

Дисципліна

«Бекенд-розробка, сервіси та мікросервіси»

КОНСПЕКТ ЛЕКЦІЇ. ТЕМА 9

Патерн MVC та архітектура серверного застосунку "з нуля" (без фреймворку)

Спеціальність: Комп'ютерна інженерія, 3 курс

Обсяг: 1 год 20 хв (лекція 9 з 18)

План лекції

1. Навіщо потрібна архітектура застосунку
2. Патерн MVC: розподіл відповідальності
3. Front Controller та маршрутизація запитів
4. Модель (Model): робота з даними
5. Представлення (View): шаблони
6. Контролер (Controller): об'єднання логіки
7. Структура проєкту та автозавантаження класів

Мета лекції: навчитися організовувати код серверного застосунку за архітектурним патерном MVC, побудувати власний Front Controller і простий маршрутизатор, а також зрозуміти, звідки походять принципи, на яких побудовані сучасні PHP-фреймворки.

1. Навіщо потрібна архітектура застосунку

Усі попередні теми курсу — робота з формами, базою даних, сесіями — розглядалися в окремих, ізольованих PHP-файлах. Якщо просто писати весь код (SQL-запити, HTML-розмітку, перевірку прав доступу) підряд в одному файлі для кожної сторінки сайту, застосунок швидко перетворюється на «спагеті-код»: логіку неможливо повторно використати, протестувати окремо від виводу чи змінити один аспект (наприклад, дизайн сторінки), не ризикуючи зламати інший (бізнес-логіку).

```
<?php
// products.php — типовий приклад "спагеті-коду"
session_start();
if (!isset($_SESSION['user_id'])) { header('Location: /login.php');
exit; }

$pdo = new PDO('mysql:host=localhost;dbname=shop', 'user', 'pass');
$stmt = $pdo->query('SELECT * FROM products');
$products = $stmt->fetchAll();
?>
<!DOCTYPE html>
<html><body>
<h1>Товари</h1>
<?php foreach ($products as $p): ?>
    <p><?= htmlspecialchars($p['name']) ?> — <?= $p['price'] ?> грн</p>
<?php endforeach; ?>
</body></html>
```

У цьому файлі змішано щонайменше чотири різні відповідальності: перевірка автентифікації, підключення до бази даних, отримання даних і формування HTML. Архітектурні патерни, зокрема MVC, пропонують чіткі правила розподілу цих відповідальностей між окремими частинами коду.

2. Патерн MVC: розподіл відповідальності

MVC (Model-View-Controller) — архітектурний патерн, що розділяє застосунок на три взаємопов'язані, але незалежні частини, кожна з яких відповідає лише за одну сферу.

Компонент	Відповідальність	НЕ відповідає за
Model	Дані та бізнес-логіка: робота з базою даних, правила предметної області	HTML, вивід, обробку HTTP-запиту
View	Представлення даних користувачу: HTML-розмітка, форматування	SQL-запити, бізнес-логіку, перевірку прав
Controller	Приймає запит, координує Model і View, повертає відповідь	Прямий доступ до БД, деталі HTML-розмітки

Потік виконання типового запиту в MVC-застосунку: HTTP-запит потрапляє до контролера → контролер звертається до моделі за потрібними даними → контролер передає отримані дані у представлення → представлення формує HTML → готова відповідь повертається клієнту. Контролер виступає координатором, але сам не містить ні SQL-запитів, ні HTML-розмітки.

Головна перевага MVC — можливість незалежно змінювати кожен шар: дизайнер може переробити View, не торкаючись бізнес-логіки; розробник бази даних може оптимізувати Model, не змінюючи жодного рядка HTML; а сам Controller залишається тонким «диспетчером», який рідко потребує змін.

3. Front Controller та маршрутизація запитів

Front Controller (єдина точка входу) — патерн, за яким усі HTTP-запити до застосунку спрямовуються на один файл (зазвичай `index.php`), який далі сам вирішує, який саме контролер і метод повинен обробити конкретний запит. Це на відміну від підходу «один файл — одна сторінка» (`products.php`, `contact.php`, ...), яким користуються найпростіші сайти.

```
# .htaccess — перенаправляє всі запити на index.php,  
# якщо реального файлу чи теки з таким шляхом не існує  
RewriteEngine On  
RewriteCond %{REQUEST_FILENAME} !-f  
RewriteCond %{REQUEST_FILENAME} !-d  
RewriteRule ^ index.php [QSA,L]
```

Маршрутизатор (Router) — компонент, який аналізує URL запиту (та іноді HTTP-метод) і визначає, який контролер і метод повинні його обробити. Найпростіша реалізація — масив відповідностей «шлях → контролер».

```
<?php  
// index.php — Front Controller  
require __DIR__ . '/vendor/autoload.php';  
  
$uri = parse_url($_SERVER['REQUEST_URI'], PHP_URL_PATH);  
  
$routes = [  
    '/' => [App\Controllers\HomeController::class, 'index'],  
    '/products' => [App\Controllers\ProductController::class, 'list'],  
    '/products/create' => [App\Controllers\ProductController::class,  
    'create'],  
];  
  
if (!isset($routes[$uri])) {  
    http_response_code(404);  
    exit('Сторінку не знайдено');  
}  
  
[$controllerClass, $method] = $routes[$uri];  
$controller = new $controllerClass();  
$controller->$method();
```

Реальні фреймворки (Laravel, Symfony) мають набагато потужніші маршрутизатори — з підтримкою динамічних параметрів у шляху (`/products/{id}`), груп маршрутів, проміжного ПЗ (middleware) тощо. Наведений приклад — навмисно спрощена версія, що демонструє сам принцип, а не production-ready рішення.

4. Модель (Model): робота з даними

Модель інкапсулює доступ до даних і бізнес-правила предметної області. У попередній темі курсу ми вже створювали клас `ProductRepository`, який якраз і є прикладом моделі — контролер звертається до нього за даними, не знаючи деталей SQL-запитів.

```
<?php
namespace App\Models;

use PDO;

class Product {
    public function __construct(
        public int $id,
        public string $name,
        public float $price,
    ) {}
}

class ProductRepository {
    public function __construct(private PDO $pdo) {}

    public function findAll(): array {
        $rows = $this->pdo->query('SELECT * FROM products')->fetchAll(PDO::FETCH_ASSOC);
        return array_map(
            fn($row) => new Product($row['id'], $row['name'], $row['price']),
            $rows
        );
    }
}
```

Клас `Product` тут — проста модель даних (Data Model), яка описує структуру одного товару; `ProductRepository` — модель доступу до даних, яка вміє отримувати, створювати й оновлювати ці об'єкти в базі даних. Розділення на ці два типи класів (сама сутність і клас для роботи з нею) — поширена практика в PHP-застосунках.

5. Представлення (View): шаблони

View відповідає виключно за формування HTML на основі даних, переданих контролером. Найпростіший спосіб реалізувати шаблони в PHP без стороннього движка — окремі .php файли, що містять здебільшого HTML з невеликими вставками PHP-коду для виводу змінних і найпростіших циклів/умов.

```
<?php // views/products/list.php ?>
<!DOCTYPE html>
<html lang="uk">
<head><meta charset="UTF-8"><title>Товари</title></head>
<body>
    <h1>Каталог товарів</h1>
    <ul>
        <?php foreach ($products as $product): ?>
            <li>
                <?= htmlspecialchars($product->name) ?> –
                <?= number_format($product->price, 2) ?> грн
            </li>
        <?php endforeach; ?>
    </ul>
</body>
</html>
```

Щоб контролер міг передати дані у шаблон і отримати готовий HTML як рядок (наприклад, для подальшої обробки чи вставки в загальний layout), зручно скористатися функцією `extract()` разом із вихідним буфером із теми 5.

```
<?php
function renderView(string $viewPath, array $data = []): string {
    extract($data); // перетворює ['products' => $arr] на змінну $products
    ob_start();
    require $viewPath;
    return ob_get_clean();
}

// Використання в контролері
$html = renderView(__DIR__ . '/views/products/list.php', ['products' => $products]);
echo $html;
```

У реальних проєктах замість такого «саморобного» рендерера часто використовують шаблонізатори (наприклад, Twig, Blade у Laravel), які додають зручний власний синтаксис, автоматичне екранування виводу за замовчуванням та успадкування шаблонів (layouts). Принцип, однак, залишається тим самим: View отримує дані й повертає готовий HTML, не звертаючись до бази даних.

6. Контролер (Controller): об'єднання логіки

Контролер — тонкий координаційний шар: приймає вхідні дані запиту, викликає потрібні методи моделі, передає отриманий результат у представлення. Хороший контролер зазвичай короткий — уся справжня логіка винесена в модель, а вся розмітка — у представлення.

```
<?php
namespace App\Controllers;

use App\Models\ProductRepository;

class ProductController {
    private ProductRepository $repository;

    public function __construct() {
        $pdo = new PDO('mysql:host=localhost;dbname=shop;charset=utf8mb4',
            'user', 'pass');
        $this->repository = new ProductRepository($pdo);
    }

    public function list(): void {
        $products = $this->repository->findAll();
        echo renderView(__DIR__ . '/../views/products/list.php', [
            'products' => $products,
        ]);
    }

    public function create(): void {
        if ($_SERVER['REQUEST_METHOD'] === 'POST') {
            $this->repository->create($_POST['name'], (float)
                $_POST['price']);
            header('Location: /products');
            exit;
        }
        echo renderView(__DIR__ . '/../views/products/create.php');
    }
}
```

Метод create() демонструє типовий патерн для форм: якщо запит GET — показати порожню форму; якщо POST — обробити дані форми, зберегти й перенаправити (Post/Redirect/Get). Такий редирект після успішного POST запобігає повторній відправці форми при оновленні сторінки браузером.

7. Структура проєкту та автозавантаження класів

Коли класів стає багато (моделі, контролери, допоміжні сервіси), незручно вручну підключати кожен файл через `require`. Автозавантаження (autoloading) дозволяє PHP самому знаходити й підключати потрібний файл класу в момент першого звернення до нього.

7.1. Типова структура проєкту

```
project/
├── public/
│   ├── index.php          (Front Controller – єдина точка входу)
│   └── .htaccess
├── src/
│   ├── Controllers/
│   │   ├── HomeController.php
│   │   └── ProductController.php
│   ├── Models/
│   │   ├── Product.php
│   │   └── ProductRepository.php
│   └── views/
│       └── products/
│           ├── list.php
│           └── create.php
└── vendor/                (залежності, встановлені через Composer)
```

7.2. Автозавантаження через `spl_autoload_register()`

Функція `spl_autoload_register()` реєструє власну функцію, яку PHP викликає щоразу, коли зустрічає посилання на клас, який ще не підключено — усередині неї можна перетворити ім'я простору імен класу на шлях до файлу.

```
<?php
spl_autoload_register(function (string $class) {
    // App\Models\Product -> src/Models/Product.php
    $prefix = 'App\\';
    $baseDir = __DIR__ . '/src/';

    if (!str_starts_with($class, $prefix)) {
        return;
    }

    $relativeClass = substr($class, strlen($prefix));
    $file = $baseDir . str_replace('\\', '/', $relativeClass) . '.php';

    if (file_exists($file)) {
        require $file;
    }
});
```

Наведена логіка — спрощена реалізація стандарту автозавантаження PSR-4, прийнятого в спільноті PHP: простір імен класу відповідає структурі тек проекту. На практиці замість написання власного автозавантажувача використовують Composer — менеджер пакетів PHP, який генерує оптимізований автозавантажувач автоматично на основі конфігурації в файлі `composer.json`.

Усе, що ми реалізували вручну в цій лекції — Front Controller, простий Router, функція `renderView()`, автозавантаження класів — це, по суті, у мініатюрі ті самі механізми, на яких побудовані Laravel, Symfony та інші сучасні PHP-фреймворки. Розуміння цих основ робить набагато простішим подальше вивчення будь-якого фреймворка.

Питання для самоперевірки

- Які саме відповідальності зазвичай змішуються в "спагеті-кодi" одного PHP-файлу?
- За що відповідає кожен з трьох компонентів MVC і чого кожен з них НЕ повинен робити?
- Що таке Front Controller і чим цей підхід відрізняється від структури "один файл — одна сторінка"?
- Опишіть, як найпростіший маршрутизатор визначає, який контролер обробить конкретний запит.
- Чим модель даних (наприклад, клас Product) відрізняється від моделі доступу до даних (ProductRepository)?
- Для чого функція `renderView()` використовує `ob_start()` і `extract()`?
- Що таке патерн Post/Redirect/Get і яку проблему він вирішує?
- Як `spl_autoload_register()` дозволяє уникнути ручного `require` для кожного класу?

Короткий глосарій

Термін	Означення
MVC	Архітектурний патерн, що розділяє застосунок на Model, View, Controller
Front Controller	Єдина точка входу, через яку проходять усі запити застосунку
Маршрутизатор (Router)	Компонент, що визначає, який контролер обробить запит за URL
Модель (Model)	Компонент, відповідальний за дані та бізнес-логіку
Представлення (View)	Компонент, відповідальний за формування HTML з переданих даних
Контролер (Controller)	Координаційний шар між запитом, моделлю та представленням
Автозавантаження (autoloading)	Автоматичне підключення файлу класу під час першого звернення до нього
PSR-4	Стандарт відповідності простору імен класу структурі тек проекту