

Дисципліна

«Бекенд-розробка, сервіси та мікросервіси»

КОНСПЕКТ ЛЕКЦІЇ. ТЕМА 11

Тестування бекенду: unit/functional тести, Postman/Insomnia, документування API (OpenAPI/Swagger)

Спеціальність: Комп'ютерна інженерія, 3 курс
Обсяг: 1 год 20 хв (лекція 11 з 18)

План лекції

1. Навіщо тестувати бекенд: піраміда тестування
2. Unit-тестування з PHPUnit
3. Функціональне тестування API
4. Ручне тестування API: Postman та Insomnia
5. Документування API: навіщо потрібна специфікація
6. OpenAPI (Swagger): структура специфікації
7. Практика: опис ендпоінту через OpenAPI

Мета лекції: навчитися перевіряти коректність роботи бекенду на різних рівнях (від окремої функції до цілого API-запиту), опанувати інструменти ручного тестування API та навчитися документувати API у форматі, зрозумілому як людям, так і машинам.

1. Навіщо тестувати бекенд: піраміда тестування

Ручна перевірка коду («запустив, подивився, що не впало») працює, доки застосунок малий. У міру зростання проєкту одна й та сама логіка (наприклад, обчислення знижки чи перевірка прав доступу) може використовуватись у десятках місць, а ручна перевірка кожної зміни стає неможливою. Автоматизовані тести — код, який перевіряє інший код, — вирішують цю проблему: раз написаний тест можна запускати необмежену кількість разів за секунди.

Класична «піраміда тестування» описує оптимальне співвідношення різних видів тестів у проєкті: багато швидких unit-тестів в основі, менше повільніших функціональних (інтеграційних) тестів посередині, і зовсім небагато End-to-End тестів на вершині.

Рівень	Що перевіряє	Швидкість
Unit-тести	Одну функцію чи метод ізольовано від решти системи	Дуже швидкі (мілісекунди)
Функціональні (інтеграційні) тести	Взаємодію кількох компонентів (напр. контролер + БД)	Повільніші (секунди)
End-to-End тести	Весь застосунок очима користувача (через браузер)	Найповільніші (хвилини)

Правило "чим нижче рівень піраміди, тим більше тестів" існує не випадково: unit-тести дешево писати й підтримувати, вони швидко вказують на точне місце помилки. End-to-End тести цінні, але крихкі (ламаються від найменшої зміни інтерфейсу) і повільні — тому їх пишуть для небагатьох найкритичніших сценаріїв.

2. Unit-тестування з PHPUnit

PHPUnit — стандартний фреймворк для unit-тестування в PHP-екосистемі, який використовують практично всі сучасні проєкти й фреймворки (Laravel, Symfony). Unit-тест перевіряє одну функцію чи метод ізольовано, часто замінюючи (мокуючи) залежності, щоб тест не залежав від реальної бази даних чи мережі.

2.1. Структура тесту

Типовий unit-тест складається з трьох частин, відомих як шаблон AAA: Arrange (підготувати вхідні дані), Act (викликати тестований код), Assert (перевірити результат).

```
<?php
// tests/PriceCalculatorTest.php
use PHPUnit\Framework\TestCase;

class PriceCalculatorTest extends TestCase {
    public function testAddsTaxCorrectly(): void {
        // Arrange
        $calculator = new PriceCalculator();

        // Act
        $result = $calculator->addTax(100, 0.2);

        // Assert
        $this->assertEquals(120, $result);
    }

    public function testThrowsExceptionForNegativePrice(): void {
        $calculator = new PriceCalculator();

        $this->expectException(InvalidArgumentException::class);
        $calculator->addTax(-100, 0.2);
    }
}
```

2.2. Основні методи перевірки (assertions)

Метод	Перевіряє
<code>assertEquals(\$a, \$b)</code>	Значення рівні (з нежорстким порівнянням типів)
<code>assertSame(\$a, \$b)</code>	Значення й тип ідентичні (<code>===</code>)
<code>assertTrue()</code> / <code>assertFalse()</code>	Значення є <code>true</code> / <code>false</code>
<code>assertNull()</code>	Значення дорівнює <code>null</code>
<code>assertCount(\$n, \$array)</code>	Масив містить рівно <code>\$n</code> елементів
<code>expectException(\$class)</code>	Викликаний код кидає вказаний виняток

2.3. Мокування залежностей

Коли тестований клас залежить від інших об'єктів (наприклад, від бази даних через PDO), у unit-тесті цю залежність підміняють тестовим дублером (mock) — об'єктом, що імітує потрібну поведінку без реального звернення до БД. Це дозволяє тестувати логіку класу-репозиторію, не піднімаючи справжню базу даних.

```
<?php
public function testFindByIdReturnsNullWhenNotFound(): void {
    $pdoMock = $this->createMock(PDO::class);
    $stmtMock = $this->createMock(PDOStatement::class);

    $stmtMock->method('fetch')->willReturn(false);
    $pdoMock->method('prepare')->willReturn($stmtMock);

    $repository = new ProductRepository($pdoMock);
    $result = $repository->findById(999);

    $this->assertNull($result);
}
```

Мокування — потужний, але не завжди потрібний інструмент: для простих класів без зовнішніх залежностей (наприклад, PriceCalculator із першого прикладу) тест можна писати без жодних моків. Мокувати варто саме межі системи — базу даних, мережеві запити, файлову систему.

3. Функціональне тестування API

Функціональний (інтеграційний) тест перевіряє не окрему функцію, а взаємодію кількох реальних компонентів разом — наприклад, повний цикл обробки HTTP-запиту: маршрутизатор → контролер → база даних → відповідь. Для REST API це означає надіслати справжній HTTP-запит до застосунку й перевірити код статусу та вміст відповіді.

```
<?php
// tests/Feature/ProductApiTest.php
use PHPUnit\Framework\TestCase;
use GuzzleHttp\Client;

class ProductApiTest extends TestCase {
    private Client $client;

    protected function setUp(): void {
        $this->client = new Client(['base_uri' =>
'http://localhost:8000']);
    }

    public function testGetProductReturns200AndCorrectData(): void {
        $response = $this->client->get('/api/products/1');

        $this->assertEquals(200, $response->getStatusCode());

        $data = json_decode($response->getBody(), true);
        $this->assertArrayHasKey('name', $data);
    }

    public function testGetNonExistentProductReturns404(): void {
        $response = $this->client->get('/api/products/99999', [
            'http_errors' => false,
        ]);

        $this->assertEquals(404, $response->getStatusCode());
    }
}
```

Такі тести зазвичай запускають проти окремої тестової бази даних (не проти бойової/production), яку перед кожним запуском тестів очищують і заповнюють наперед відомими тестовими даними (fixtures) — це гарантує відтворюваність результату незалежно від того, скільки разів тести запускались раніше.

Функціональні тести повільніші за unit-тести (реальний HTTP-запит, реальна база даних), тому їх пишуть здебільшого для найважливіших сценаріїв ("щасливий шлях" і кілька критичних помилок), залишаючи детальну перевірку окремих правил бізнес-логіки unit-тестам.

4. Ручне тестування API: Postman та Insomnia

Перш ніж писати автоматизовані тести, розробники здебільшого перевіряють API вручну під час розробки. Postman та Insomnia — найпопулярніші графічні інструменти для ручного тестування API: вони дозволяють сформулювати HTTP-запит із потрібним методом, заголовками й тілом, надіслати його та наочно переглянути відповідь, без потреби писати код чи використовувати командний рядок.

4.1. Типовий робочий процес у Postman

1. Обрати HTTP-метод (GET, POST, PUT, DELETE) з випадного списку.
1. Ввести URL ендпоінта (наприклад, `http://localhost:8000/api/products`).
1. За потреби додати заголовки (Headers), наприклад `Authorization: Bearer <token>`.
1. Для POST/PUT — заповнити тіло запиту (Body) у форматі JSON.
1. Надіслати запит (Send) і переглянути код статусу, заголовки й тіло відповіді.

4.2. Колекції запитів

І Postman, і Insomnia дозволяють групувати пов'язані запити в колекції (наприклад, усі ендпоінти для роботи з товарами) і зберігати змінні середовища (base URL, токен автентифікації), щоб не вводити ці дані вручну щоразу. Колекцію можна експортувати у файл і поділитися нею з іншими членами команди — фактично, це стає живою документацією API, яку можна одразу виконати.

Інструмент	Особливості
Postman	Найпопулярніший, хмарна синхронізація колекцій, вбудоване тестове середовище (написання перевірок на JavaScript)
Insomnia	Легший і швидший інтерфейс, зручна робота з GraphQL, безкоштовний та відкритий вихідний код (частково)

Postman дозволяє писати прості тестові скрипти прямо у вкладці Tests (JavaScript-код, що виконується після отримання відповіді, наприклад `pm.test("Status is 200", () => pm.response.to.have.status(200))`), а потім запускати всю колекцію автоматично (Collection Runner) — це проміжний крок між суто ручним тестуванням і повноцінними автоматизованими тестами на PHPUnit.

5. Документування API: навіщо потрібна специфікація

Коли API використовують інші розробники (команда фронтенду, мобільні розробники, зовнішні партнери), їм потрібно знати, які ендпоінти існують, які параметри вони приймають, який формат відповіді повертають і які помилки можливі — без цього доводиться вивчати серверний код напряду або постійно перепитувати автора API.

Написана вручну текстова документація (наприклад, у Word чи вікі) швидко застаріває: розробники змінюють код, але забувають оновити опис. Специфікація API у машинозчитуваному форматі (як-от OpenAPI) вирішує цю проблему частково: з неї можна автоматично згенерувати наочну документацію, тестові запити й навіть заглушки клієнтського коду, тому підтримувати її актуальною економічно вигідніше.

OpenAPI (раніше відомий як Swagger Specification) — це формат опису API, тоді як Swagger — набір інструментів (Swagger UI, Swagger Editor тощо), які працюють із файлами специфікації OpenAPI. У розмові ці терміни часто вживають як синоніми, хоча формально OpenAPI — це стандарт, а Swagger — конкретна реалізація інструментів навколо нього.

6. OpenAPI (Swagger): структура специфікації

Специфікація OpenAPI — це YAML- або JSON-файл, що формально описує всі ендпоінти API: шляхи, методи, параметри, тіла запитів і відповідей, коди статусів. На основі цього файлу інструмент Swagger UI автоматично генерує інтерактивну сторінку документації, де кожен ендпоінт можна одразу випробувати прямо в браузері.

```
openapi: 3.0.0
info:
  title: Products API
  version: 1.0.0

paths:
  /products/{id}:
    get:
      summary: Отримати товар за id
      parameters:
        - name: id
          in: path
          required: true
          schema:
            type: integer
      responses:
        '200':
          description: Товар знайдено
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Product'
        '404':
          description: Товар не знайдено
```

6.1. Опис моделі даних (schemas)

Розділ components/schemas дозволяє один раз описати структуру об'єкта (наприклад, Product) і надалі посилатися на неї через \$ref у будь-якому ендпоінті, що приймає чи повертає такий об'єкт — це уникає дублювання опису в кожному ендпоінті.

```
components:
  schemas:
    Product:
      type: object
      properties:
        id:
          type: integer
          example: 42
        name:
          type: string
          example: Ноутбук
        price:
          type: number
          example: 25000
```

Розділ специфікації	Призначення
info	Назва, версія та опис API загалом
paths	Перелік ендпоінтів, методів, параметрів і відповідей
components/schemas	Багаторазово використовувані описи структур даних
components/securitySchemes	Опис способу автентифікації (напр. Bearer JWT)

7. Практика: опис ендпоінту через OpenAPI

Розглянемо повний приклад опису двох ендпоінтів (GET і POST для товарів), який поєднує все розглянуте в цій темі — параметри, тіло запиту, схему даних і можливі коди відповідей.

```
paths:
  /products:
    get:
      summary: Отримати список товарів
      responses:
        '200':
          description: Список товарів
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/Product'
    post:
      summary: Створити новий товар
      requestBody:
        required: true
        content:
          application/json:
            schema:
              type: object
              required: [name, price]
              properties:
                name:
                  type: string
                price:
                  type: number
      responses:
        '201':
          description: Товар створено
        '422':
          description: Помилка валідації
```

Такий файл `specification.yaml` зазвичай зберігають у самому репозиторії проєкту поряд з кодом і підключають до Swagger UI (окремий вебзастосунок, часто наданий у вигляді Docker-контейнера чи бібліотеки), який автоматично відображає його як інтерактивну документацію.

У великих PHP-проєктах специфікацію OpenAPI часто генерують не вручну, а з коментарів-анотацій прямо в коді контролерів (наприклад, за допомогою бібліотеки `zircote/swagger-php`) — тоді документація й код фізично лежать поряд і рідше розходяться між собою, оскільки одна зміна в контролері одразу оновлює й специфікацію при наступній генерації.

Питання для самоперевірки

- Чому в піраміді тестування unit-тестів має бути найбільше, а End-to-End — найменше?
- З яких трьох частин складається типовий unit-тест за шаблоном AAA?
- Для чого потрібне мокування залежностей і які саме залежності зазвичай мокають?
- Чим функціональний (інтеграційний) тест API відрізняється від unit-тесту?
- Навіщо для функціональних тестів використовують окрему тестову базу даних?
- Порівняйте Postman і ручне тестування через звичайний браузер — які переваги дає Postman?
- Чим OpenAPI відрізняється від Swagger як термін?
- Для чого в специфікації OpenAPI використовують розділ components/schemas замість опису структури в кожному ендпоінті окремо?

Короткий глосарій

Термін	Означення
Unit-тест	Тест, що перевіряє одну функцію чи метод ізольовано від решти системи
Функціональний (інтеграційний) тест	Тест, що перевіряє взаємодію кількох реальних компонентів разом
Мок (mock)	Тестовий дублер об'єкта, що імітує поведінку реальної залежності
Fixture	Наперед підготовлені тестові дані для відтворюваного запуску тестів
Postman / Insomnia	Графічні інструменти для ручного тестування HTTP-запитів
OpenAPI	Формат специфікації для формального опису REST API
Swagger UI	Інструмент, що генерує інтерактивну документацію з файлу специфікації OpenAPI