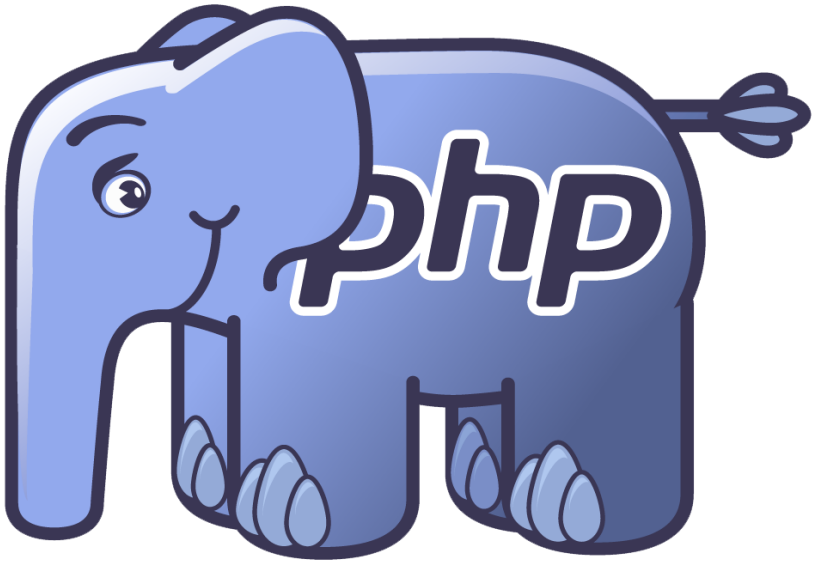


DECOUVRIR
LE PATTERN MVC
Modèle – Vue - Contrôleur

PARTIE 2

Vous allez dans cette partie structurer votre projet selon une architecture MVC, en abordant la notion de contrôleur frontal. Vous allez également mettre en pratique l'héritage et l'encapsulation notions essentielles à la POO.

Réalisation Stéphane Pontonnier

1 - Amélioration de l'architecture MVC

Mise en œuvre d'un contrôleur frontal (front controller)

L'architecture actuelle, basée sur *n* contrôleurs indépendants, souffre de certaines limitations :

- Elle expose la structure interne du site (noms des fichiers PHP) ;
- Elle rend délicate l'application de politiques communes à tous les contrôleurs (authentification, sécurité, etc.).

Pour remédier à ces défauts, il est fréquent d'ajouter au site un **contrôleur frontal**.

Le contrôleur frontal constitue le point d'entrée unique du site. Son rôle est de centraliser la gestion des requêtes entrantes. Il utilise le service d'un autre contrôleur pour réaliser l'action demandée et renvoyer son résultat sous la forme d'une vue.

Un choix fréquent consiste à transformer le fichier principal ***index.php*** en contrôleur frontal. Vous allez mettre en œuvre cette solution.

Ce changement d'architecture implique un changement d'utilisation du site. Voici comment fonctionne actuellement votre blog :

- L'exécution de **index.php** permet d'afficher la liste des 3 dernières recettes ;
- L'exécution de **recette.php?id=*<id de la recette>*** affiche les détails de la recette identifiée dans l'URL.

La mise en œuvre d'un contrôleur frontal implique que ***index.php*** recevra à la fois les demandes d'affichage de la liste des recettes et les demandes d'affichage d'une recette précise. Il faut donc lui fournir de quoi lui permettre d'identifier l'action à réaliser. Une solution courante est d'ajouter à l'URL un paramètre **action**. Dans votre exemple, voici comment ce paramètre sera interprété :

• si **action** vaut « recette », le contrôleur principal déclenchera l'affichage d'une recette ;

Toutes les actions réalisables sont rassemblées sous la forme de fonctions dans le fichier ***Controleur.php***.

```php
<?php
// Controleur.php

require 'Modele.php';

// Affiche la liste des 3 dernières recettes du blog
function accueil() {
  $recipes = threeLastRecipes();
  $comments = threeLastComments();
  require 'vueAccueil.php';
}
// Affiche les détails sur une recette
function oneRecipe($idRecipe) {
  $recipes = getRecipe($idRecipe);
  $comments = getComments($idRecipe);
  require 'vueRecette.php';
}
// Affiche une erreur
function erreur($msgErreur) {
require 'vueErreur.php';
}

?>
```

L'action à réaliser est déterminée par le fichier ***index.php*** de notre blog, réécrit sous la forme d'un contrôleur frontal.

```php
<?php
// index.php
require('Controleur.php');
try {
  if (isset($_GET['action'])) {
    if ($_GET['action'] == 'recette') {
        if (isset($_GET['id'])) {
            $idRecipe = intval($_GET['id']);
            if ($idRecipe != 0)
                oneRecipe($idRecipe);
            else
                throw new Exception("Identifiant de recette non valide");
        }
        else
            throw new Exception("Identifiant de recette non défini");
        }
     else
        throw new Exception("Action non valide");
     }
  else {
    accueil(); // action par défaut
  }
} // fin de try
catch (Exception $e) {
    erreur($e->getMessage());
}
```

Remarque : l'ancien fichier contrôleur ***recette.php*** est désormais inutile et peut être supprimé.

Enfin, le lien vers une recette doit être modifié afin de refléter la nouvelle architecture.

```php
<?php
// vueAccueil.php
...
<a href="index.php?action=recette&id=<?= $recipe['rec_id']; ?>">lien</a>
.....
?>
```

La mise en œuvre d'un contrôleur frontal a permis de préciser les responsabilités et de clarifier la dynamique de la partie **Contrôleur** de votre site :

1. Le contrôleur frontal analyse la requête entrante et vérifie les paramètres fournis ;
2. Il sélectionne et appelle l'action à réaliser en lui passant les paramètres nécessaires ;
3. Si la requête est incohérente, il signale l'erreur à l'utilisateur.

Autre bénéfice : l'organisation interne du site est totalement masquée à l'utilisateur, puisque seul le fichier ***index.php*** est visible dans les URL. Cette **encapsulation** facilite les réorganisations internes, comme celle que vous allez entreprendre maintenant.

Réorganisation des fichiers source

Par souci de simplicité, vous avez jusqu'à présent stocké tous vos fichiers source dans le même répertoire. À mesure que le site gagne en complexité, cette organisation montre ses limites. Il est maintenant difficile de deviner le rôle de certains fichiers sans les ouvrir pour examiner leur code.

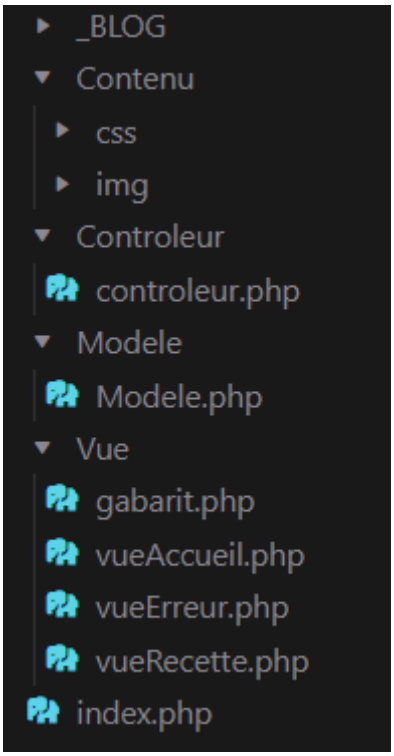
Vous allez donc restructurer votre site. La solution la plus évidente consiste à créer des sous-répertoires en suivant le découpage MVC :

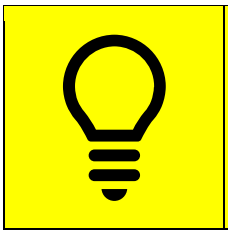
• le répertoire **Modele** contiendra le fichier ***Modele.php*** ;

• le répertoire **Vue** contiendra les fichiers ***vueAccueil.php***, ***vueRecette.php*** et ***vueErreur.php***, ainsi que la page commune ***gabarit.php*** ;

• le répertoire **Controleur** contiendra le fichier des actions ***Controleur.php***.

On peut également prévoir un répertoire **Contenu** pour les contenus statiques (fichier CSS, images, etc.) et un répertoire **BD** pour le script de création de la base de données. On aboutit à l'organisation suivante :





Rappel :
Il est évidemment nécessaire de mettre à jour les inclusions et les liens pour prendre en compte la nouvelle organisation des fichiers source.

On remarque au passage que les mises à jour sont localisées et internes : grâce au contrôleur frontal, les URL permettant d'utiliser votre site ne changent pas.

Bilan provisoire

Votre blog culinaire est maintenant structuré selon les principes du modèle MVC, avec une séparation nette des responsabilités entre composants qui se reflète dans l'organisation des sources. Votre solution est avant tout **procédurale :** les actions du contrôleur et les services du modèle sont implémentés sous la forme de fonctions.

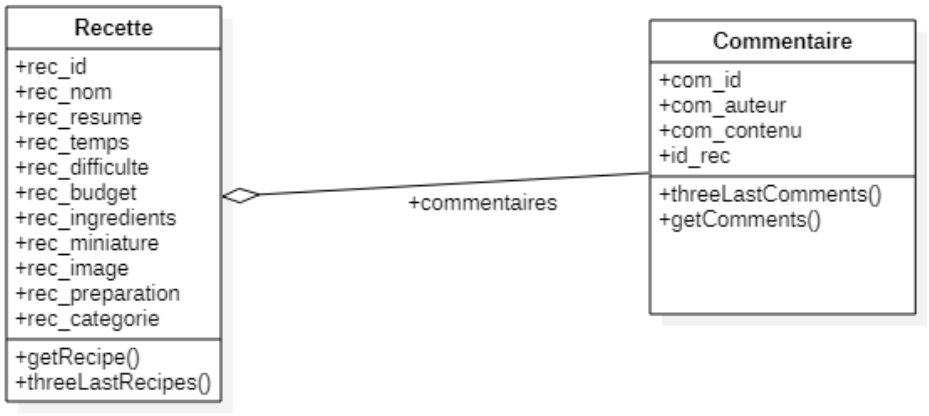
L'amélioration de l'architecture passe maintenant par la mise en œuvre des concepts de la programmation orientée objet, que PHP supporte pleinement depuis plusieurs années.

2 – Passage à une architecture MVC orientée objet

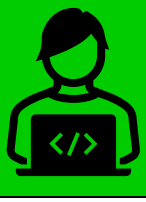
Mise en œuvre du modèle objet de PHP

Passage à un modèle orienté objet

Dans le cadre d'un passage à la POO, il serait envisageable de créer des classes métier modélisant les entités du domaine, en l'occurrence **Recette** et **Commentaire**



Plus modestement, vous allez vous contenter de définir les services d'accès aux données en tant que méthodes et non comme simples fonctions. Voici une première version de la classe **Modele**.



Modifiez votre fichier ***modele.php*** comme ci-dessous. Par rapport à votre ancien modèle procédural, la seule réelle avancée offerte par cette classe est l'encapsulation (mot-clé *private*) de la méthode de connexion à la base.

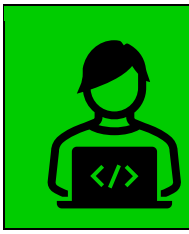
```php
<?php

// modele.php

class Modele {
  // Affiche les 3 dernières recettes

  // Affiche une recette

  // Affiche les 3 derniers commentaires

  // Affiche les commentaires pour une recette

  // Connexion à la base de données en private

}

?>
```

Rappel :
Cependant cette classe regroupe des services liés à des entités distinctes (recette et commentaire), ce qui est contraire au principe de **cohésion forte**, qui recommande de regrouper des éléments (par exemple des méthodes) en fonction de leur problématique.

Une meilleure solution consiste à créer un modèle par entité du domaine, tout en regroupant les services communs dans une super-classe commune.

On peut écrire la classe Recette, en charge de l'accès aux données des recettes, comme ci-dessous. Complétez ce script pour avoir accès à toutes vos méthodes.

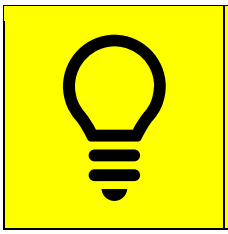
```php
<?php
// Recette.php

require_once 'Modele/Modele.php';

class Recette extends Modele {
  // Renvoie la liste des recettes du blog
  public function getRecipes() {
    $sql = 'SELECT * FROM t_recipe ....';
    $recipes = $this->executerRequete($sql);
    return $recipes;
  }
  // Renvoie les informations sur une recette
  public function getRecipe($idRecipe) {
    $sql = 'SELECT * FROM t_recipe ....';
    $recipe = $this->executerRequete($sql, array($idRecipe));
    if ($recipe->rowCount() == 1)
        return $recipe->fetch(); // Accès à la première ligne de résultat
    else
        throw new Exception("Aucune recette ne correspond à l'identifiant '$idRecipe'");
  }
}
?>
```



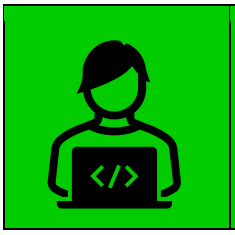
La classe **Modele** est désormais abstraite (mot-clé *abstract*) et fournit à ses classes dérivées un service d'exécution d'une requête SQL. Voir ci-dessous un exemple du fichier ***modele.php***. Adaptez le vôtre selon ce modèle.

```php
<?php
// Modele.php
abstract class Modele {
  // Objet PDO d'accès à la BD
  private $bdd;
  // Exécute une requête SQL éventuellement paramétrée
  protected function executerRequete($sql, $params = null) {
    if ($params == null) {
      $resultat = $this->getBdd()->query($sql); // exécution directe
    }
    else {
      $resultat = $this->getBdd()->prepare($sql); // requête préparée
      $resultat->execute($params);
    }
      return $resultat;
  }
  // Renvoie un objet de connexion à la BD en initialisant la connexion au besoin
  private function getBdd() {
    if ($this->bdd == null) {
        // Création de la connexion
        $this->bdd = new PDO('mysql:host=localhost;dbname=leblog;charset=utf8', 'root', '', array(PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION));
    }
    return $this->bdd;
    }
}
?>
```

On remarque au passage que la technologie d'accès à la base est totalement masquée aux modèles concrets, et que **Modele** utilise la technique du chargement tardif (« **Lazy loading** ») pour retarder l'instanciation de l'objet **$bdd** à sa première utilisation.



Recherchez ce qu'est le « lazy loading ». Décrivez en quoi il consiste.



Vous pouvez écrire une classe **Commentaire** sur le même modèle que la classe **Recette**.

À présent, l'architecture de la partie **Modele** tire parti des avantages de la POO (encapsulation, héritage). Cette architecture facilite les évolutions : si le contexte métier s'enrichit (exemple : gestion des auteurs de recettes), il suffit de créer une nouvelle classe modèle dérivée de **Modele** (ici : **Auteur**) qui s'appuiera sur les services communs fournis par sa superclasse.

Passage à une Vue orientée objet

Pour l'instant, vos vues sont des fichiers HTML/PHP qui exploitent des variables PHP contenant les données dynamiques. Elles utilisent un gabarit commun regroupant les éléments d'affichage communs. Voici par exemple la vue (simplifiée) d'affichage d'une recette et de ses commentaires.

```php
<?php
// vueRecette.php

<?php $titre = "Blog Culinaire - " . $recipe['rec_nom']; ?>
<?php ob_start(); ?>
  <article>
  <header>
  <h1 class="titreRecette"><?= $recipe['rec_nom'] ?></h1>
  <time><?= $recipe['rec_temps'] ?></time>
  </header>
  <p><?= $recipe['rec_ingredients'] ?></p>
  </article>
  <hr />
  <header>
  <h1 id="titreReponses">Réponses à <?= $recipe['titre'] ?></h1>
  </header>
  <?php foreach ($comments as $comment): ?>
    <p><?= $comment['auteur'] ?> dit :</p>
    <p><?= $comment['contenu'] ?></p>
  <?php endforeach; ?>
<?php $contenu = ob_get_clean(); ?>
<?php require 'gabarit.php'; ?>
?>
```

Le gabarit centralise les éléments d'affichage communs et utilise les variables **$titre** et **$contenu** pour intégrer les éléments spécifiques.

Cette approche simple souffre de plusieurs limitations :

- Les appels aux fonctions PHP **ob_start** et **ob_get_clean** sont dupliqués ;
- La génération des fichiers vue (y compris dans le contrôleur) utilise directement la fonction PHP **require**, sans protection contre une éventuelle absence du fichier demandé.

Vous allez créer une classe **Vue** dont le rôle sera de gérer la génération des vues.

```php
<?php
// Vue.php

class Vue {
  // Nom du fichier associé à la vue
  private $fichier;
  // Titre de la vue (défini dans le fichier vue)
  private $titre;
  public function __construct($action) {
    // Détermination du nom du fichier vue à partir de l'action
    $this->fichier = "Vue/vue" . $action . ".php";
  }
  // Génère et affiche la vue
  public function generer($donnees) {
    // Génération de la partie spécifique de la vue
    $contenu = $this->genererFichier($this->fichier, $donnees);
    // Génération du gabarit commun utilisant la partie spécifique
    $vue = $this->genererFichier('Vue/gabarit.php', array('titre' => $this->titre, 'contenu' => $contenu));
    // Renvoi de la vue au navigateur
    echo $vue;
  }
```

```php
  // Génère un fichier vue et renvoie le résultat produit
    private function genererFichier($fichier, $donnees) {
        if (file_exists($fichier)) {
          // Rend les éléments du tableau $donnees accessibles dans la vue
          extract($donnees);
          // Démarrage de la temporisation de sortie
          ob_start();
          // Inclut le fichier vue
          // Son résultat est placé dans le tampon de sortie
          require $fichier;
          // Arrêt de la temporisation et renvoi du tampon de sortie
          return ob_get_clean();
        }
        else {
            throw new Exception("Fichier '$fichier' introuvable");
        }
    }
}
?>
```

Explications :
Le constructeur de **Vue** prend en paramètre une action, qui détermine le fichier vue utilisé.
Sa méthode **generer**() génère d'abord la partie spécifique de la vue afin de définir son titre (attribut **$titre**) et son contenu (variable locale **$contenu**). Ensuite, le gabarit est généré en y incluant les éléments spécifiques de la vue. Sa méthode interne **genererFichier**() encapsule l'utilisation de **require** et permet en outre de vérifier l'existence du fichier vue à afficher.
Elle utilise la fonction **extract** pour que la vue puisse accéder aux variables PHP requises, rassemblées dans le tableau associatif **$donnees**.

Il n'est pas nécessaire de modifier le fichier gabarit. Cependant, les fichiers de chaque vue doivent être modifiés pour définir **$this->titre** et supprimer les appels aux fonctions PHP de temporisation. Voici par exemple la nouvelle vue d'accueil :

```php
<?php
// vueAccueil.php
$this->titre = "Blog Culinaire";

foreach ($recipes as $recipe):
    # code....
endforeach;

?>
```

L'affichage d'une vue se fera désormais en instanciant un objet de la classe **Vue**, puis en appelant sa méthode **generer**().

Passage à un Contrôleur orienté objet

A suivre …..