Comment vous faites, si ce menu se trouve dans `header.php` ?

Ou le titre de la page dans la balise `<title>` ? Vous n'y avez pas accès pour le personnaliser en fonction des pages ! A priori, tout ça est dans `header.php`...que vous ne pouvez pas changer d'ici.

Pour être plus flexible, il faut... inverser complètement notre approche.

2.2 Créez un layout

On va créer un ***layout*** (une disposition, traduit littéralement) de page. On va y retrouver **toute la structure de la page**, avec des "trous" à remplir.

Attention à ne pas faire l'amalgame entre layout et template ! Un **layout** est une façon spécifique d'utiliser un **template**. Il sert à créer une disposition d'affichage. Dans un fichier layout, les "trous" à remplir seront très souvent comblés... par des templates !

Voici notre fichier `templates/layout.php` :

```
<!DOCTYPE html>
<html>
   <head>
      <meta charset="utf-8" />
      <title><?= $title ?></title>
      <link href="style.css" rel="stylesheet" />
   </head>

   <body>
      <?= $content ?>
   </body>
</html>
```

Il y a 2 "trous" à remplir dans ce layout : le `<title>` et le contenu de la page.

Évidemment, on pourrait faire plus compliqué si on voulait (par exemple, on pourrait réserver un espace pour personnaliser le menu). Mais vous voyez l'idée : vous **créez la structure** de votre page et vous **remplissez les trous par des variables**.

Il faut maintenant définir ce qu'on met dans ces variables. Voici comment on peut le faire dans la vue `templates/homepage.php` qui affiche la liste des derniers billets :

```
<?php $title = "Le blog de l'AVBN"; ?>

<?php ob_start(); ?>
<h1>Le super blog de l'AVBN !</h1>
<p>Derniers billets du blog :</p>

<?php
foreach ($posts as $post) {
?>
        <div class="news">
            <h3>
                <?= htmlspecialchars($post['title']); ?>
                <em>le <?= $post['french_creation_date']; ?></em>
            </h3>
            <p>
                <?= nl2br(htmlspecialchars($post['content'])); ?>
                <br />
                <em>
                <a href="post.php?id=<?= urlencode($post['identifier']) ?>">
                Commentaires
                </a></em>
            </p>
```

```
        </div>
<?php
}
?>
<?php $content = ob_get_clean(); ?>

<?php require('layout.php') ?>
```

Ce code fait 3 choses :

1. Il définit le **titre** de la page dans `$title`. Celui-ci sera intégré dans la balise `<title>` dans le template.
2. Il définit le **contenu** de la page dans `$content`. Il sera intégré dans la balise `<body>` du template.
   Comme ce contenu est un peu gros, on utilise une astuce pour le mettre dans une variable. On appelle la fonction `ob_start()` (ligne 3) qui "mémorise" toute la sortie HTML qui suit. Puis, à la fin, on récupère le contenu généré avec `ob_get_clean()`(ligne 28) et on met le tout dans `$content`.
3. Enfin, il **appelle le template** avec un 'require'. Celui-ci va récupérer les variables `$title` et `$content` qu'on vient de créer... pour afficher la page.

À part l'astuce du `ob_start()` et `ob_get_clean()`(qui nous sert juste à mettre facilement beaucoup de code HTML dans une variable), le principe est simple. En inversant l'approche, on a ainsi obtenu un code bien plus flexible pour définir des "morceaux" de page dans des variables.

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

```
> git add --all
> git commit -m "Créez un template et un layout"
> git push -u origin main
```

3 Créez un routeur

Pour l'instant, 2 fichiers permettent d'accéder aux pages de notre site. Ce sont les 2 contrôleurs :

- `index.php` : accueil du site, liste des derniers billets ;
- `post.php` : affichage d'un billet et de ses commentaires.

Si on continue comme ça, on va avoir une quantité faramineuse de fichiers à la racine de notre dépôt de code : `contact.php`, `editComment.php`, ...

Plus vous ajouterez de fonctionnalités sur le blog, plus le nombre de fichiers va augmenter. Ça deviendra très difficile pour l'équipe de développeurs de s'y retrouver quand une modification sera

demandée.

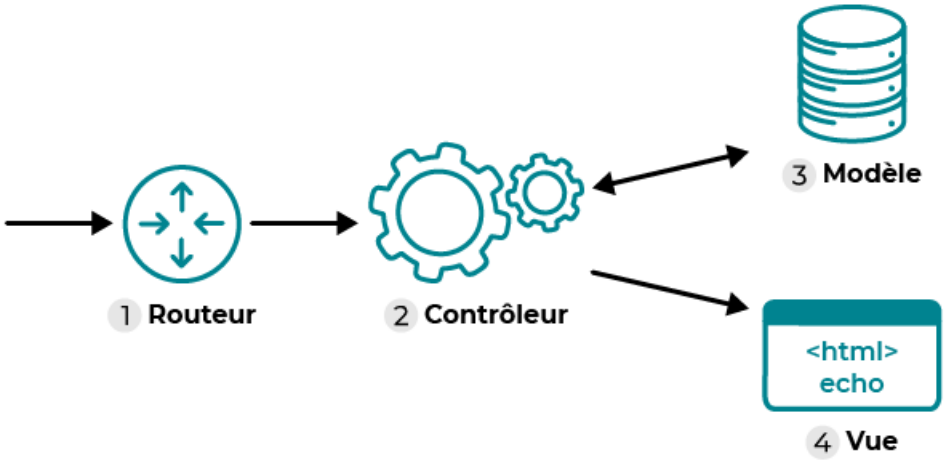
En plus de ça, on a souvent besoin d'exécuter du code en amont des contrôleurs :

- pour monitorer ce qui se passe sur votre site ;
- pour gérer les sessions utilisateurs ;
- ou encore du code "outillage", pour simplifier l'accès au contenu des requêtes HTTP.

Quand vous voudrez mettre en place ou modifier ces points, il faudra que vous passiez sur chacun des fichiers contrôleurs !

3.1 Appréhendez la nouvelle structure des fichiers

Pour faciliter la maintenance, il est plus simple de passer par un contrôleur frontal, qui va jouer le rôle de **routeur**. Son objectif va être d'**appeler le bon contrôleur** (on dit qu'il route les requêtes).



Le routeur

On va travailler ici sur deux sections de code bien distinctes :

- `index.php` : ce sera le nom de notre routeur. Le routeur étant le premier fichier qu'on appelle en général sur un site, c'est normal de le faire dans `index.php`. Il va se charger d'appeler le bon contrôleur.
- `src/controllers/` : ce dossier contiendra nos contrôleurs dans des fonctions. On va y regrouper nos anciens `index.php` et `post.php`.

Chaque contrôleur a le droit à son propre fichier. C'est une **unité de code** qui a une taille souvent suffisamment grande pour ne pas devoir la mélanger avec d'autres.

On va faire passer un paramètre `action` dans l'URL de notre routeur `index.php` pour savoir quelle page on veut appeler. Par exemple :

- `index.php` : va afficher la page d'accueil avec la liste des billets ;
- `index.php?action=post` : va afficher un billet et ses commentaires.

Certains trouvent que l'URL n'est plus très jolie sous cette forme. Peut-être préféreriez-vous voir `monsite.com/post` plutôt que `index.php?action=post`.
Heureusement, cela peut se régler avec un mécanisme de réécriture d'URL (*URL rewriting*). On ne l'abordera pas ici, car ça se fait dans la configuration du serveur web (Apache), mais vous pouvez vous renseigner sur le sujet si vous voulez !

3.2 Créez les contrôleurs

Commençons par créer notre dossier `src/controllers`. On va y créer nos contrôleurs, un par fichier :

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

require_once('src/model.php');

function homepage() {
        $posts = getPosts();

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

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

require_once('src/model.php');

function post(string $identifier)
{
        $post = getPost($identifier);
        $comments = getComments($identifier);

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

On a apporté deux changements majeurs :

- **Nos contrôleurs sont placés dans des fonctions.** Chaque fichier devient du type "bibliothèque de code" et ne fait plus rien par lui-même. Il va simplement fournir à notre routeur un point d'accès pour lancer notre code.

- **Notre fichier `src/model.php` est inclus avec `require_once`.** C'est une fonction très semblable à `require`, mais qui vérifie d'abord si le fichier a déjà été inclus ! Étant donné que `src/model.php` est aussi un fichier de type "bibliothèque de code", on souhaite qu'il ne soit inclus qu'une seule fois. Sans ça, l'inclusion de nos deux contrôleurs va déclencher une double inclusion de notre modèle et donc un plantage de PHP.

Les plus attentifs auront aussi noté qu'on a simplifié le contrôleur `post`, en lui enlevant la responsabilité de chercher lui-même l'identifiant du billet dans la requête ! On préfère déplacer ce travail sur notre routeur. Chaque contrôleur (et donc chaque nouvelle fonctionnalité métier) sera ainsi plus facile à développer.

3.3 Créez le routeur index.php

Intéressons-nous maintenant à notre routeur `index.php` :

```php
<?php

require_once('src/controllers/homepage.php');
require_once('src/controllers/post.php');

if (isset($_GET['action']) && $_GET['action'] !== '') {
        if ($_GET['action'] === 'post') {
            if (isset($_GET['id']) && $_GET['id'] > 0) {
                $identifier = $_GET['id'];

                post($identifier);
            } else {
                echo 'Erreur : aucun identifiant de billet envoyé';

                die;
            }
        } else {
            echo "Erreur 404 : la page que vous recherchez n'existe pas.";
        }
} else {
        homepage();
}
```

Il a l'air un peu compliqué parce qu'on y fait pas mal de tests, mais le principe est tout simple : **appeler le bon contrôleur**. Ça donne :

1. On charge nos contrôleurs `src/controllers/homepage.php` et `src/controllers/post.php`.
2. On teste le paramètre `action` pour savoir quel contrôleur appeler. Si le paramètre n'est pas présent, on charge le contrôleur de la page d'accueil contenant la liste des derniers billets (ligne 21).
3. On teste les différentes valeurs possibles pour notre paramètre `action` et on redirige vers le bon contrôleur à chaque fois.

3.4 A vous de terminer

Il nous reste une petite modification à faire pour que notre blog soit de nouveau utilisable. Vous avez deviné ?

Eh oui, on a changé les URL auxquelles répondait notre application, mais on n'est pas encore repassé sur les liens de l'interface !

Essayez d'apporter cette modification par vous-même. Ça vous permettra de vérifier que vous maîtrisez votre nouveau système de routage.

Vous pouvez finalement supprimer le fichier `post.php` de la racine du site.

Un bon nombre de changements on été apporté. On a créé un dossier, déplacé du code entre plusieurs fichiers, supprimé d'anciens fichiers...

Le screencast `CreezUnRouteur.asf` passe en revue toutes ces modifications pour faire le point.

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

```
> git add --all
> git commit -m "Créez un routeur"
> git push -u origin main
```

4 Ajoutez des commentaires

Cette fois, on a pour mission de permettre aux lecteurs d'ajouter des commentaires sur les billets. Que faut-il faire ?

Vous devriez commencer à avoir l'habitude. Ce sera une bonne occasion de pratiquer ! On va faire les choses dans cet ordre :

1. Modifier la vue, pour afficher le formulaire. Il a deux champs : l'auteur et le commentaire.
2. Écrire le nouveau contrôleur, qui traite les données envoyées via le formulaire.
3. Mettre à jour le routeur, pour envoyer vers le bon contrôleur.
4. Écrire le modèle. À chaque commentaire ajouté, sa date de création est sauvegardée en base automatiquement. On va aussi en profiter pour refactoriser le modèle selon le métier.

Voyons voir comment ça se passe en pratique !

4.1 Mettez à jour la vue

Il faut commencer par modifier un peu la vue qui affiche un billet et ses commentaires (`templates/post.php`). En effet, nous devons ajouter le formulaire pour pouvoir envoyer des commentaires.

```
<!-- templates/post.php:18 -->

<!-- ... -->
<h2>Commentaires</h2>

<form action="index.php?action=addComment&id=<?= $post['identifier'] ?>"
        method="post">
   <div>
          <label for="author">Auteur</label><br />
          <input type="text" id="author" name="author" />
   </div>
   <div>
          <label for="comment">Commentaire</label><br />
          <textarea id="comment" name="comment"></textarea>
   </div>
   <div>
          <input type="submit" />
   </div>
</form>
<!-- ... -->
```

Rien de spécial, c'est un formulaire.

Il faut juste bien écrire l'URL vers laquelle le formulaire est censé envoyer. Ici, vous voyez que j'envoie vers une action `addComment`. Il faudra bien penser à mettre à jour le routage.

Vous voyez aussi qu'on a besoin de `$post['identifier']` que nous n'avions pas récupéré jusque là. On va modifier notre modèle, à la ligne 31, pour aller la chercher depuis la base de données :

```
<?php
// src/model.php:31

// ...
$post = [
        'title' => $row['title'],
        'french_creation_date' => $row['french_creation_date'],
        'content' => $row['content'],
        'identifier' => $row['id'],
];
// ...
```

4.2 Écrivez le contrôleur

L'écriture de notre nouveau contrôleur va être un peu plus conséquente. Le contrôleur va contrôler les données soumises par l'utilisateur via le formulaire.

Le contrôleur va prendre en paramètres les deux **entrées utilisateur** :

- L'**identifiant du billet** auquel le commentaire doit être associé. Ce sera une chaîne de caractères.
- Les données soumises par le formulaire, sous la forme d'un tableau associatif de chaînes de caractères. Ça sera pratique, car c'est le format que PHP utilise dans la super variable `$_POST`.

Créons donc ce nouveau fichier `src/controllers/add_comment.php` :

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

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

function addComment(string $post, array $input)
{
        $author = null;
        $comment = null;
        if (!empty($input['author']) && !empty($input['comment'])) {
            $author = $input['author'];
            $comment = $input['comment'];
        } else {
            die('Les données du formulaire sont invalides.');
        }

        $success = createComment($post, $author, $comment);
        if (!$success) {
            die('Impossible d\'ajouter le commentaire !');
        } else {
            header('Location: index.php?action=post&id=' . $post);
        }
}
```

Vous noterez qu'on teste le résultat renvoyé par notre modèle. Écrire en base de données est une opération qui peut échouer, alors on demandera à notre modèle de renvoyer `true` en cas de succès ou `false` en cas d'échec. Pour résumer : on teste s'il y a eu une erreur et on arrête tout (avec un `die`) si jamais il y a un souci.

Si tout va bien, il n'y a aucune page à afficher. Les données ont été insérées, on redirige donc le visiteur vers la page du billet pour qu'il puisse voir son beau commentaire qui vient d'être inséré !

Vous remarquez qu'il nous faudra créer un nouveau fichier modèle `src/model/comment.php` où nous coderons la nouvelle fonction `createComment()`.

Nous allons d'abord nous occupez du routeur.

4.3 Mettez à jour le routeur

Il faut maintenant qu'on enregistre notre contrôleur au niveau de notre routeur. Ajoutons un `elseif` dans notre routeur `index.php` pour appeler le nouveau contrôleur `addComment` qu'on vient de créer.

```
<?php
// index.php

require_once('src/controllers/add_comment.php');
require_once('src/controllers/homepage.php');
require_once('src/controllers/post.php');

if (isset($_GET['action']) && $_GET['action'] !== '') {
        if ($_GET['action'] === 'post') {
            if (isset($_GET['id']) && $_GET['id'] > 0) {
                $identifier = $_GET['id'];

                post($identifier);
            } else {
                echo 'Erreur : aucun identifiant de billet envoyé';

                die;
            }
        } elseif ($_GET['action'] === 'addComment') {
            if (isset($_GET['id']) && $_GET['id'] > 0) {
                $identifier = $_GET['id'];

                addComment($identifier, $_POST);
            } else {
                echo 'Erreur : aucun identifiant de billet envoyé';

                die;
            }
        } else {
            echo "Erreur 404 : la page que vous recherchez n'existe pas.";
        }
} else {
        homepage();
}
```

Comme vous pouvez le voir, je teste si on a bien un ID de billet. Si c'est le cas, j'appelle le contrôleur `addComment` (ne pas oublier le `require_once('src/controllers/add_comment.php');`), qui appelle le modèle pour enregistrer les informations en base. Pour la validation des champs du

formulaire, on a donné cette responsabilité au contrôleur, alors on lui passe la variable `$_POST` directement.

Le routeur devient dur à lire avec tous ces `if` imbriqués ! Nous verrons une meilleure façon de gérer les erreurs dans la section suivante.

Quand vous commencerez à utiliser des frameworks, vous verrez que les routeurs sont vraiment très puissants. Ils permettent de ne presque pas écrire de code, tout en faisant quand même ces vérifications. Notre routeur n'en est pas encore là, mais ce ne serait qu'une question de temps si on voulait l'améliorer.

4.4 Écrivez le modèle

On va donc créer un nouveau fichier `src/model/comment.php`, où vous allez mettre la nouvelle fonction `createComment()`. On va aussi en profiter pour y **déplacer** la fonction `getComments()` qui se trouve actuellement dans le fichier `src/model.php` :

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

function getComments(string $post)
{
        $database = commentDbConnect();
        $statement = $database->prepare(
            "SELECT id, author, comment,
            DATE_FORMAT(comment_date, '%d/%m/%Y à %Hh%imin%ss')
            AS french_creation_date FROM comments WHERE post_id = ?
            ORDER BY comment_date DESC"
        );
        $statement->execute([$post]);

        $comments = [];
        while (($row = $statement->fetch())) {
            $comment = [
                'author' => $row['author'],
                'french_creation_date' => $row['french_creation_date'],
                'comment' => $row['comment'],
            ];

            $comments[] = $comment;
        }

        return $comments;
}
```

```php
function createComment(string $post, string $author, string $comment)
{
        $database = commentDbConnect();
        $statement = $database->prepare(
            'INSERT INTO comments(post_id, author, comment, comment_date)
            VALUES(?, ?, ?, NOW())'
        );
        $affectedLines = $statement->execute([$post, $author, $comment]);

        return ($affectedLines > 0);
}

function commentDbConnect()
{
        try {
            $database = new PDO('mysql:host=localhost;dbname=root;charset=root',
             'blog', 'password');

            return $database;
        } catch(Exception $e) {
            die('Erreur : '.$e->getMessage());
        }
}
```

Rien de bien folichon. Il faut juste penser à récupérer en paramètres les informations dont on a besoin :

- l'ID du billet auquel se rapporte le commentaire ;
- le nom de l'auteur ;
- le contenu du commentaire.

Le reste des informations (l'ID du commentaire, la date) sera généré automatiquement.

Comme nous avons refactorisé la manière de récupérer nos commentaires, nous devons donc mettre à jour le contrôleur `src/controllers/post.php` :

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

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

function post(string $identifier)
{
        $post = getPost($identifier);
        $comments = getComments($identifier);
```

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

Vous me direz : « Vous laissez les fichiers src/model.php et src/model/comment.php, ça ne vous pose pas de problème ? »
Ensuite, on a dupliqué la fonction `dbConnect()`, ce n'est pas idéal, non ?

Tout d'abord, le premier point. Je suis d'accord avec vous, ce n'est pas ce que je trouve de plus beau. Cependant, quand on travaille sur un projet, on doit en permanence faire des concessions. Ici, je ne touche pas aux billets de blog. D'ailleurs, la plupart du temps, je ne me souviendrais sans doute plus de comment ce code fonctionne.

Je fais donc le choix de **construire à côté de l'existant**. Quand on souhaitera refactoriser notre gestion des billets de blog, on le fera dans un autre commit. C'est une des grandes forces de ce qu'on appelle la **segmentation métier** du code.

Pour le second point, c'est un peu plus complexe. D'une part, l'unité de code "Comment" est différente de l'unité de code "Post". Ce qui affecte l'un ne doit pas forcément affecter l'autre. Ici, on peut considérer comme une coïncidence que la base de données utilisée soit la même entre ces deux bouts de code : par exemple, on pourrait très bien stocker nos commentaires via une API spécialisée.

Cela étant dit, toute la partie configuration (nom de la base, mot de passe) est ici dupliquée. Et ça, ce n'est pas une question de coïncidence. C'est une faiblesse dans notre code. Mais c'est une faiblesse qu'on résoudra beaucoup plus facilement quand on maîtrisera la programmation orientée objet et tous les outils qu'elle nous apportera.

Le screencast `AjoutezDesCommentaires.asf` reprend l'ensemble des modifications apportées dans cette section.

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

```
> git add --all
> git commit -m "Ajoutez des commentaires"
> git push -u origin main
```

5 Gérez les erreurs

La gestion des erreurs est un sujet important en programmation. Il y a souvent des erreurs et il faut savoir vivre avec. Mais comment faire ça bien ?

Si vous vous souvenez de notre routeur, il contient beaucoup de if. On fait des tests et on affiche des erreurs à chaque fois qu'il y a un problème.

Les développeurs ont en particulier du mal à gérer comme ça les erreurs qui ont lieu **à l'intérieur** des fonctions.

Que se passe-t-il s'il y a une erreur dans le contrôleur ou dans le modèle ? Va-t-on les laisser se charger d'afficher des erreurs ? Ça ne devrait normalement pas être à eux de le faire. Ils devraient **remonter** qu'il y a une erreur et laisser une partie spécialisée du code **traiter l'erreur**.

5.1 Tirez parti des exceptions

Les exceptions sont un moyen en programmation de gérer les erreurs. Vous en avez déjà vu dans du code PHP. C'est par exemple ce qu'on fait ici pour se connecter à la base de données :

```php
<?php

// Code avant

try {
        $database = new PDO('mysql:host=localhost;dbname=blog;charset=utf8',
        'root', 'root');
} catch(Exception $e) {
        die('Erreur : '.$e->getMessage());
}

// Code après
```

Pour générer une erreur, il faut "jeter une exception", ou "lancer une exception". Dès qu'il y a une erreur quelque part dans votre code, dans une fonction par exemple, vous utiliserez cette ligne :

```php
<?php
throw new Exception('Message d\'erreur à transmettre');
```

On va utiliser ce mécanisme dans notre code.

En PHP, les exceptions sont intimement liées à la programmation orientée objet. Pour l'instant, on ne va pas approfondir les détails. La prochaine grande partie du cours sera dédiée à la POO en PHP.

5.2 Ajoutez la gestion des exceptions dans le routeur

Commencez par entourer tout votre routeur par un bloc `try/catch` comme ceci :

```php
<?php
// index.php

require_once('src/controllers/add_comment.php');
require_once('src/controllers/homepage.php');
require_once('src/controllers/post.php');

try {
        if (isset($_GET['action']) && $_GET['action'] !== '') {
            if ($_GET['action'] === 'post') {
                if (isset($_GET['id']) && $_GET['id'] > 0) {
                    $identifier = $_GET['id'];

                    post($identifier);
                } else {
                    throw new Exception('Aucun identifiant de billet envoyé');
                }
            } elseif ($_GET['action'] === 'addComment') {
                if (isset($_GET['id']) && $_GET['id'] > 0) {
                    $identifier = $_GET['id'];

                    addComment($identifier, $_POST);
                } else {
                    throw new Exception('Aucun identifiant de billet envoyé');
                }
            } else {
                throw new Exception("La page que vous recherchez n'existe pas.");
            }
        } else {
            homepage();
        }
} catch (Exception $e) { // S'il y a eu une erreur, alors...
        echo 'Erreur : '.$e->getMessage();
}
```

Comme vous pouvez le voir, à l'endroit où les erreurs se produisent j'ai mis des `throw new Exception`. Cela **arrête** le bloc `try` et **amène directement** l'ordinateur au bloc `catch`.

Ici, notre bloc `catch` se contente de récupérer le message d'erreur qu'on a transmis et de l'afficher.

5.3 Remontez les exceptions

Quand il se passe une erreur à l'intérieur d'une fonction située dans le bloc`try`, celle-ci est "remontée" jusqu'au bloc `catch`.

Par exemple, notre routeur appelle la fonction du contrôleur `addComment`, que se passe-t-il quand il y a une erreur dans le contrôleur ? Notre contrôleur arrête tout et affiche ses erreurs avec des `die`. Remplaçons les `die` par des `throw new Exception()` , le code s'y arrêtera, et **les erreurs seront remontées jusque dans le routeur qui contenait le bloc** `try` :

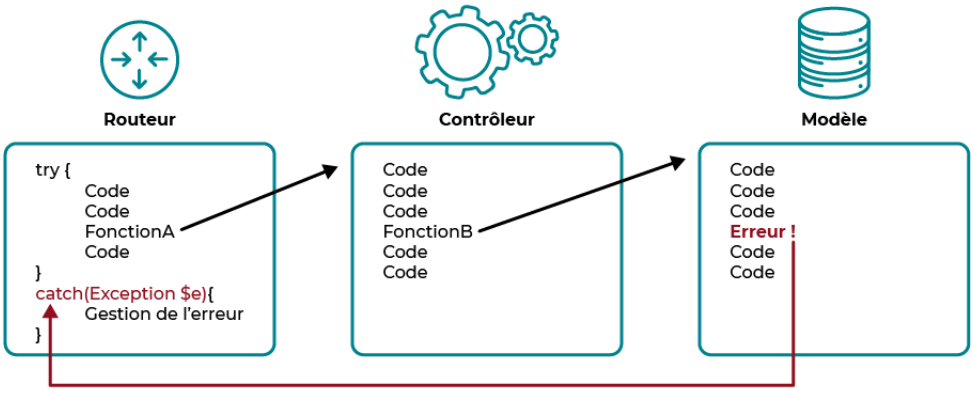
```php
<?php
// src/controllers/add_comment.php

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

function addComment(string $post, array $input)
{
        $author = null;
        $comment = null;
        if (!empty($input['author']) && !empty($input['comment'])) {
            $author = $input['author'];
            $comment = $input['comment'];
        } else {
            throw new Exception('Les données du formulaire sont invalides.');
        }

        $success = createComment($post, $author, $comment);
        if (!$success) {
            throw new Exception('Impossible d\'ajouter le commentaire !');
        } else {
            header('Location: index.php?action=post&id=' . $post);
        }
}
```

Ce principe de "remontée" de l'erreur jusqu'à l'endroit du code qui contenait le bloc try est un avantage des exceptions :



L'erreur remonte jusqu'au bloc catch

Du coup, dans la fonction `dbConnect()` de notre modèle, supprimez le bloc `try/catch` car l'erreur de connexion à la base, s'il y en a une, sera remontée jusqu'au routeur. Même chose pour la fonction `commentDbConnect()`.

Le screencast `CreezLesErreurs.asf` détaille le fonctionnement des exceptions en PHP.

5.4 A vous de terminer

Pour l'instant, notre bloc `catch` affiche une erreur avec un simple `echo`. Si nous voulons faire quelque chose de plus joli, nous pouvons appeler une vue `templates/error.php` qui affiche joliment le message d'erreur.

Il faudrait faire quelque chose comme ça :

```php
<?php
// index.php

require_once('src/controllers/add_comment.php');
require_once('src/controllers/homepage.php');
require_once('src/controllers/post.php');

try {
        // ...
} catch (Exception $e) {
        $errorMessage = $e->getMessage();

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

Je vous laisse créer la vue vous-même...

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

```
> git add --all
> git commit -m "Gérez les erreurs"
> git push -u origin main
```