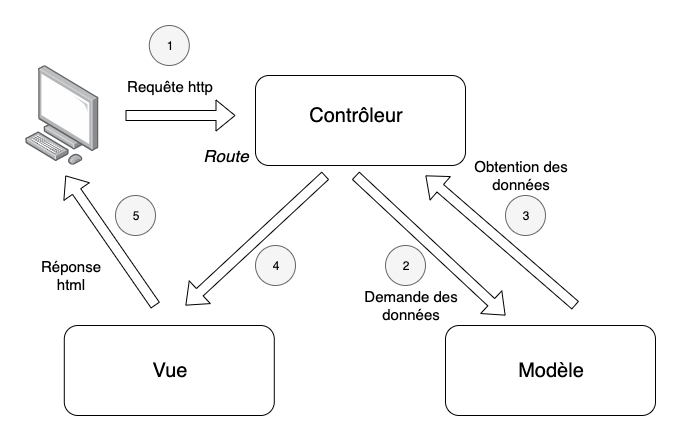
## L’ARCHITECTURE MVC

Le modèle **MVC** (Modèle-Vue-Contrôleur) est un type d’architecture (patron de conception) largement utilisé dans le développement d'applications web.

Il permet de structurer le code de manière à favoriser la séparation des responsabilités, ce qui rend l'application plus modulaire, plus facile à maintenir et plus évolutive.



Dans le contexte du développement web, MVC sépare l'application en trois composants distincts :

* **Modèle (Model)** : Représente les données de l'application et la logique métier. Il interagit avec la base de données, effectue des calculs et applique la logique métier.
* **Vue (View)** : Responsable de l'affichage de l'interface utilisateur (UI). Elle présente les données du modèle à l'utilisateur sous forme graphique, sans contenir de logique métier.
* **Contrôleur (Controller)** : Sert d'intermédiaire entre le modèle et la vue. Il reçoit les entrées de l'utilisateur (par exemple, via un formulaire ou une action sur l'interface), les traite, et met à jour la vue ou le modèle en conséquence.

L'architecture MVC permet ainsi de découpler ces trois éléments pour favoriser la réutilisabilité et la testabilité du code. Dans ce cours, nous allons explorer ces trois composants en détail et leur rôle dans la construction d'une application.

### **1. Le Modèle (Model)**

Le modèle représente la logique métier de l'application. Il contient toutes les données nécessaires à l'application et les règles qui gouvernent la manipulation de ces données.

#### **Rôle du modèle :**

* Gérer la persistance des données (enregistrement, récupération, modification, suppression).
* Appliquer la logique métier liée aux données.

**Exemple de modèle en PHP** : Imaginons une application de gestion des utilisateurs. Voici un exemple simple de modèle qui récupère des utilisateurs depuis une base de données.

<?php

class UserModel {

private $pdo;

public function \_\_construct($pdo) {

$this->pdo = $pdo;

}

// Récupérer tous les utilisateurs

public function getAllUsers() {

$sql = "SELECT \* FROM users";

$stmt = $this->pdo->prepare($sql);

$stmt->execute();

return $stmt->fetchAll(PDO::FETCH\_ASSOC);

}

// Récupérer un utilisateur par ID

public function getUserById($id) {

$sql = "SELECT \* FROM users WHERE id = :id";

$stmt = $this->pdo->prepare($sql);

$stmt->execute([':id' => $id]);

return $stmt->fetch(PDO::FETCH\_ASSOC);

}

// Ajouter un nouvel utilisateur

public function addUser($username, $email) {

$sql = "INSERT INTO users (username, email) VALUES (:username, :email)";

$stmt = $this->pdo->prepare($sql);

return $stmt->execute([':username' => $username, ':email' => $email]);

}

}

?>

#### **Points clés :**

* **Responsabilité** : Le modèle se charge des interactions avec la base de données et de la logique métier.
* **Absence de présentation** : Le modèle ne s'occupe pas de l'affichage de l'interface utilisateur.

### **2. La Vue (View)**

La vue est responsable de l'affichage des informations à l'utilisateur. Elle récupère les données traitées par le modèle et les présente de manière compréhensible et agréable pour l'utilisateur final.

#### **Rôle de la vue :**

* Présenter les données fournies par le modèle de manière graphique.
* Mettre à jour l'interface utilisateur (UI) suite à des modifications des données.
* Ne contient pas de logique métier, ni de logique de traitement des données.

**Exemple de vue en PHP** : Supposons que nous voulons afficher une liste d'utilisateurs dans une page web. Voici un exemple de vue en PHP.

<?php

class UserView {

// Afficher la liste des utilisateurs

public function renderUserList($users) {

echo "<h1>Liste des utilisateurs</h1>";

echo "<table>";

echo "<tr><th>ID</th><th>Nom d'utilisateur</th><th>Email</th></tr>";

foreach ($users as $user) {

echo "<tr>";

echo "<td>{$user['id']}</td>";

echo "<td>{$user['username']}</td>";

echo "<td>{$user['email']}</td>";

echo "</tr>";

}

echo "</table>";

}

}

?>

#### **Points clés :**

* **Affichage uniquement** : La vue ne contient aucune logique métier, elle se contente de formater et d'afficher les données.
* **Interaction avec l'utilisateur** : La vue reçoit des données du contrôleur et les affiche de manière lisible.

### **3. Le Contrôleur (Controller)**

Le contrôleur est l'intermédiaire entre le modèle et la vue. Il reçoit les requêtes de l'utilisateur, interagit avec le modèle pour récupérer ou modifier les données, puis met à jour la vue en conséquence.

#### **Rôle du contrôleur :**

* Gérer les requêtes de l'utilisateur.
* Mettre à jour le modèle en fonction des actions de l'utilisateur.
* Mettre à jour la vue avec les données appropriées.

**Exemple de contrôleur en PHP** : Supposons qu'un utilisateur soumet un formulaire pour ajouter un nouvel utilisateur. Le contrôleur interagit avec le modèle pour ajouter l'utilisateur, puis affiche une vue mise à jour.

<?php

class UserController {

private $userModel;

private $userView;

public function \_\_construct($userModel, $userView) {

$this->userModel = $userModel;

$this->userView = $userView;

}

// Afficher la liste des utilisateurs

public function showUsers() {

$users = $this->userModel->getAllUsers();

$this->userView->renderUserList($users);

}

// Ajouter un nouvel utilisateur

public function addUser($username, $email) {

if ($this->userModel->addUser($username, $email)) {

echo "Utilisateur ajouté avec succès.";

} else {

echo "Erreur lors de l'ajout de l'utilisateur.";

}

}

}

?>

#### **Points clés :**

* **Gestion des actions de l'utilisateur** : Le contrôleur récupère les requêtes de l'utilisateur (soumissions de formulaires, clics, etc.).
* **Interaction avec le modèle** : Il récupère les données via le modèle et décide de l'affichage approprié en utilisant la vue.
* **Navigation** : Le contrôleur peut rediriger vers différentes vues ou actions en fonction des besoins.

### **4. Exemple de Flux MVC**

Imaginons un scénario simple où un utilisateur visite une page qui liste tous les utilisateurs. Le processus dans une architecture MVC ressemblerait à ceci :

1. **Requête de l'utilisateur** : L'utilisateur accède à la page de la liste des utilisateurs (par exemple, /users).
2. **Contrôleur** : Le contrôleur UserController capte cette requête, appelle le modèle pour récupérer les utilisateurs via la méthode getAllUsers().
3. **Modèle** : Le modèle UserModel interroge la base de données et renvoie la liste des utilisateurs.
4. **Vue** : Le contrôleur envoie la liste des utilisateurs à la vue UserView, qui les affiche sous forme de tableau.

### **5. Avantages de l'Architecture MVC**

* **Séparation des préoccupations** : Chaque composant (modèle, vue, contrôleur) a une responsabilité clairement définie, ce qui rend le code plus facile à comprendre et à maintenir.
* **Testabilité** : Le modèle et le contrôleur peuvent être testés de manière indépendante de la vue, ce qui facilite l'écriture de tests unitaires.
* **Réutilisation** : Les vues peuvent être réutilisées avec différents modèles ou contrôleurs, et vice versa.
* **Extensibilité** : En cas de besoin, vous pouvez modifier la vue ou le modèle sans affecter les autres parties du système.

### **Conclusion**

Le modèle **MVC** est un patron de conception puissant qui permet de structurer efficacement une application en séparant les responsabilités des différents composants. En suivant l'architecture MVC, vous pouvez créer des applications modulaires, maintenables et évolutives. Ce modèle est très utilisé dans le développement web moderne, notamment avec des frameworks comme Laravel (PHP), Ruby on Rails (Ruby), Django (Python), et d'autres.

### **Exercices Pratiques**

1. **Exercice 1** : Créez un modèle pour gérer les articles dans une application de blog, et une vue pour afficher une liste d'articles.
2. **Exercice 2** : Ajoutez une fonctionnalité de mise à jour et de suppression d'articles à votre contrôleur.
3. **Exercice 3** : Utilisez des formulaires dans votre vue pour permettre aux utilisateurs de soumettre de nouveaux articles, et mettez à jour le modèle et le contrôleur pour traiter ces informations.

Ces exercices vous permettront de vous familiariser avec la séparation des responsabilités dans une architecture MVC et de comprendre l'importance de chaque composant dans le processus de développement d'une application web.

### **Implémentation d'un Contrôleur avec Twig en PHP**

Pour implémenter un **contrôleur** dans votre application PHP tout en utilisant **Twig** pour le rendu des templates, nous pouvons étendre la structure que vous avez fournie en introduisant une classe **Contrôleur** qui gère la logique métier et décide quel template Twig doit être rendu. L'objectif d'utiliser un contrôleur dans le cadre du **patron de conception MVC** est de séparer la logique métier (comme la consultation d'une base de données ou le traitement d'entrées) de la logique d'affichage (la vue).

Voici comment nous pouvons structurer votre application en introduisant un contrôleur.

### **1. Créer une classe Contrôleur**

Un **contrôleur** va gérer la logique pour les différentes routes (URLs) et rendra les templates Twig appropriés.

#### **Définition de la classe Contrôleur**

La classe **Contrôleur** va gérer la logique de routage et interagir avec Twig pour afficher les vues. Voici un exemple simple de la classe **Contrôleur**.

##### **Fichier : Controller.php**

<?php

namespace App\Controllers;

class Controller {

protected $twig;

public function \_\_construct($twig) {

$this->twig = $twig;

}

public function home() {

// Logique pour la page d'accueil (optionnelle, comme récupérer des données depuis un modèle)

echo $this->twig->render('home.twig');

}

public function crew() {

// Logique pour la page "équipe" (optionnelle)

echo $this->twig->render('crew.twig');

}

public function destination() {

// Logique pour la page "destination" (optionnelle)

echo $this->twig->render('destination.twig');

}

public function technology() {

// Logique pour la page "technologie" (optionnelle)

echo $this->twig->render('technology.twig');

}

public function notFound() {

// Page 404

echo $this->twig->render('404.twig');

}

}

Dans cet exemple, la classe **Controller** contient plusieurs méthodes correspondant à des routes spécifiques (comme la page d'accueil, l'équipe, les destinations, etc.). Chaque méthode utilise Twig pour rendre le template correspondant.

### **2. Réécrire le script principal pour utiliser le Contrôleur**

Au lieu de gérer directement la logique de routage dans le script principal, vous pouvez désormais créer une instance de la classe **Contrôleur** et déléguer la responsabilité de rendre les vues à cette classe.

#### **Fichier : index.php (Point d'entrée principal)**

<?php

require\_once('./vendor/autoload.php');

use App\Controllers\Controller;

// Configuration de Twig

$loader = new \Twig\Loader\FilesystemLoader('./views');

$twig = new \Twig\Environment($loader);

// Créer une instance du contrôleur

$controller = new Controller($twig);

// Récupérer l'URI de la requête actuelle

$request = $\_SERVER['REQUEST\_URI'];

// Logique de routage

switch ($request) {

case '':

case '/':

$controller->home();

break;

case '/crew':

$controller->crew();

break;

case '/destination':

$controller->destination();

break;

case '/technology':

$controller->technology();

break;

default:

// Si la route n'est pas trouvée, appeler la méthode 404

http\_response\_code(404);

$controller->notFound();

}

Ici, le fichier index.php utilise la classe **Controller** pour gérer les routes. Selon l'URL demandée ($request), le contrôleur appelle la méthode correspondante pour rendre la vue appropriée. Si l'URL n'est pas reconnue, il renvoie une page d'erreur 404.

### **3. Gestion des données dynamiques dans le contrôleur**

Les contrôleurs peuvent être étendus pour passer des données dynamiques depuis le **modèle** ou d'autres logiques dans les vues. Par exemple, si vous voulez récupérer des informations dynamiques pour afficher la page "équipe", vous pourriez faire quelque chose comme ceci :

public function crew() {

// Exemple de données pour la page "équipe"

$crewMembers = [

['name' => 'John Doe', 'role' => 'Capitaine'],

['name' => 'Jane Smith', 'role' => 'Ingénieur']

];

// Rendre le template 'crew.twig' et passer les données

echo $this->twig->render('crew.twig', ['crewMembers' => $crewMembers]);

}

Ensuite, dans votre template crew.twig, vous pouvez accéder à ces données de la manière suivante :

<h1>Rencontrez l'équipe</h1>

<ul>

{% for member in crewMembers %}

<li>{{ member.name }} - {{ member.role }}</li>

{% endfor %}

</ul>

### **4. Avantages de l'utilisation d'un contrôleur**

* **Séparation des responsabilités** : Le contrôleur gère toute la logique métier et le routage, ce qui permet de séparer la logique d'affichage (dans les vues) et la gestion des données (dans le modèle).
* **Extensibilité** : Lorsque votre application grandit, vous pouvez ajouter de nouvelles méthodes au contrôleur pour gérer de nouvelles fonctionnalités (comme la gestion de formulaires, la consultation de bases de données, etc.) sans alourdir le fichier de routage principal.
* **Réutilisabilité** : Vous pouvez réutiliser un contrôleur pour plusieurs routes et passer différentes données aux vues.

### **Résumé**

1. **Classe Contrôleur** : Nous avons créé une classe **Controller** pour gérer le rendu des vues selon les différentes routes.
2. **Routage** : Le fichier index.php utilise un **switch** pour appeler la méthode appropriée du contrôleur en fonction de l'URL.
3. **Données dynamiques** : Le contrôleur peut passer des données dynamiques aux templates Twig pour personnaliser l'affichage en fonction du contenu.

Cette approche permet de structurer votre application de manière plus propre et modulaire, et facilite la maintenance et l'extension du projet. L'architecture MVC permet également de mieux organiser votre code et de séparer les différentes préoccupations de votre application.