

Дисципліна

«Бекенд-розробка, сервіси та мікросервіси»

КОНСПЕКТ ЛЕКЦІЇ. ТЕМА 12

Від моноліту до мікросервісів: принципи, переваги й недоліки, коли доцільно переходити

Спеціальність: Комп'ютерна інженерія, 3 курс
Обсяг: 1 год 20 хв (лекція 12 з 18)

План лекції

1. Монолітна архітектура: що це таке
2. Проблеми моноліту при зростанні застосунку
3. Мікросервісна архітектура: основні принципи
4. Переваги мікросервісів
5. Недоліки та складнощі мікросервісів
6. Комунікація між сервісами
7. Коли доцільно переходити на мікросервіси

Мета лекції: зрозуміти різницю між монолітною й мікросервісною архітектурами, усвідомлено оцінювати їхні переваги та недоліки, і навчитися визначати, коли перехід на мікросервіси виправданий, а коли лише додає складності без реальної користі.

1. Монолітна архітектура: що це таке

Моноліт (monolith) — це застосунок, увесь код якого (обробка HTTP-запитів, бізнес-логіка, доступ до бази даних) розгортається й виконується як єдиний процес. Саме такий застосунок ми будували протягом усього курсу: один PHP-проект із контролерами, моделями й поданнями, що звертається до однієї спільної бази даних.

```
project/
├── src/
│   ├── Controllers/ (Products, Orders, Users, Payments...)
│   ├── Models/
│   └── views/
└── одна база даних (MySQL)
```

→ Розгортається як ОДИН застосунок на ОДНОМУ (чи кількох однакових) сервері

Важливо розрізняти: моноліт не означає "поганий" чи "немасштабований" код — правильно спроектований моноліт з чіткою внутрішньою структурою (наприклад, за принципами MVC із теми 9) може роками успішно обслуговувати мільйони користувачів. Більшість успішних стартапів і сьогодні починають саме з моноліту, і це свідомий та часто правильний вибір.

Моноліт не є синонімом "спагеті-коду" з теми 9 — навпаки, добре структурований моноліт з чітким поділом на модулі (Products, Orders, Users) усередині одного застосунку є цілком нормальною й поширеною архітектурою, придатною для переважної більшості проектів.

2. Проблеми моноліту при зростанні застосунку

Проблеми монолітної архітектури зазвичай проявляються не одразу, а в міру зростання проєкту — і збільшення команди розробників, і зростання навантаження від користувачів.

Проблема	Прояв
Складність розгортання	Найменша зміна в одному модулі вимагає передеплою всього застосунку
Масштабування "всього і одразу"	Якщо навантажений лише модуль оплат, доводиться масштабувати весь моноліт цілком
Технологічна прив'язка	Увесь застосунок пишеться однією мовою й версією фреймворку
Конфлікти команди	Кілька команд одночасно змінюють один кодовий репозиторій — часті конфлікти злиття
Крихкість	Помилка в одному, навіть незначному, модулі може "вивести з ладу" весь застосунок
Довгий цикл збірки й тестів	З ростом кодової бази повний прогін тестів і збірка займають дедалі більше часу

Ці проблеми стають відчутними здебільшого у великих компаніях із десятками команд розробників, що працюють над одним продуктом (Amazon, Netflix, Uber — класичні приклади компаній, які перейшли від моноліту до мікросервісів саме через масштаб команди й навантаження).

3. Мікросервісна архітектура: основні принципи

Мікросервісна архітектура — підхід, за якого застосунок розбивається на набір невеликих, незалежних сервісів, кожен з яких відповідає за одну бізнес-можливість (наприклад, окремо: сервіс товарів, сервіс замовлень, сервіс користувачів), розгортається й масштабується незалежно від інших, і має власну базу даних.

Принцип	Суть
Єдина відповідальність сервісу	Кожен сервіс відповідає за одну бізнес-можливість (Products, Orders, Users)
Незалежне розгортання	Кожен сервіс можна оновити й розгорнути окремо, без зупинки інших
Децентралізовані дані	Кожен сервіс має власну базу даних; пряме звернення до чужої БД заборонено
Комунікація через мережу	Сервіси взаємодіють через API (HTTP/REST, повідомлення), а не прямі виклики функцій
Незалежність технологій	Різні сервіси можуть використовувати різні мови й бази даних

Той самий моноліт, розбитий на мікросервіси, міг би виглядати так: окремий сервіс Products (PHP + MySQL), окремий сервіс Orders (PHP чи інша мова + власна база даних), окремий сервіс Users/Auth (можливо, написаний іншою командою на іншій технології) — кожен розгортається й масштабується незалежно.

Принцип "власна база даних для кожного сервісу" — один із найважливіших і водночас найчастіше порушуваних на практиці. Якщо два "мікросервіси" продовжують звертатися до однієї спільної бази даних, вони насправді залишаються прихованим монолітом, лише розбитим на кілька процесів — це поширена помилка на початку переходу до мікросервісів.

4. Переваги мікросервісів

- Незалежне масштабування: сервіс товарів під час розпродажу можна масштабувати окремо від сервісу оплат, не витрачаючи ресурси на копіювання всього застосунку.
- Незалежне розгортання: команда сервісу Orders може випускати оновлення кілька разів на день, не чекаючи готовності команди Users.
- Ізоляція збоїв: якщо впаде сервіс рекомендацій, основний функціонал (перегляд каталогу, оформлення замовлення) може продовжувати працювати.
- Свобода вибору технологій: сервіс, що потребує високої обчислювальної швидкості, можна написати на Go чи Rust, залишивши решту на PHP.
- Менші, зрозуміліші кодові бази: новому розробнику простіше розібратися в одному невеликому сервісі, ніж у всьому моноліті одразу.

5. Недоліки та складнощі мікросервісів

Мікросервіси не є "безкоштовним покращенням" — вони обмінюють одні проблеми (складність зростаючого моноліту) на інші, часто менш очевидні на старті проєкту, але цілком реальні.

Складність	Опис
Розподілені транзакції	Операція, що зачіпає кілька сервісів (наприклад, оформлення замовлення), більше не може бути однією простою транзакцією БД (тема 7)
Мережева ненадійність	Виклик іншого сервісу через мережу може "зависнути" чи не дійти — потрібна обробка таймаутів і повторних спроб
Складність відлагодження	Один запит користувача може пройти через 5–10 сервісів — знайти, де саме сталася помилка, значно важче
Операційне навантаження	Замість одного застосунку потрібно розгортати, моніторити й підтримувати десятки окремих сервісів
Узгодженість даних	Дані про той самий об'єкт (наприклад, товар) можуть тимчасово розходитись між сервісами, доки не завершиться синхронізація
Складність тестування	Функціональний тест (тема 11) може вимагати одночасного запуску кількох сервісів

Мартін Фаулер, один із авторитетних голосів у галузі архітектури ПЗ, сформулював принцип "MonolithFirst": більшість команд повинні починати з моноліту і переходити до мікросервісів лише тоді, коли моноліт справді почав створювати проблеми, описані в розділі 2 — а не з самого початку проєкту "про всяк випадок".

6. Комунікація між сервісами

Оскільки сервіси — окремі процеси, що виконуються незалежно (часто на різних серверах), вони не можуть просто викликати функції одне одного напряму, як методи класів усередині моноліту. Існує два основних підходи до міжсервісної комунікації.

6.1. Синхронна комунікація: REST API

Найпростіший підхід — один сервіс звертається до іншого через REST API (тема 10), так само, як зовнішній клієнт звертався б до звичайного вебAPI, і синхронно чекає на відповідь.

```
// Сервіс Orders звертається до сервісу Products, щоб перевірити наявність товару
GET http://products-service/api/products/42

// Отримавши відповідь, Orders продовжує обробку замовлення
```

6.2. Асинхронна комунікація: черги повідомлень

Альтернативний підхід — сервіс не чекає на негайну відповідь, а публікує подію (message) у чергу повідомлень (наприклад, RabbitMQ чи Kafka), а інші сервіси, яким ця подія цікава, підписуються на неї й обробляють у власному темпі, незалежно від того, чи доступні вони саме в цю мілісекунду.

```
// Сервіс Orders публікує подію "замовлення створено"
publish('order.created', ['orderId' => 501, 'userId' => 42]);

// Сервіс Notifications підписаний на цю подію й надішле email,
// коли забажає обробити повідомлення з черги — незалежно від Orders
```

Підхід	Коли доцільний
Синхронний (REST)	Потрібна негайна відповідь для продовження обробки запиту користувача
Асинхронний (черга повідомлень)	Дію можна виконати з затримкою; важлива стійкість до тимчасової недоступності сервіса-отримувача

Надмірне використання синхронних викликів між сервісами створює приховану залежність (якщо Products недоступний, Orders теж перестав працювати) — цей ефект називають "розподіленим монолітом": формально сервіси окремі, але фактично так само жорстко зв'язані, як методи одного класу, тільки повільніше й ненадійніше через мережу.

7. Коли доцільно переходити на мікросервіси

Рішення про перехід на мікросервіси має ґрунтуватися на конкретних, вимірюваних проблемах поточного моноліту, а не на модній назві архітектурного стилю. Нижче — практичний орієнтир для оцінки.

7.1. Мікросервіси, ймовірно, ВИПРАВДАНІ, якщо:

- Кілька незалежних команд (5+ розробників кожна) регулярно заважають одна одній у спільному репозиторії.
- Окремі частини системи мають кардинально різні вимоги до масштабування (наприклад, каталог читають мільйони разів, а оплати обробляють тисячі).
- Потрібно розгортати оновлення окремих частин системи в різному темпі, не блокуючи одне одного.
- Команда вже має досвід і інфраструктуру для експлуатації розподілених систем (моніторинг, логування, оркестрація).

7.2. Мікросервіси, ймовірно, ЗАЙВИ, якщо:

- Проєкт лише починається чи має невелику команду (1–5 розробників) — операційні витрати переважають будь-яку користь.
- Моноліт поки не створює реальних проблем, описаних у розділі 2 — немає сенсу вирішувати проблему, якої ще немає.
- У команди немає досвіду роботи з розподіленими системами — ризик отримати "найгірше з обох світів" (складність мікросервісів без вигод) дуже високий.
- Бізнес-логіка застосунку ще активно змінюється й межі майбутніх сервісів не є очевидними.

Практична рекомендація для більшості навчальних і невеликих комерційних проєктів: будувати добре структурований, модульний моноліт (як у темі 9) і свідомо тримати межі між логічними модулями чіткими (Products, Orders, Users окремими просторами імен чи навіть окремими пакетами) — це залишає можливість виокремити частину в мікросервіс пізніше, коли (і якщо) виникне реальна потреба, значно дешевше, ніж проєктувати розподілену систему з першого дня.

Питання для самоперевірки

- Чи є моноліт синонімом поганого чи неструктурованого коду? Обґрунтуйте відповідь.
- Назвіть три проблеми, які моноліт створює саме при зростанні команди розробників (а не лише навантаження).
- Що означає принцип "децентралізованих даних" у мікросервісній архітектурі і чому його порушення робить сервіси "прихованим монолітом"?
- Наведіть приклад ситуації, де ізоляція збоїв мікросервісів дає реальну практичну перевагу.
- Чому розподілену транзакцію між кількома сервісами не можна реалізувати так само просто, як звичайну транзакцію бази даних із теми 7?
- Чим синхронна комунікація через REST відрізняється від асинхронної через чергу повідомлень? Наведіть по одному сценарію для кожної.
- Що таке "розподілений моноліт" і як його уникнути?
- Сформулюйте принцип "MonolithFirst" Мартіна Фаулера власними словами.

Короткий глосарій

Термін	Означення
Моноліт	Застосунок, що розгортається й виконується як єдиний процес
Мікросервіс	Невеликий незалежний сервіс, відповідальний за одну бізнес-можливість
Децентралізовані дані	Принцип, за яким кожен сервіс має власну, ізольовану базу даних
Синхронна комунікація	Взаємодія, за якої сервіс чекає на негайну відповідь (напр. REST)
Асинхронна комунікація	Взаємодія через чергу повідомлень без очікування негайної відповіді
Черга повідомлень	Проміжний компонент (RabbitMQ, Kafka) для передачі подій між сервісами
Розподілений моноліт	Формально окремі сервіси, фактично жорстко зв'язані через надмірні синхронні виклики
MonolithFirst	Принцип Мартіна Фаулера: починати з моноліту, переходити до мікросервісів за потреби