

Дисципліна

«Бекенд-розробка, сервіси та мікросервіси»

КОНСПЕКТ ЛЕКЦІЇ. ТЕМА 10

REST API: принципи, проєктування ендпоінтів, коди відповідей, версіонування

Спеціальність: Комп'ютерна інженерія, 3 курс

Обсяг: 1 год 20 хв (лекція 10 з 18)

План лекції

1. Що таке REST API
2. Принципи REST архітектури
3. Проектування ендпоінтів: ресурси та URI
4. HTTP-методи та відповідність CRUD-операціям
5. Коди відповідей HTTP у REST API
6. Версіонування API
7. Практична реалізація REST API на PHP

Мета лекції: навчитися проектувати й реалізовувати REST API — уніфікований спосіб організації взаємодії між клієнтом і сервером, на якому побудована переважна більшість сучасних вебсервісів, мобільних застосунків і мікросервісів.

1. Що таке REST API

API (Application Programming Interface) — інтерфейс, через який одна програма може звертатися до функціональності іншої. ВебAPI дозволяє клієнту (браузеру, мобільному застосунку, іншому серверу) звертатися до сервера через HTTP-запити й отримувати дані, найчастіше у форматі JSON, замість готової HTML-сторінки.

REST (Representational State Transfer) — архітектурний стиль побудови вебAPI, запропонований Роем Філдіном 2000 року. REST не є протоколом чи стандартом у строгому значенні — це набір принципів проектування, дотримання яких робить API передбачуваним, масштабованим і зручним для незалежної розробки клієнта й сервера.

```
GET /api/products/42 HTTP/1.1
Host: shop.example.com
Accept: application/json

HTTP/1.1 200 OK
Content-Type: application/json

{"id": 42, "name": "Ноутбук", "price": 25000}
```

API, що використовує JSON поверх HTTP, часто називають "REST API" навіть тоді, коли воно дотримується не всіх формальних принципів REST — на практиці цей термін здебільшого означає просто "вебAPI на основі ресурсів, HTTP-методів і JSON". Це варто розуміти як усталену галузеву термінологію, а не завжди строге академічне визначення.

2. Принципи REST архітектури

REST спирається на кілька архітектурних обмежень (constraints), дотримання яких і робить API "REST-подібним".

Принцип	Суть
Клієнт-серверна архітектура	Чіткий розподіл: клієнт відповідає за інтерфейс, сервер — за дані й логіку
Відсутність стану (statelessness)	Кожен запит містить усю потрібну інформацію; сервер не зберігає стан клієнта між запитами
Кешованість (cacheability)	Відповіді явно позначаються як придатні чи непридатні для кешування
Уніфікований інтерфейс	Однакові принципи звернення до будь-якого ресурсу (URI, HTTP-методи, формати)
Шарувата система	Клієнт не повинен знати, чи звертається він напряму до сервера, чи через проміжні шари (proxy, load balancer)

Найважливіший із цих принципів для практичної розробки — відсутність стану (statelessness): сервер не повинен "пам'ятати" попередні запити клієнта між викликами API. Якщо для виконання запиту потрібна автентифікація, токен передається в **КОЖНОМУ** запиті (наприклад, у заголовку Authorization), а не покладається на сесію, збережену після попереднього запиту.

Саме принцип відсутності стану пояснює, чому токен-базована автентифікація (JWT, розглянута в темі 8) настільки природно поєднується з REST API: кожен запит самодостатній і несе токен, необхідний для перевірки прав доступу, без залежності від сесії на конкретному сервері.

3. Проєктування ендпоінтів: ресурси та URI

У REST API URI ідентифікує ресурс (сутність предметної області — товар, користувача, замовлення), а не дію над ним. Дія визначається HTTP-методом, а не дієсловом у самому шляху — це ключова відмінність REST-стилю URI від "процедурного" стилю на кшталт `getProducts.php` чи `/deleteUser?id=5`.

Правильно (REST)	Неправильно (процедурний стиль)
GET /products	GET /getAllProducts.php
POST /products	GET /createProduct.php?name=...
GET /products/42	GET /getProduct.php?id=42
DELETE /products/42	GET /deleteProduct.php?id=42

3.1. Правила іменування ендпоінтів

- Використовувати іменники в множині для колекцій ресурсів: `/products`, `/users`, `/orders`.
- Вкладеність шляху відображає зв'язки між ресурсами: `/users/5/orders` — усі замовлення користувача з `id=5`.
- Не використовувати дієслова в шляху (`get`, `create`, `delete`) — ці дії вже виражені HTTP-методом.
- Використовувати дефіс (`-`), а не підкреслення (`_`) для розділення слів у шляху: `/product-categories`.
- Уникати надмірної вкладеності (більше 2–3 рівнів) — це ускладнює читання й використання API.

Ендпоінт	Призначення
GET /products	Отримати список усіх товарів
GET /products/42	Отримати конкретний товар за <code>id</code>
GET /products?category=electronics	Фільтрація списку через <code>query</code> -параметри
GET /users/5/orders	Усі замовлення конкретного користувача

4. HTTP-методи та відповідність CRUD-операціям

Кожен HTTP-метод (розглянутий детально в темі 1) у REST API відповідає одній із чотирьох базових CRUD-операцій, застосованій до одного і того самого URI ресурсу.

Метод	CRUD	Приклад	Ідемпотентний?
GET	Read	GET /products/42	Так
POST	Create	POST /products	Ні
PUT	Update (повна заміна)	PUT /products/42	Так
PATCH	Update (часткова зміна)	PATCH /products/42	Ні
DELETE	Delete	DELETE /products/42	Так

Різниця між PUT і PATCH часто викликає плутанину: PUT передбачає, що клієнт надсилає ПОВНИЙ новий стан ресурсу (усі поля), і сервер повністю замінює ним старий; PATCH передбачає часткове оновлення — надсилаються лише ті поля, які потрібно змінити.

```
// PUT — повна заміна: усі поля товару обов'язкові
PUT /products/42
{"name": "Ноутбук", "price": 27000, "category": "electronics"}

// PATCH — часткове оновлення: лише ціна
PATCH /products/42
{"price": 27000}
```

5. Коди відповідей HTTP у REST API

Коректний вибір коду статусу HTTP (класи яких розглядалися в темі 1) — важлива частина проєктування REST API: клієнт орієнтується саме на код статусу, щоб зрозуміти результат запиту, а не лише парсити тіло відповіді.

Код	Значення	Типова ситуація в REST API
200 OK	Успіх	Успішний GET, PUT, PATCH
201 Created	Ресурс створено	Успішний POST, що створив новий ресурс
204 No Content	Успіх без тіла відповіді	Успішний DELETE
400 Bad Request	Некоректний запит	Невалідний JSON, відсутнє обов'язкове поле
401 Unauthorized	Не автентифікований	Відсутній або недійсний токен
403 Forbidden	Немає прав доступу	Автентифікований, але недостатньо прав
404 Not Found	Ресурс не знайдено	GET /products/9999, якщо такого товару немає
409 Conflict	Конфлікт стану	Спроба створити ресурс, що вже існує (напр. email зайнятий)
422 Unprocessable Entity	Помилка валідації	JSON синтаксично коректний, але дані не проходять перевірку
500 Internal Server Error	Помилка сервера	Неочікуваний виняток у коді сервера

Поширена помилка початківців — повертати 200 OK для всіх відповідей, включно з помилками, і покладатися лише на поле на кшталт {"success": false} у тілі JSON. Це ускладнює роботу клієнта (доводиться завжди парсити тіло, щоб дізнатися результат) і ламає стандартну поведінку HTTP-клієнтів, проксі й кешів, які орієнтуються саме на код статусу.

6. Версіонування API

З часом API розвивається: додаються нові поля, змінюється структура відповіді, видаляється застарілий функціонал. Якщо просто змінити поведінку існуючого ендпоінта, усі клієнти, які вже його використовують (мобільні застосунки, сторонні інтеграції), можуть одразу зламатися. Версіонування дозволяє розвивати API, не порушуючи роботу вже наявних клієнтів.

6.1. Версіонування через URI

Найпоширеніший і найпростіший підхід — включити номер версії безпосередньо в шлях URI.

```
GET /api/v1/products/42
GET /api/v2/products/42
```

6.2. Версіонування через заголовок

Альтернативний підхід — вказувати версію в заголовку запиту (наприклад, Accept або власному заголовку), залишаючи URI незмінним. Це вважається "чистішим" з погляду REST (URI ідентифікує саме ресурс, а не його представлення), але менш очевидне й важче тестувати вручну (наприклад, у браузері).

```
GET /api/products/42
Accept: application/vnd.shop.v2+json
```

Підхід	Переваги	Недоліки
У шляху (/v1/, /v2/)	Простий, очевидний, легко тестувати	"Забруднює" URI версією представлення
У заголовку	URI лишається чистим	Менш очевидний, складніше дебажити

Незалежно від обраного підходу, головне правило версіонування — ніколи не вносити зворотньо несумісні (breaking) зміни в наявну версію API. Якщо потрібно змінити структуру відповіді чи видалити поле, це робиться у новій версії, а стара версія продовжує працювати для існуючих клієнтів протягом певного перехідного періоду.

7. Практична реалізація REST API на PHP

Спираючись на MVC-архітектуру з попередньої теми, REST API реалізується так само через контролери — лише замість формування HTML вони формують JSON-відповідь із коректним кодом статусу.

```
<?php
namespace App\Controllers;

use App\Models\ProductRepository;

class ProductApiController {
    public function __construct(private ProductRepository $repository) {}

    public function show(int $id): void {
        $product = $this->repository->findById($id);

        header('Content-Type: application/json');

        if ($product === null) {
            http_response_code(404);
            echo json_encode(['error' => 'Товар не знайдено']);
            return;
        }

        http_response_code(200);
        echo json_encode($product);
    }

    public function store(): void {
        header('Content-Type: application/json');
        $data = json_decode(file_get_contents('php://input'), true);

        if (empty($data['name']) || empty($data['price'])) {
            http_response_code(422);
            echo json_encode(['error' => 'Поля name і price обов\'язкові']);
            return;
        }

        $id = $this->repository->create($data['name'], $data['price']);
        http_response_code(201);
        echo json_encode(['id' => $id, 'name' => $data['name'], 'price' => $data['price']]);
    }

    public function destroy(int $id): void {
        $this->repository->delete($id);
        http_response_code(204);
    }
}
```

Маршрутизатор для такого API повинен враховувати не лише URI, а й HTTP-метод, оскільки той самий шлях `/products/42` обробляється по-різному залежно від методу (`GET` → `show`, `PUT` → `update`, `DELETE` → `destroy`).

```
<?php
// Спрощена маршрутизація з урахуванням методу
$method = $_SERVER['REQUEST_METHOD'];
$uri = parse_url($_SERVER['REQUEST_URI'], PHP_URL_PATH);

if (preg_match('#^/products/(\d+)$#', $uri, $matches)) {
    $id = (int) $matches[1];
    match ($method) {
        'GET' => $controller->show($id),
        'PUT', 'PATCH' => $controller->update($id),
        'DELETE' => $controller->destroy($id),
        default => http_response_code(405), // Method Not Allowed
    };
} elseif ($uri === '/products' && $method === 'POST') {
    $controller->store();
}
```

Код 405 Method Not Allowed повертається, коли URI впізнано, але для нього не передбачено обробку саме такого HTTP-методу (наприклад, DELETE /products без вказаного id) — це відрізняється від 404 Not Found, яке означає, що самого ресурсу за таким шляхом не існує.

Питання для самоперевірки

- Чим REST API як усталений галузевий термін відрізняється від суворого академічного визначення REST Роя Філдінга?
- Поясніть принцип відсутності стану (statelessness) і чому він важливий для масштабованості API.
- Чому `/getAllProducts.php` вважається поганим прикладом REST-ендпоінта, а `GET /products` — хорошим?
- Чим PUT відрізняється від PATCH з погляду того, які дані очікує сервер?
- У яких ситуаціях доцільно повернути 422 замість 400, а коли — 409 замість 422?
- Чому повертати завжди 200 OK з полем `{"success": false}` вважається поганою практикою?
- Порівняйте версіонування API через шлях URI і через заголовок Accept — переваги й недоліки кожного.
- Чим відрізняються ситуації, коли сервер повинен повернути 404 Not Found і коли — 405 Method Not Allowed?

Короткий глосарій

Термін	Означення
API	Інтерфейс для взаємодії між програмними компонентами
REST	Архітектурний стиль побудови вебAPI на основі ресурсів та уніфікованого інтерфейсу
Ресурс	Сутність предметної області, що ідентифікується URI (товар, користувач тощо)
Statelessness	Принцип, за яким сервер не зберігає стан клієнта між запитами
Ідемпотентність	Властивість, за якої повторний виклик дає той самий результат, що й одноразовий
Версіонування API	Практика підтримки кількох версій API для сумісності зі старими клієнтами
Breaking change	Зворотньо несумісна зміна, що ламає роботу існуючих клієнтів API