

Дисципліна

«Бекенд-розробка, сервіси та мікросервіси»

КОНСПЕКТ ЛЕКЦІЇ. ТЕМА 14

API Gateway, service discovery, конфігурація мікросервісів, кешування, масштабування та балансування навантаження

Спеціальність: Комп'ютерна інженерія, 3 курс
Обсяг: 1 год 20 хв (лекція 14 з 18)

План лекції

1. API Gateway: єдина точка входу для клієнтів
2. Service Discovery: як сервіси знаходять один одного
3. Конфігурація мікросервісів
4. Кешування в розподіленій системі
5. Масштабування: горизонтальне та вертикальне
6. Балансування навантаження: алгоритми
7. Як усе це працює разом: практичний сценарій

Мета лекції: опанувати ключові інфраструктурні компоненти, без яких мікросервісна архітектура на практиці не працює — від єдиної точки входу для клієнтів до механізмів, що дозволяють системі витримувати зростання навантаження.

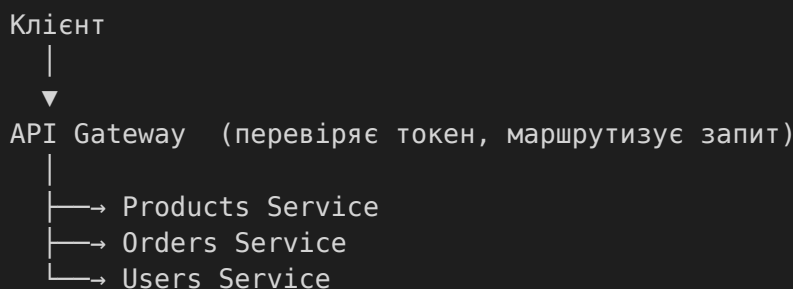
1. API Gateway: єдина точка входу для клієнтів

У мікросервісній архітектурі (тема 12) клієнт (браузер, мобільний застосунок) логічно повинен би звертатися напямку до потрібного сервіса — окремо до Products, окремо до Orders, окремо до Users. На практиці це незручно й небезпечно: клієнт мусить знати внутрішню адресу кожного сервіса, дублювати логіку автентифікації для кожного виклику, а зміна внутрішньої структури системи (об'єднання чи розбиття сервісів) ламає клієнтський код.

API Gateway — проміжний сервіс, що стає єдиною точкою входу для всіх зовнішніх запитів: клієнт звертається лише до Gateway, а той сам маршрутизує запит до потрібного внутрішнього сервіса, приховуючи від клієнта всю внутрішню структуру системи.

1.1. Основні функції API Gateway

Функція	Опис
Маршрутизація	Спрямовує запит <code>/api/products</code> до сервіса Products, <code>/api/orders</code> — до Orders
Автентифікація	Перевіряє токен (тема 8) один раз, не дублюючи логіку в кожному сервісі
Агрегація відповідей	Об'єднує дані з кількох сервісів в одну відповідь клієнту
Rate limiting	Обмежує кількість запитів від одного клієнта за одиницю часу
Логування та моніторинг	Централізовано фіксує всі запити до системи в одному місці



Популярні готові рішення для API Gateway — Kong, NGINX (з відповідною конфігурацією), AWS API Gateway, Traefik. Для навчальних і невеликих проєктів API Gateway можна реалізувати і власноруч як звичайний PHP-застосунок, що приймає запит і виконує внутрішній REST-виклик (тема 13) до потрібного сервіса.

2. Service Discovery: як сервіси знаходять один одного

У монолітній архітектурі різні частини коду знаходять одна одну просто через прямий виклик функції — адреса "вшита" в сам факт того, що код перебуває в тому самому процесі. У мікросервісній архітектурі кожен сервіс може мати кілька запущених копій (для масштабування), їхні мережеві адреси можуть змінюватися при кожному новому розгортанні, а самі сервіси — динамічно з'являтися й зникати. Service Discovery — механізм, що дозволяє сервісам знаходити актуальні адреси одне одного без жорсткого "зашивання" IP-адрес у код.

2.1. Реєстр сервісів (Service Registry)

Центральний компонент Service Discovery — реєстр сервісів: база даних, у якій кожен запущений екземпляр сервіса реєструє власну адресу під час старту (реєстрація) і періодично підтверджує, що він досі живий (heartbeat). Коли інший сервіс хоче звернутися до Products, він спочатку запитує реєстр: "де зараз доступні екземпляри Products?", і отримує актуальний перелік адрес.

1. Products Service запускається → реєструється в реєстрі:
"products-service" → 10.0.1.15:8080
2. Orders Service хоче звернутись до Products:
Запитує реєстр: "де зараз products-service?"
Отримує: 10.0.1.15:8080 (може бути кілька адрес)
3. Products Service падає чи зупиняється →
реєстр перестає отримувати heartbeat → видаляє адресу

Інструмент	Особливості
Consul	Популярний реєстр сервісів із вбудованою перевіркою здоров'я (health checks)
etcd	Розподілене сховище ключ-значення, часто використовується в Kubernetes
Kubernetes DNS	У Kubernetes service discovery вбудований — сервіси звертаються одне до одного за іменем

У сучасних хмарних платформах (наприклад, Kubernetes) service discovery здебільшого вбудований у саму інфраструктуру: достатньо звернутися до сервіса за його логічним іменем (наприклад, <http://products-service>), і платформа сама прозоро направить запит на актуальну адресу однієї з живих копій — розробнику не потрібно вручну звертатися до реєстру.

3. Конфігурація мікросервісів

Кожен сервіс потребує налаштувань: адресу бази даних, ключі API сторонніх сервісів, прапорці ввімкнення функціоналу (feature flags), ліміти й таймаути (тема 13). У моноліті такі налаштування зазвичай зберігають в одному .env файлі; коли сервісів десятки, а копій кожного — кілька, розкидані по різних серверах .env файли стає неможливо синхронізовано оновлювати.

3.1. Централізована конфігурація

Рішення — винести конфігурацію в окремий централізований сервіс (наприклад, Consul, etcd чи Spring Cloud Config), звідки кожен екземпляр сервіса зчитує свої налаштування під час старту чи навіть динамічно, без перезапуску, коли значення змінюється.

```
<?php
// Спрощений приклад: зчитування конфігурації з централізованого сховища
class ConfigClient {
    public function __construct(private string $configServiceUrl) {}

    public function get(string $key): mixed {
        $response =
file_get_contents("{$this->configServiceUrl}/config/{$key}");
        return json_decode($response, true)['value'];
    }
}

$config = new ConfigClient('http://config-service');
$dbHost = $config->get('database.host');
$featureFlagEnabled = $config->get('features.new_checkout');
```

Рівень середовища	Приклад
Development (розробка)	Локальна база даних, вимкнене логування помилок у зовнішній сервіс
Staging (тестове)	Копія бойових даних, увімкнені всі логи для діагностики
Production (бойове)	Реальна база даних, суворі ліміти, моніторинг критичних метрик

Секрети (паролі баз даних, API-ключі) не варто зберігати у звичайній конфігурації чи в системі контролю версій навіть у зашифрованому вигляді — для цього існують спеціалізовані сховища секретів (HashiCorp Vault, AWS Secrets Manager), які додатково контролюють, хто саме й коли має доступ до конкретного секрету.

4. Кешування в розподіленій системі

Кешування — збереження результату "дорогої" операції (складний SQL-запит, виклик зовнішнього API) у швидкому сховищі, щоб за повторного запиту тих самих даних не виконувати обчислення заново. У розподіленій системі кешування набуває особливого значення: воно знижує навантаження не лише на базу даних, а й на мережеві виклики між сервісами (тема 13).

4.1. Рівні кешування

Рівень	Приклад
Кешування на клієнті	Браузер кешує статичні файли (CSS, зображення) за заголовком Cache-Control
Кешування на API Gateway	Gateway кешує відповідь GET /products на кілька секунд для всіх клієнтів
Кешування всередині сервіса	Сервіс зберігає результат запиту до БД у Redis (тема 13) на певний час
Кешування бази даних	Сама СУБД кешує часто використовувані сторінки даних у пам'яті

```
<?php
function getProduct(int $id, Redis $redis, PDO $pdo): array {
    $cacheKey = "product:{$id}";

    $cached = $redis->get($cacheKey);
    if ($cached !== false) {
        return json_decode($cached, true);    // швидко: дані вже в пам'яті
    }

    $stmt = $pdo->prepare('SELECT * FROM products WHERE id = :id');
    $stmt->execute(['id' => $id]);
    $product = $stmt->fetch(PDO::FETCH_ASSOC);

    $redis->setex($cacheKey, 300, json_encode($product));    // кешувати на 5 хвилин

    return $product;
}
```

4.2. Інвалідація кешу

Найскладніша частина кешування — не саме збереження даних, а своєчасне видалення застарілого кешу (інвалідація), коли дані змінюються. Забутий кеш призводить до того, що користувачі бачать застарілу інформацію (наприклад, стару ціну товару) навіть після її оновлення в базі даних.

```
<?php
function updateProductPrice(int $id, float $newPrice, Redis $redis, PDO
$pdo): void {
    $stmt = $pdo->prepare('UPDATE products SET price = :price WHERE id
= :id');
    $stmt->execute(['price' => $newPrice, 'id' => $id]);

    $redis->del("product:{$id}"); // видалити застарілий кеш одразу
після зміни
}
```

Існує відомий жарт у програмуванні: "У комп'ютерних науках є лише дві по-справжньому складні речі: іменування змінних та інвалідація кешу" — і це не перебільшення. Занадто довгий термін дії кешу показує застарілі дані; занадто короткий — не дає реальної користі від кешування.

5. Масштабування: горизонтальне та вертикальне

Коли одного екземпляра сервіса недостатньо для обробки навантаження, існує два принципово різні підходи до збільшення потужності системи.

Підхід	Суть	Обмеження
Вертикальне масштабування	Збільшити потужність одного сервера (більше CPU, RAM)	Фізична межа заліза; єдина точка відмови
Горизонтальне масштабування	Запустити кілька копій сервіса на різних серверах	Потребує балансування навантаження й, часто, децентралізованих даних

Мікросервісна архітектура (тема 12) створена саме для зручного горизонтального масштабування: оскільки кожен сервіс — окремий незалежний процес без внутрішнього стану (stateless, тема 10), можна запустити скільки завгодно його копій, і кожна однаково здатна обробити будь-який вхідний запит.

Ключова умова для горизонтального масштабування — сервіс має бути stateless: жодних даних, специфічних для конкретного запиту чи користувача, не повинно зберігатись у пам'яті самого процесу сервіса (наприклад, сесія користувача в оперативній пам'яті одного сервера). Дані сесії, кешу тощо повинні зберігатися в спільному зовнішньому сховищі (Redis, база даних), доступному з будь-якої копії сервіса.

6. Балансування навантаження: алгоритми

Коли сервіс масштабовано горизонтально (запущено кілька його копій), потрібен компонент, що розподіляє вхідні запити між цими копіями — балансувальник навантаження (load balancer). Він отримує кожен запит і вирішує, якій саме копії сервіса його передати.

Алгоритм	Принцип
Round Robin	Запити розподіляються по черзі: 1-й екземпляр, 2-й, 3-й, знову 1-й...
Least Connections	Запит спрямовується до екземпляра з найменшою кількістю активних з'єднань
IP Hash	Клієнт із певної IP-адреси завжди потрапляє на той самий екземпляр
Weighted Round Robin	Round Robin з урахуванням "ваги" (потужності) кожного екземпляра

```
# Спрощена конфігурація NGINX як балансувальника навантаження
upstream products_service {
    server 10.0.1.15:8080;
    server 10.0.1.16:8080;
    server 10.0.1.17:8080;
}

server {
    location /api/products {
        proxy_pass http://products_service;    # NGINX сам бере екземпляр
    }
}
```

IP Hash важливий для сценаріїв, де потрібна "липкість" сесії (sticky session) — коли з якихось причин стан усе ж частково зберігається на конкретному сервері. Утім, як зазначено в розділі 5, кращою практикою вважається взагалі уникати такої залежності від конкретного екземпляра, роблячи сервіс повністю stateless.

7. Як усе це працює разом: практичний сценарій

Розглянемо, як розглянуті компоненти взаємодіють у типовому запиті користувача до інтернет-магазину, побудованого на мікросервісах.

1. Клієнт (браузер) надсилає запит GET /api/products/42 на єдину адресу API Gateway.
2. API Gateway перевіряє токен автентифікації клієнта (тема 8) і, якщо він дійсний, готовий маршрутизувати запит далі.
3. Gateway звертається до Service Discovery, щоб дізнатися поточні адреси живих екземплярів Products Service.
4. Балансувальник навантаження (часто вбудований у сам механізм service discovery чи Gateway) обирає один із доступних екземплярів за одним із розглянутих алгоритмів.
5. Обраний екземпляр Products Service спершу перевіряє Redis-кеш; якщо даних там немає — звертається до бази даних, а потім зберігає результат у кеш.
6. Відповідь повертається через Gateway клієнту; конфігурація (адреса бази даних, налаштування кешу) для кожного кроку зчитувалась із централізованого сховища конфігурації під час старту сервісів.

Жоден із цих компонентів не є обов'язковим для щонайменшого проєкту — і, за принципом MonolithFirst (тема 12), додавати їх варто поступово, у міру реальної потреби: спершу, можливо, знадобиться лише кешування, потім — горизонтальне масштабування й балансувальник, і лише при появі десятків сервісів — повноцінний API Gateway і Service Discovery.

Питання для самоперевірки

- Які проблеми виникають у клієнта, якщо він звертається напямку до кожного мікросервіса, минаючи API Gateway?
- Для чого потрібен heartbeat у механізмі Service Discovery?
- Чому конфігурацію мікросервісів (особливо секрети) не можна просто зберігати в .env файлі, скопійованому на кожен сервер вручну?
- Наведіть по одному прикладу кешування на кожному з чотирьох розглянутих рівнів.
- Чому інвалідація кешу вважається складнішою задачею, ніж саме кешування даних?
- Чим горизонтальне масштабування принципово відрізняється від вертикального і чому мікросервіси зазвичай масштабують саме горизонтально?
- Чому властивість stateless є обов'язковою умовою для ефективного горизонтального масштабування?
- Порівняйте алгоритми балансування навантаження Round Robin і Least Connections — коли перевага буде в другого?

Короткий глосарій

Термін	Означення
API Gateway	Єдина точка входу, що маршрутизує зовнішні запити до внутрішніх сервісів
Service Discovery	Механізм, що дозволяє сервісам динамічно знаходити адреси одне одного
Service Registry	Реєстр адрес живих екземплярів сервісів
Heartbeat	Періодичне підтвердження сервісом власної працездатності реєстру
Інвалідація кешу	Своєчасне видалення застарілих даних із кешу після їх зміни
Горизонтальне масштабування	Збільшення потужності через запуск кількох копій сервіса
Вертикальне масштабування	Збільшення потужності через посилення одного сервера
Load balancer	Компонент, що розподіляє вхідні запити між кількома екземплярами сервіса
Stateless	Властивість сервіса не зберігати стан конкретного запиту в пам'яті процесу