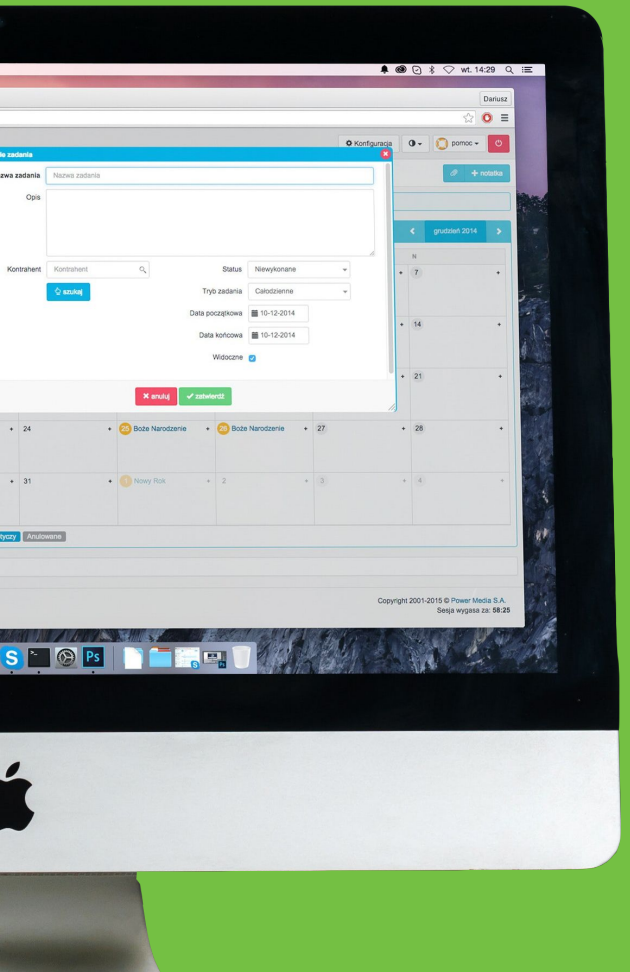


5CS045

Full Stack Development

Full Stack Development

Lecture Week 8

Template Systems & MVC Architecture

This week's agenda

- Web Template System
    - Why Template Systems
    - Twig Template Engine
    - Blade Template Engine
- Software Architectural Pattern
    - Separation of Concern
    - MVC Architecture
    - Structuring Project using MVC

Problem with default coding approach

- When building dynamic PHP websites, HTML and PHP logic are often written together in the same file.

```
<h1>User List</h1>

<p>Total users: <?php echo count($users); ?></p>

<ul>
<?php
foreach ($users as $user) {
    if ($user['active']) {
        echo "<li>" . $user['name'] . "</li>";
    }
}
?>
</ul>
}
```

Problem with default coding approach

- Mixing HTML, PHP, and SQL in one file creates a tangled, inefficient codebase.
- Code maintenance is hard as design changes can break the logic.
- Security risks increase since htmlspecialchars() is often forgotten.
- Code reuse is difficult with duplicated HTML across files.
- Testing logic is tough without separating it from HTML.
- Team collaboration issues arise between designers and developers.

Why Template Systems?

- Template systems SEPARATE the logic (PHP) from the presentation (HTML).
- They provide a **structured way** to output dynamic content while maintaining clean, readable templates that can be safely edited by front-end developers without breaking backend functionality.
- We will look at two popular templating systems in PHP

  Twig Templating System

  Blade Templating System

Twig Template Engine

- Twig is created by SensioLabs, it's part of the Symfony framework but works standalone as well.
- Twig template uses the .twig file extension.
- The recommended way to install Twig is via Composer.

```
composer require "twig/twig:^3.0"
```

Twig improves code maintainability, enhances security with auto-escaping, and supports reusability through inheritance.

1. First, Install Composer

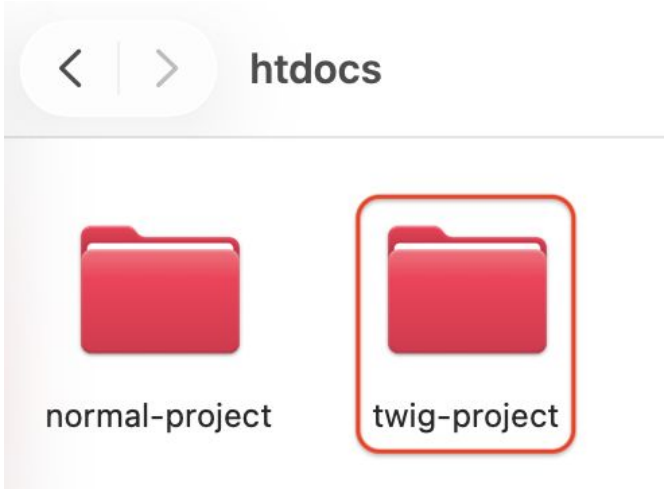
Composer is the Dependency Manager for PHP

It allows you to declare the libraries your project depends on and it will (install/update) them

```
➜  ~ php -r "copy('https://getcomposer.org/installer', 'composer-setup.php');"
php -r "if (hash_file('sha384', 'composer-setup.php') === 'c8b085408188070d5f52bcfe4ecfbee5f727afa458b2573b8eaaf77b3419b0bf2768dc67c869
44da1544f06fa544fd47') { echo 'Installer verified'.PHP_EOL; } else { echo 'Installer corrupt'.PHP_EOL; unlink('composer-setup.php'); ex
it(1); }"
php composer-setup.php
php -r "unlink('composer-setup.php');"
Installer verified
All settings correct for using Composer
Downloading...

Composer (version 2.9.2) successfully installed to: /Users/sarayugautam/composer.phar
Use it: php composer.phar

➜  ~ sudo mv composer.phar /usr/local/bin/composer

Password:
```

2. Then, create an empty project

(This is the final project structure)




```
twig-project/
├── composer.json
├── composer.lock
├── index.php
├── vendor/
│   └── (Composer dependencies)
└── templates/
    ├── base.twig
    └── example.twig
```

3. Finally, install Twig inside that project

```
➜  ~ composer require "twig/twig:^3.0"

./composer.json has been created
Running composer update twig/twig
Loading composer repositories with package information
Updating dependencies
Lock file operations: 4 installs, 0 updates, 0 removals
  - Locking symfony/deprecation-contracts (v3.6.0)
  - Locking symfony/polyfill-ctype (v1.33.0)
  - Locking symfony/polyfill-mbstring (v1.33.0)
  - Locking twig/twig (v3.22.2)
Writing lock file
Installing dependencies from lock file (including require-dev)
Package operations: 4 installs, 0 updates, 0 removals
  - Downloading symfony/polyfill-mbstring (v1.33.0)
  - Downloading symfony/polyfill-ctype (v1.33.0)
  - Downloading symfony/deprecation-contracts (v3.6.0)
  - Downloading twig/twig (v3.22.2)
  - Installing symfony/polyfill-mbstring (v1.33.0): Extracting archive
  - Installing symfony/polyfill-ctype (v1.33.0): Extracting archive
  - Installing symfony/deprecation-contracts (v3.6.0): Extracting archive
  - Installing twig/twig (v3.22.2): Extracting archive
Generating autoload files
4 packages you are using are looking for funding.
```

Example

example.twig

```twig
{% extends 'base.twig' %}  {# Inheritance #}

{% block content %}
    <h1>Hello, {{ name|upper }}!</h1>  {# Variable & Filter #}

    {% if users|length > 0 %}  {# Conditional #}
        <ul>
            {% for user in users %}  {# Loop #}
                <li>{{ user.name }} ({{ user.age }} years)</li>
            {% endfor %}
        </ul>
    {% else %}
        <p>No users found.</p>
    {% endif %}
{% endblock %}
```

Example

base.twig

```
<html><body>{% block content %}{% endblock %}</body></html>
```

index.php

```
$twig->render('example.twig', ['name' => 'world',
'users' =>
    [['name' => 'Ram', 'age' => 30]]]);
```

What makes Twig better

The PHP language is verbose and becomes ridiculously verbose when it comes to output escaping

```
<?php echo $var ?>
<?php echo htmlspecialchars($var, ENT_QUOTES, 'UTF-8') ?>
```

In comparison, Twig has a very concise syntax, which make templates more readable

```
{{ var }}
{{ var|escape }}
{{ var|e }}          {# shortcut to escape a variable #}
```

What makes Twig better

Twig has shortcuts for common patterns, like having a default text displayed when you iterate over an empty array

```
{% for user in users %}
    * {{ user.name }}
{% else %}
    No users have been found.
{% endfor %}
```

The syntax is easy to learn and has been optimized to allow web designers to get their job done fast

What makes Twig better

Twig supports everything you need to build powerful templates with ease. It supports multiple inheritance, blocks, automatic output-escaping, and much more.

```
{% extends "layout.html" %}

{% block content %}
    Content of the page...
{% endblock %}
```

What makes Twig better

Twig is secure. To be on the safe side, you can enable automatic output escaping globally or for a block of code.

```
{% autoescape "html" %}
    {{ var }}
    {{ var|raw }}     {# var won't be escaped #}
    {{ var|escape }}  {# var won't be doubled-escaped #}
{% endautoescape %}
```


Any Questions

Blade Template Engine

- Blade is Laravel's built-in templating engine, designed to be simple yet powerful. It can still be used in standalone PHP projects.
- Blade templates are compiled into plain PHP code and cached, offering excellent performance.
- Blade templates use the .blade.php file extension.

- Unlike Twig, Blade allows **seamless inline PHP execution** when needed, thereby offering greater flexibility without sacrificing readability.

Syntax Comparison

Blade's syntax is more intuitive

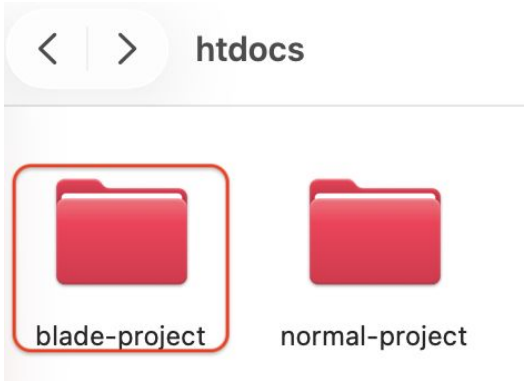
```blade
@foreach($users as $user)
    @if($user->isActive())
        <li>{{ $user->name }}</li>
    @endif
@endforeach

{{-- Need raw PHP? No problem: --}}
@php
    $customVar = array_map('strtoupper', $names);
@endphp
```

Twig Syntax

```twig
{% for user in users %}
    {% if user.active %}
        <li>{{ user.name }}</li>
    {% endif %}
{% endfor %}

{# No direct PHP—must use custom functions/filters in PHP controller #}
```

Installing Blade

- Similar to Twig, Blade is
  - Installed **inside the required project** using **Composer.**
  - `composer require jenssegers/blade`




```
→  blade-project composer require jenssegers/blade

./composer.json has been created
Running composer update jenssegers/blade
Loading composer repositories with package information
Updating dependencies
Lock file operations: 25 installs, 0 updates, 0 removals
  - Locking carbonphp/carbon-doctrine-types (3.2.0)
  - Locking doctrine/inflector (2.1.0)
  - Locking illuminate/bus (v11.47.0)
  - Locking illuminate/collections (v11.47.0)
  - Locking illuminate/conditionable (v11.47.0)
```

Instantiating the Blade Engine

```php
// After installing with: composer require jenssegers/blade

// Include Composer's autoloader
require 'vendor/autoload.php';

// Import the Blade class
use Jenssegers\Blade\Blade;

// Create the Blade engine instance
$blade = new Blade('views', 'cache');
```

Basic Blade Syntax: Displaying Data

We use double curly braces to show variables. {{}} automatically **escapes output** making the syntax **secure by default** (prevents XSS).

```
<!-- In your template: welcome.blade.php -->
<h1>Hello, {{ $name }}!</h1>
```

```
<!-- In your PHP controller or route -->
echo $blade->render('welcome', ['name' => 'Ram']);
```

Basic Blade Syntax: Conditional Statements

We use @ sign with conditionals. No PHP tags are needed.

Just write @if, @else, @elseif, @endif

```
@if($isLoggedIn)
  <p>Welcome back!</p>
@else
  <p>Please log in.</p>
@endif
```

Basic Blade Syntax: Loops

Display lists easily with @foreach

```
<ul>
  @foreach($users as $user)
    <li>{{ $user }}</li>
  @endforeach
</ul>
```

Basic Blade Syntax: Including Other Templates

Reuse parts like headers/footers with @include

```
<!-- index.blade.php -->
@include('includes.header')
<section>Page content</section>
@include('includes.footer')
```

```
views/
├── index.blade.php
└── includes/
    ├── header.blade.php
    └── footer.blade.php
```

Basic Blade Syntax: Template Inheritance using Layouts

Use a master layout. Then, extend it in other pages.

```
//layout.blade.php

<!DOCTYPE html>
<html>
<head><title>App</title></head>
<body>
  @yield('content')
</body>
</html>
```

```
//dashboard.blade.php

@extends('layout')
@section('content')
  <h1>My Dashboard</h1>
@endsection
```


Any Questions

Software Architecture:

Patterns, Separation of Concerns & MVC

Building Maintainable, Scalable Systems

What Are Software Architectural Patterns?

Reusable blueprints for **organizing code** to solve **common design problems**.

- Architectural Patterns provide high-level structure (not a detailed code).
- They are proven solutions to recurring challenges in software development.

They address **common software design problems** like:

How do I separate UI from business and data logic? → Use MVC pattern

How do I handle thousands of users? → Use Client-Server pattern

How do I make parts of my system replaceable? → Use Layered Architecture

Why Patterns Matter?

- Architectural patterns make software easier to build, understand, and maintain.
- They make workflow **predictable**. Teams use consistent, well-defined steps processes for building and changing code.
- Using **standard structure** means developers can quickly understand how a project is organized, even if they didn't write it.
- It also **reduces ambiguity**. For example in MVC, developers know exactly where to implement business logic, how to process user input, and how to manage UI updates.

Example Folder Structure

Without Architectural Patterns

```
project/
├── login.php
├── dashboard.php
├── db.php
├── user.php
├── styles.css
└── script.js
```

With Layered Architecture Pattern

```
project/
├── presentation/
│   ├── login.php
│   └── dashboard.php
│
├── logic/
│   └── database.php
│
├── data/
│   └── User.php
│
└── public/
    ├── css/
    └── js/
```

Key Architectural Pattern Examples

- **Layered** (n-Tier): Separates concerns into distinct layers (e.g., Presentation, Business Logic, Data Access).
- **Client-Server**: Distributes responsibilities between service requesters (clients) and service providers (servers).
- **Model-View-Controller** (MVC): Separates an application into three interconnected components.
- **Microservices**: Structures an application as a collection of loosely coupled, independently deployable services.

Principle: Separation of Concerns (SoC)

- **SoC** principle implies that a system should be designed by breaking it into distinct sections, where each section addresses a separate, specific concern.
- This principle **reduces complexity** and **improves maintainability** by minimizing dependencies between different parts of the system.
- Components can be tested in isolation (unit tests) and teams can work on different components simultaneously.
- Most importantly, the code is easier to **understand** and **navigate**.

MVC Architecture Pattern

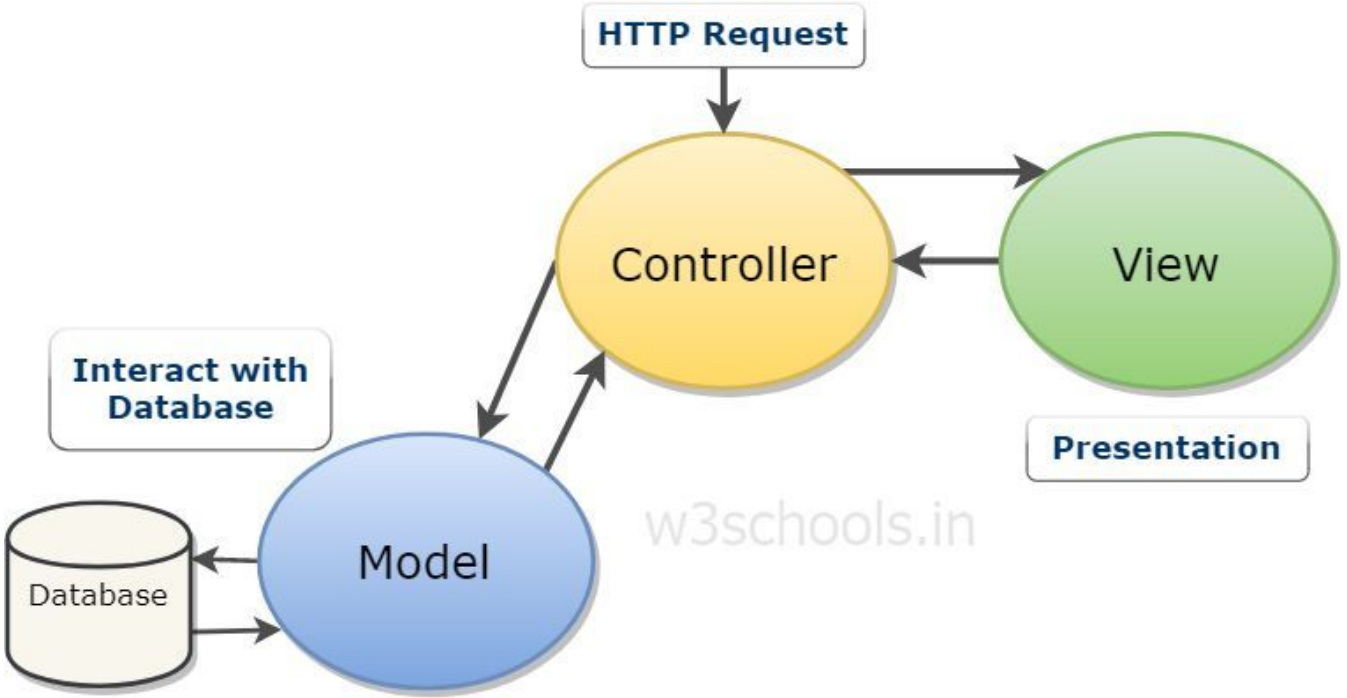
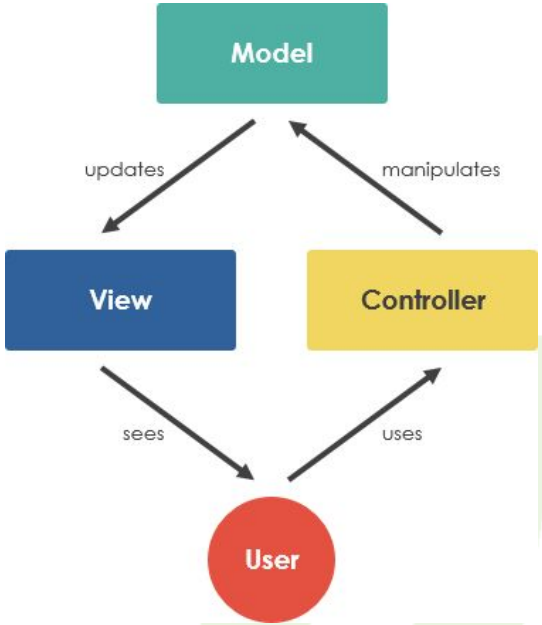


Fig: MVC Architecture

Introducing MVC Architecture

- A design pattern that implements SOC by dividing an application into three interconnected parts
  - **M**odel (Data & Business Logic)
  - **V**iew (Presentation)
  - **C**ontroller (Intermediary user input handler)
- The goal is to separate the internal **representation of data** from how it is **presented to** and **accepted from** the user.



Component 1: The Model

- Model manages the application's data, logic, and rules.
- It represents the core knowledge and state of the application.
- It acts as the **single source of truth** for the system's data and schema.
- When **data changes**, Model **notifies** the **View**.
- Examples: User data model, product inventory model, app state model, game logic and level calculation algorithms etc.

The Model knows everything about the data, but nothing about how it is displayed or controlled.

Component 2: The View

- View is responsible for **presenting data** to the user. It represents the user interface of the application.
- View does not contain business logic or data manipulation. It **reads data from the Model** and displays it in an appropriate format..
- It is updated when the **Controller re-renders it** using the latest data from the Model.
- Examples: Web pages, HTML templates, UI screens etc.

The View displays the data, but does not know where it comes from or how it is processed.

Component 3: The Controller

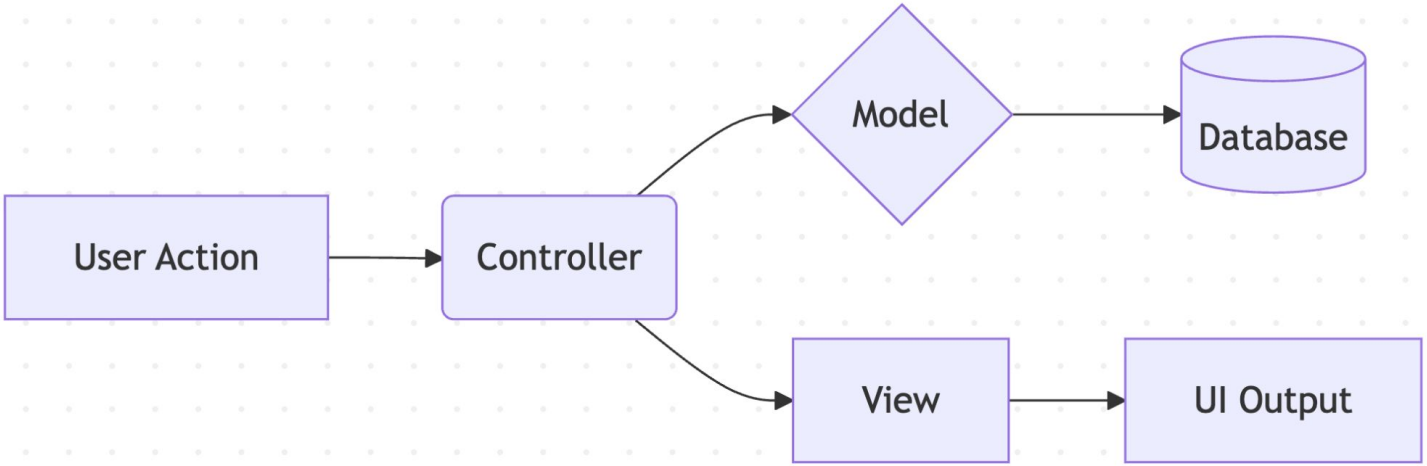
- Controller **handles user input** and interactions.
- It acts as an **intermediary** between the View and the Model.
- Controller interprets user actions (clicks, form submissions, API requests etc.).
- It **updates the Model** or **selects the appropriate View** to render.
- Controller contains application flow and coordination logic, but not data storage.
- Examples: Request handlers, controllers in web frameworks, event handlers, input processors etc.

The Controller decides what should happen, but does not store data or display it.

MVC Data Flow

- **User Interaction:** The user performs an action in the View (clicks a button).
- **Request Routing:** The View forwards the action to the Controller.
- **Logic Processing:** The Controller interprets the request. It may:
    - Update the Model (e.g. "add this new item")
    - Request data from the Model
- **State Change:** The Model updates its state, and the Controller re-renders the View using the updated data.
- **UI Update:** The View refreshes itself, displaying the updated data from the Model. User sees the change.

MVC Data Flow



MVC Folder Structure

Without MVC

```
project/
├── login.php
├── dashboard.php
├── db.php
├── user.php
├── styles.css
└── script.js
```

With MVC

```
project/
├── models/
│   └── User.php
│
├── views/
│   ├── login.php
│   └── dashboard.php
│
├── controllers/
│   └── UserController.php
│
├── config/
│   └── database.php
│
└── public/
    ├── index.php      ← ENTRY POINT (Front Controller)
    ├── css/
    └── js/
```

Architecting Project using MVC

models/User.php

```php
<?php

class User {

    private $name;

    public function __construct($name) {
        $this->name = $name;
    }

    public function getName() {
        return $this->name;
    }
}
```

controllers/UserController.php

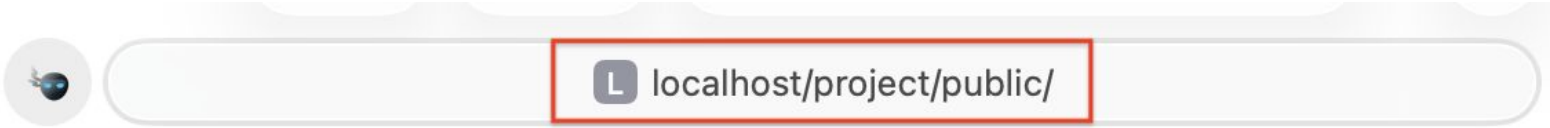
```php
require_once "../models/User.php";

class UserController {
    public function showDashboard() {
        $user = new User("Ram");
        $name = $user->getName();
        require "../views/dashboard.php";
    }
}
```

views/dashboard.php

```
<h1>Welcome, <?= $name ?></h1>
```

public/index.php

```
<?php

require_once
"../controllers/UserController.php";

(new UserController())->showDashboard();
```



Welcome, Ram

Connecting MVC Components

- **The Problem:** How do our Model, View, and Controller share resources (especially like the database connections)?
- **Bad approach:** Each component creates its own resource.
- **Better approach:** Resources are created once and shared with components that need them.
- The approach makes the code more flexible and testing easier.
- It also hides the implementation details where components only need to know what other components can do, not how they do it.

Connecting MVC Components

```php
// Bad: Hard-coded inside the class
class StudentModel {
    public function __construct() {
        $this->db = new PDO("mysql:host=localhost...");
    }
}

// Good: Receive connection from outside
class StudentModel {
    public function __construct($db) {
        $this->db = $db;
    }
}
```

Dependency Injection

- Dependency injection is the approach of programming where objects are given the resources they need from outside (from other components) rather than having them create those resources themselves.
- In our example, we passed the database connection to the **StudentModel** from outside (through the constructor) rather than having the model create it by itself.
- This simple technique keeps each components focused on their single job and makes the entire application easier to maintain and test.

References

- Twig
- Blade
- Smarty
- PHP


THANK
YOU