Факультет комп'ютерних наук та кібернетики
Київського національного університету
імені Тараса Шевченка

О.В. Галкін, О.С. Шкільняк

Вступ до мікросервісів

Конспект лекцій

Київ – 2025

УДК 004.41

Рецензенти:

доктор фіз.-мат. наук, проф. О.І. Провотар

кандидат фіз.-мат. наук, доц. О.В. Шишацька.

Рекомендовано до друку вченою радою факультету комп'ютерних наук та кібернетики (протокол №15 від 10 червня 2025 року)

Ухвалено науково-методичною комісією факультету комп'ютерних наук та кібернетики (протокол №10 від 5 червня 2025 року)

Галкін О.В., Шкільняк О.С.

Вступ до мікросервісів: конспект лекцій / Галкін О.В., Шкільняк О.С. – Київ, 2025. – 39 с.

Конспект лекцій покликаний надати початкове уявлення про підхід до розробки програмного забезпечення на основі мікросервісів. Головна увага приділяється шаблонам мікросервісної архітектури. Порівнюються два архітектурні стилі, монолітний та мікросервісний. Дається огляд шаблонів, які служать для декомпозиції на сервіси та призначених для роботи з даними.

Для студентів третього курсу факультету комп'ютерних наук та кібернетики Київського національного університету імені Тараса Шевченка, які навчаються за освітньо-професійною програмою «Програмна інженерія» спеціальності «Інженерія програмного забезпечення».

УДК 004.41

ЗМІСТ

ПЕРЕДМОВА

У сучасному світі дедалі більше компаній стикаються з необхідністю створювати масштабовані, гнучкі та легко підтримувані системи. Традиційні монолітні підходи, попри свою простоту, часто виявляються неефективними в умовах швидких змін вимог, високого навантаження та потреби в частих оновленнях.

З кожним роком мікросервіси продовжують набувати популярності як ефективна архітектурна парадигма для побудови складних застосунків. Вони передбачають створення системи як набору незалежних, спеціалізованих сервісів, які взаємодіють між собою та спільно реалізують бізнес-логіку великого програмного продукту. Такий підхід забезпечує високу гнучкість, масштабованість та стійкість до змін, що особливо цінно в умовах динамічної розробки.

Мікросервісна архітектура дозволяє розробникам оновлювати й підтримувати окремі компоненти без необхідності втручання у всю систему. Це спрощує процеси тестування, налагодження, а також прискорює впровадження змін і нових функцій. Крім того, така модель сприяє зниженню витрат, адже дає змогу повторно використовувати існуючі рішення, не починаючи розробку з нуля.

Саме тому важливо дати майбутнім інженерам цілісне уявлення про основи мікросервісів, їх переваги й недоліки, типові архітектурні рішення, а також практичні аспекти реалізації.

Конспект лекцій є вступом до мікросервісів. Основна увага приділяється шаблонам мікросервісної архітектури. Мова шаблонів

мікросервісів – набір методик, що допомагають в проєктуванні застосунків на основі мікросервісів.

Шаблони діляться на три рівні: інфраструктурні шаблони (розв'язують проблеми, пов'язані з інфраструктурою, але не з розробкою), інфраструктура застосунку (призначені для інфраструктурних задач, що впливають на розробку), шаблони застосунку (безпосередньо пов'язані з розробкою). Окрім рівнів, шаблони групуються відповідно до роду проблем, що вони розв'язують.

В рукописі дається огляд шаблонів рівня шаблонів застосунку: які служать для декомпозиції на сервіси та описують взаємодію між сервісами, в першу чергу при роботі з даними, а також порівнюються два архітектурні стилі, монолітний та мікросервісний. Інші види шаблонів планується описати в наступних роботах.

Конспект лекцій передусім спрямований на слухачів дисципліни «Об'єктно-орієнтоване програмування», що читається студентам третього курсу бакалаврату факультету комп'ютерних наук та кібернетики Київського національного університету імені Тараса Шевченка, які навчаються за освітньо-професійною програмою «Програмна інженерія» спеціальності «Інженерія програмного забезпечення». В силу відсутності україномовних видань з цієї тематики, рукопис може бути корисним для першого знайомства з мікросервісами для усіх охочих, хто займається розробкою програмного забезпечення.

1 АРХІТЕКТУРНИЙ СТИЛЬ

Уявімо, що ви розробляєте критично важливий бізнес-застосунок. Вам потрібно швидко, часто й надійно впроваджувати зміни, щоб ваш бізнес міг процвітати у сучасному складному і нестабільному світі. Організаційно, розробка ведеться невеликими, слабко зв'язаними, кросфункціональними командами, що працюють за принципами DevOps. Зокрема, використовується безперервне розгортання (Continuous Deployment), постійно вносячи малі, часті зміни, що тестуються за допомогою автоматизованого конвеєра розгортання і впроваджуються у продакшн.

Кожна команда відповідає за один або декілька піддоменів (subdomains).

Піддомен – це реалізована модель певної частини бізнес-функціоналу. Він складається з:

- бізнес-логіки, що містить бізнес-сутності (DDD-агрегати, Domain-Driven Design aggregates), які реалізують бізнес-правила;
- адаптерів, які взаємодіють із зовнішнім світом.

Наприклад, Java-піддомен складається з класів, організованих у пакети, які компілюються у JAR-файл.

Піддомени реалізують поведінку застосунку, яка включає набірсистемних операцій. Операція може викликатися трьома способами:

- синхронні та асинхронні запити від клієнтів;
- події, опубліковані іншими застосунками або сервісами;

- механізми на основі часу (наприклад, планувальники).

Операції можуть змінювати або запитувати бізнес-сутності в одному або кількох піддоменах.

Постає питання: як організувати піддомени у розгортовувані/виконувані компоненти? Слід обрати один з альтернативних архітектурних стилів (рис. 1):

- **Монолітна архітектура** – застосунок розгортається як єдиний цілісний блок.
- **Мікросервісна архітектура** – застосунок складається з незалежно розгортуваних, слабко зв'язаних сервісів.

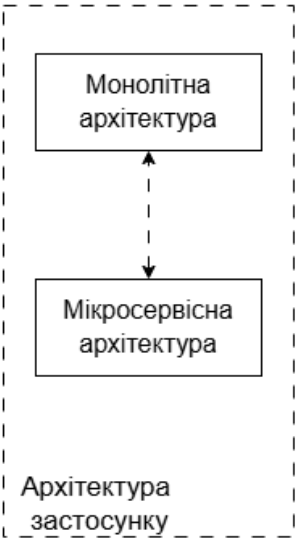

Рисунок 1. – Вибір архітектури

Розглянемо фактори вибору архітектури. До переваг моноліту можна віднести:

- прості взаємодії – локальні виклики швидші та надійніші, ніж розподілені комунікації (однак взаємодія між піддоменами може бути складною);
- ефективність – вся комунікація локальна, немає накладних витрат на мережеву взаємодію;
- ACID-транзакції (Atomicity, Consistency, Isolation, Durability – Атомарність, Узгодженість, Ізольованість, Довговічність) – простіше гарантувати атомарність, на відміну від BASE-моделі узгодженості в кінцевому рахунку (eventual consistency);
- мінімальну зв'язаність на рівні виконання – підвищує доступність і зменшує затримки операції;
- мінімальну зв'язаність на рівні розробки – знижує ризик необхідності внесення змін до декількох сервісів одночасно, немає необхідності синхронізувати API між сервісами.

В свою чергу, перевагами мікросервісів є:

- прості компоненти – невеликі компоненти легше зрозуміти й супроводжувати;
- автономія команд – команди можуть розробляти, тестувати та розгортати сервіси незалежно;
- швидкий конвеєр розгортання – швидший зворотний зв'язок і частіші деплойменти;
- підтримка кількох стеків технологій – різні піддомени можуть використовувати різні технології;

- розділення за характеристиками – піддомени можна розділити на окремі сервіси за різними характеристиками, щоб покращити, наприклад, масштабованість, безпеку чи доступність (availability) тощо.

1.1 Шаблон: Монолітна архітектура

Монолітна архітектура проєктується так, щоб застосунок був єдиним розгортовуваним/виконуваним компонентом**,** який використовує єдину базу даних**.** У цьому компоненті містяться всі піддомени застосунку. Оскільки є лише один компонент, всі операції виконуються локально. Приміром, подібний Java-застосунок може складатися з єдиного WAR-файлу, що виконується у вебконтейнері (Tomcat), а Rails-застосунок буде розгортатися як єдина структура каталогів, наприклад, у JRuby на Tomcat. Для масштабування та підвищення доступності можна запускати декілька екземплярів застосунку за балансувальником навантаження**.**

Однак особливості монолітної архітектури можуть породжувати певні проблеми:

- один великий компонент може стати складним у розумінні та підтримці**;**
- обмежена автономія команд розробників**:** вони змушені працювати в одному кодовому базисі й частіше координувати зміни;
- розгортання займає більше часу, оскільки оновлюється весь застосунок;

- усі піддомени використовують один стек технологій, що може бути неефективним;
- піддомени не можна розділити за характеристиками, що потенційно погіршує масштабованість, безпеку тощо.

Ці недоліки посилюються зі збільшенням розміру та складності застосунку і кількості команд розробників.

Деякі підходи допоможуть мінімізувати частину потенційних проблем. Зокрема, розбиття моноліту на модулі підвищить підтримуваність та автономність команд: замість традиційної багаторівневої архітектури, піддомени організуються у вертикальні зрізи, що містять засоби доступу, бізнес-логіку та логіку збереження даних. Конвеєр розгортання можна прискорити, врахувавши фізичну структуру модулів, використовуючи автоматизовані черги злиття, засоби збірки із підтримкою покрокових дій, шляхом розпаралелення та кластеризації процесів збірки і тестування.

Відомі приклади використання монолітної архітектури:

- багато великих компаній, такі як Netflix, Amazon, eBay**,** починали з монолітної архітектури**;**
- до 2012 року більшість вебзастосунків створювалися як моноліти**.**

1.2 Шаблон: Мікросервісна архітектура

Мікросервісна архітектура проєктується так, щоб застосунок складався з двох або більше незалежно розгортовуваних, слабко зв’язаних компонентів (сервісів).

Кожен сервіс містить один або кілька піддоменів. Піддомен може бути частиною лише одного сервісу, за винятком спільних бібліотечних піддоменів, які використовуються кількома сервісами. Сервіс належить команді (або командам), яка відповідає за відповідні (не бібліотечні) піддомени.

Зазвичай точкою входу до застосунку є API-шлюз. Деякі операції будуть локальними для одного сервісу, тоді як інші будуть розподіленими між кількома сервісами. Розподілені операції реалізуються за допомогою шаблонів колаборації сервісів (service collaboration patterns).

Щоб забезпечити незалежне розгортання, кожен сервіс зазвичай має власний репозиторій вихідного коду та власний конвеєр розгортання, який збирає, тестує та розгортає сервіс.

Мікросервісна архітектура теж має свої недоліки:

- деякі розподілені операції можуть бути складними й важкими для налагодження та розуміння**;**
- певні розподілені операції можуть бути неефективними;
- частину операцій доведеться реалізовувати за допомогою складного, неатомарного управління транзакціями (не-ACID), оскільки слабка зв’язаність вимагає окремої бази даних для кожного сервісу;
- деякі розподілені операції можуть передбачати тісну зв’язаність під час виконання, що зменшує їх доступність;
- існує ризик тісної зв’язаності під час розробки сервісів, що вимагає узгоджених змін**.**

При виборі мікросервісної архітектури головним викликом є визначити хорошу архітектуру сервісів**.** Необхідно уникати або хоча б мінімізувати такі потенційні недоліки, як наявність складних, неефективних взаємодій, складних неатомарних транзакцій та тісної зв'язаності під час виконання.

Ще одним ключовим моментом при використанні мікросервісів є реалізація розподілених операцій**,** які охоплюють кілька сервісів. Складності додає той факт, що кожен сервіс має свою базу даних**.** Рішенням є використання шаблонів колаборації сервісів**,** які реалізують розподілені операції як послідовність локальних транзакцій, серед яких:

- **Сага (Saga)** – реалізує розподілену команду як серію локальних транзакцій;
- **Репліка на боці команди (Command-side replica)** – реплікує дані у формі тільки для читання до сервісу, що виконує команду;
- **Композиція API (API Composition)** – реалізує розподілений запит як серію локальних запитів;
- **Розділення відповідальності командних запитів (CQRS)** – також реалізує розподілений запит як серію локальних запитів.

Шаблони **Сага (Saga)**, **Репліка на боці команди (Command-side replica)** та **Розділення відповідальності командних запитів (CQRS)** використовують асинхронний обмін повідомленнями**.** Сервіси зазвичай використовують шаблон **Transaction Outbox,** щоб атомарно оновлювати постійні бізнес-об'єкти та надсилати повідомлення.

Відомі приклади використання мікросервісної архітектури:

- Netflix – має масштабовану сервіс-орієнтовану архітектуру, обробляючи понад 1 мільярд запитів на день**;**
- Amazon – спочатку використовував двошарову архітектуру**,** але перейшов на сотні мікросервісів**;**
- eBay – розбив систему на багато незалежних застосунків, кожен з яких реалізує бізнес-логіку окремого функціоналу.

Ці компанії, як і багато інших, еволюціонували від моноліту до мікросервісів**.**

1.3 Висновки

Отже, якщо йдеться про вибір архітектури застосунку, моноліт краще обрати, якщо:

- важлива простота взаємодії між модулями;
- бізнес-логіка тісно пов'язана, і багато змін зачіпають одразу весь застосунок;
- потрібне просте розгортання без складного оркестрування мікросервісів;
- команда розробників маленька, і немає потреби в автономії окремих частин.

Мікросервісна архітектура буде більш вдалим вибором у випадку, якщо:

- потрібно часто змінювати та оновлювати окремі частини без перезапуску всього застосунку;

- декілька команд розробників працюють незалежно, кожна відповідає за свій сервіс;
- є піддомени з різними технологічними вимогами (наприклад, ML-сервіс на Python, а основна система – на Java);
- масштабування критичне, і різні сервіси мають різні вимоги до навантаження.

Крім того, існують і проміжні варіанти. Компромісним рішенням може бути модульний моноліт: це спрощує взаємодію, але дозволяє поступово виділяти сервіси. Гібридні підходи теж можливі, наприклад, мікросервісна архітектура для критичних піддоменів та моноліт для решти.

2 ДЕКОМПОЗИЦІЯ

Припустимо, створюється велика, складна програма з використанням мікросервісної архітектури, тобто застосунок буде організований як набір слабко зв’язаних сервісів. Мета мікросервісної архітектури – прискорити розробку програмного забезпечення, забезпечуючи безперервну інтеграцію та розгортання. Це досягається двома способами:

- спрощення тестування та можливість незалежного розгортання компонентів;
- організація інженерної команди як набору невеликих (6–10 осіб), автономних груп розробників, кожна з яких відповідає за один або кілька сервісів.

Однак ці переваги не гарантуються автоматично. Вони можуть бути досягнуті лише за умови ретельної функціональної декомпозиції застосунку на сервіси (рис. 2). Буде корисним звернутися до деяких принципів об’єктно-орієнтованого проєктування (Object-oriented design, OOD).

Сервіс повинен бути достатньо малим, щоб його могла розробляти невелика команда та щоб його було легко тестувати. Принцип єдиної відповідальності (Single-responsibility principle, SRP) визначає відповідальність класу як причину для його змінення та стверджує, що клас повинен мати лише одну таку причину. Логічно застосувати цей принцип і до проєктування сервісів: сервіси мають бути зв’язаними та реалізовувати невеликий набір тісно пов’язаних функцій.

Рисунок 2. – Шаблони декомпозиції

Застосунок також слід декомпозувати так, щоб більшість нових і змінених вимог стосувалися лише одного сервісу. Це важливо, оскільки зміни, що зачіпають кілька сервісів, вимагають координації між командами, що уповільнює розробку. Ще один принцип OOD – Принцип узгодженої зміни (Common Closure Principle, CCP) – стверджує, що класи, які змінюються з однієї причини, мають бути в одному пакеті. Наприклад, два класи можуть реалізовувати різні аспекти одного бізнес-правила. Мета – зробити так, щоб при зміні цього правила розробникам довелося змінювати код лише в одному або в дуже малій кількості пакетів. Такий підхід корисний і для

проєктування сервісів, оскільки допомагає зробити так, щоб кожна зміна зачіпала лише один сервіс.

Тож як розділити застосунок на сервіси? При проєктуванні архітектури необхідно враховувати наступне:

- архітектура має бути стабільною;
- сервіси мають бути зв'язаними: кожен сервіс повинен реалізовувати невеликий набір тісно пов'язаних функцій;
- сервіси мають відповідати Принципу узгодженої зміни: компоненти, які змінюються разом, мають бути разом згруповані, щоб кожна зміна зачіпала лише один сервіс;
- сервіси мають бути слабко зв'язаними: кожен сервіс має API, що інкапсулює його реалізацію, яка може змінюватися без впливу на клієнтів;
- сервіс має бути тестованим;
- кожен сервіс має бути достатньо малим, щоб його могла розробляти команда з 6–10 осіб (т. зв. «команда на дві піци»);
- кожна команда, відповідальна за одним або декілька сервісів, має бути автономною; команда повинна мати змогу розробляти та розгортати свої сервіси з мінімальною взаємодією з іншими командами.

2.1 Шаблон: Декомпозиція за бізнес-можливостями (Decompose by business capability)

Визначаються сервіси, що відповідають бізнес-можливостям. Бізнес-можливість — це поняття з моделювання бізнес-архітектури,

яке описує те, чим займається бізнес. Зазвичай бізнес-можливість відповідає певному бізнес-об'єкту, наприклад:

- управління замовленнями відповідає за замовлення;
- управління клієнтами відповідає за клієнтів.

Бізнес-можливості часто організовуються в багаторівневу ієрархію. Наприклад, корпоративний застосунок може мати такі категорії вищого рівня, як Розробка продуктів/послуг, Доставка продуктів/послуг, Формування попиту тощо. Відповідна мікросервісна архітектура матиме сервіси, що відповідають кожній із цих бізнес-можливостей.

Підхід має наступні переваги:

- стабільна архітектура**,** оскільки бізнес-можливості є відносно стабільними;
- кросфункціональні та автономні команди розробників**,** організовані навколо створення бізнес-цінності, а не технічних аспектів;
- зв’язані та слабко взаємопов’язані сервіси**.**

Основний виклик, що виникає при використанні цього підходу, – як визначити бізнес-можливості: для цього потрібно розуміти бізнес. Бізнес-можливості організації визначаються шляхом аналізу її цілей, структури, бізнес-процесів і сфер експертизи. Визначення меж контекстів (bounded contexts) найкраще виконувати ітеративно. На що спочатку можна орієнтуватися при визначенні бізнес-можливостей:

- організаційна структура: різні відділи можуть відповідати бізнес-можливостям або групам бізнес-можливостей;

- високорівнева модель домену: бізнес-можливості часто відповідають об'єктам домену.

2.2 Шаблон: Декомпозиція за піддоменами (Decompose by subdomain)

Сервіси визначаються відповідно до піддоменів в межах підходу предметно-орієнтованого проєктування (Domain-Driven Design, DDD). DDD розглядає предметну область застосунку – бізнес – як домен. Домен складається з декількох піддоменів. Кожен піддомен відповідає певній частині бізнесу.

Піддомени можна класифікувати наступним чином:

- ядро (Core) – ключова складова бізнесу, найцінніша частина застосунку;
- допоміжні (Supporting) – пов’язані з бізнесом, але не є його основною складовою; можуть бути реалізовані як внутрішньо, так і через аутсорсинг;
- загальні (Generic) – не специфічні для бізнесу, їх найкраще реалізовувати за допомогою готового програмного забезпечення.

Відповідна мікросервісна архітектура матиме сервіси, що відповідають кожному з цих піддоменів.

Переваги підходу:

- стабільна архітектура, оскільки піддомени є відносно стабільними;

- команди є кросфункціональними, автономними та організованими навколо бізнес-цінності, а не технічних особливостей;
- сервіси зв’язані та слабко взаємопов’язані між собою.

При використанні цього підходу виникають наступні питання:

- як визначити піддомени? Для цього потрібно розуміти бізнес. Подібно до бізнес-можливостей, піддомени визначаються шляхом аналізу бізнесу, його організаційної структури та виявлення різних зон експертизи. Визначення піддоменів є ітеративним процесом.

Хороші відправні точки при визначенні піддоменів:

- організаційна структура – різні відділи можуть відповідати піддоменам;
- високорівнева модель домену – піддомени часто мають ключовий об’єкт домену.

2.3 Шаблон: Самодостатній сервіс (Self-contained service)

Розглянемо питання взаємодії сервісу з іншими сервісами під час обробки синхронного запиту. Стандартний підхід з використанням використання синхронної взаємодії зменшує доступність (неможливо обробити запит, якщо один із сервісів недоступний). Які фактори слід враховувати в такому випадку:

- мікросервісна архітектура розподіляє обробку запиту між кількома сервісами;

- операція повинна мати високу доступність та низький час відповіді;
- доступність операції залежить від доступності всіх сервісів, що викликаються синхронно;
- можливе повторення запитів до недоступного сервісу, але це збільшує час відповіді.

Рішенням є усунути синхронну взаємодію: сервіс повинен відповідати на синхронний запит**,** не чекаючи відповіді від інших сервісів. Є два підходи для створення самодостатнього сервісу:

- об'єднання декількох сервісів в один сервісний модуль з потрібним функціоналом;
- взаємодія з іншими сервісами з використанням шаблонів **CQRS** (дозволяє сервісу зберігати локальні репліки даних інших сервісів) та **Saga** (забезпечує асинхронну узгодженість даних між сервісами).

Перевагами шаблону є покращення доступності й швидший час відповіді, оскільки сервісу не доводиться чекати на завершення всіх дій. Однак є і певні недоліки:

- додаткові витрати на підтримку **CQRS**;
- складність використання **Saga**;
- у випадку використання **Saga**, API стає менш очевидним;
- збільшений розмір сервісу внаслідок перенесення до нього частини логіки інших сервісів.

2.4 Шаблон: Сервіс на команду (Service per team)

Високопродуктивна розробка вимагає поділу на команди. Кожна команда є довгоживучою, малою (зазвичай 6–10 осіб), слабко зв’язаною, автономною і кросфункціональною. Закон Конвея стверджує, що архітектура віддзеркалює структуру комунікації організації, яка її створює. Яким буде оптимальне співвідношення між сервісами й командами у випадку архітектури мікросервісів?

Кращий підхід – модель власності коду/сервісу. Кожен сервіс належить одній команді, яка має повну відповідальність за його зміни: самостійно розробляти, тестувати, розгортати та масштабувати. Основна взаємодія з іншими командами відбувається під час узгодження API.

В ідеалі, кожна команда має лише один сервіс. Команда відповідає за одну або кілька бізнес-функцій (можливостей). Вона володіє кодовою базою (один або кілька модулів), розмір і складність якої не перевищують когнітивну здатність команди. Команда розгортає цю кодову базу як один або кілька сервісів, але має мати лише один сервіс, якщо тільки немає доведеного обґрунтування мати більше. Як наслідок, існує верхня межа для розміру та/або складності сервісу.

Загалом, шаблон надає наступні переваги:

- команди є автономними і можуть працювати з мінімальною координацією;
- команди слабко зв’язані;

- досягається автономія й слабкий зв’язок з мінімальною кількістю сервісів;
- покращується якість коду завдяки довготривалій відповідальності за нього.

Серед недоліків підходу:

- команди не обов’язково узгоджені з користувацькими функціями;
- реалізація функцій, що охоплюють кілька сервісів, ускладнюється і потребує співпраці команд.

2.5 Висновки

Отже, для розділення застосунку на сервіси можна скористатися такими підходами:

- декомпозиція за бізнес-можливостями – сервіси визначаються відповідно до бізнес-можливостей;
- декомпозиція за піддоменами – сервіси визначаються відповідно до піддоменів (DDD);
- самодостатній сервіс – сервіси проєктуються так, щоб вони відповідали на синхронні запити без очікування інших сервісів;
- сервіс на команду – кожна команда володіє одним сервісом.

Вибір конкретного підходу залежить від контексту та цілей архітектури.

3 ВЗАЄМОДІЯ СЕРВІСІВ

Як правило, більшість сервісів мають зберігати дані в певній базі даних. Розглянемо, якою може бути архітектура бази даних в мікросервісному застосунку (рис. 3). Основні чинники, які слід при цьому враховувати:

- сервіси повинні бути слабко зв’язаними, щоб їх можна було розробляти, розгортати та масштабувати незалежно;
- деякі бізнес-транзакції повинні забезпечувати узгодженість даних у кількох сервісах (наприклад, перевірка кредитного ліміту клієнта перед створенням замовлення);
- деякі бізнес-транзакції повинні оновлювати дані, що належать різним сервісам;
- деякі запити потребують об’єднання (join) даних з кількох сервісів;
- бази даних іноді потрібно реплікувати або розподіляти для масштабування;
- різні сервіси можуть мати різні вимоги до зберігання даних (SQL, NoSQL тощо).

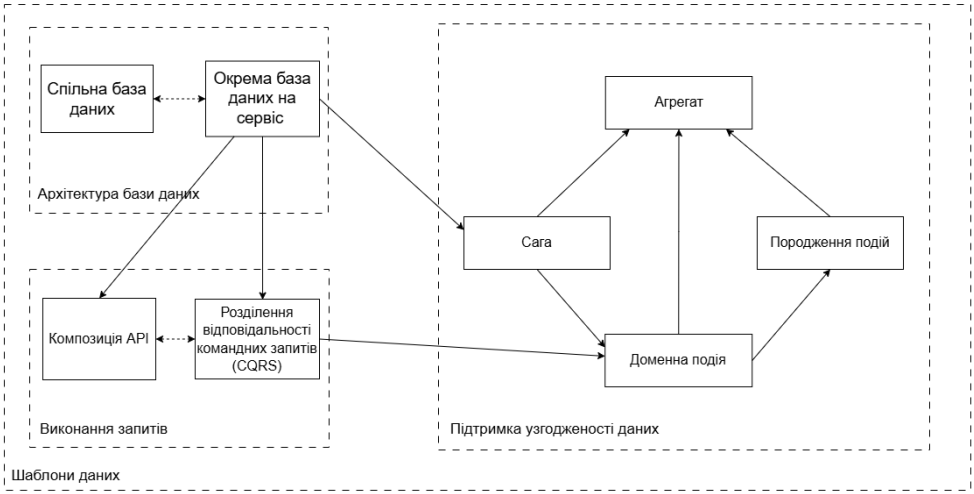

Рисунок 3. – Шаблони даних

3.1 Шаблон: Окрема база даних на сервіс (Database per Service)

Один з підходів пропонує, щоб кожен мікросервіс мав власну базу даних, доступну лише через його API. Транзакції сервісу використовують тільки його базу даних. Власне, вона є частиною імплементації цього сервісу, безпосередній доступ до бази даних з інших сервісів неможливий.

Існує кілька способів забезпечити приватність даних сервісу. Зокрема, у випадку реляційної бази даних, можливі наступні варіанти:

- окремі таблиці на сервіс (Private-tables-per-service) – кожен сервіс володіє власним набором таблиць;

- окрема схема на сервіс (Schema-per-service) – кожен сервіс має власну схему бази даних;
- окремий сервер бази даних на сервіс (Database-server-per-service) – кожен сервіс використовує власний сервер баз даних.

Окремі таблиці або схеми є менш витратними, причому використання окремої схеми робить більш зрозумілою приналежність до конкретного сервісу. Окремий сервер може бути необхідний для високонавантажених сервісів.

Щоб примусово забезпечити ізоляцію даних, можна використовувати різні облікові записи бази даних і механізми контролю доступу. Додатковий бар'єр дозволить прибрати спокусу прямого доступу до даних в обхід API сервісу.

Використання окремої бази даних для кожного сервісу має наступні переваги:

- забезпечується слабка залежність між сервісами, зміни в одній базі даних не впливають на інші сервіси;
- кожен сервіс може вибирати найбільш підходящу базу даних (наприклад, ElasticSearch для текстового пошуку, Neo4j для графових даних тощо).

Недоліки, які виникають при цьому:

- ускладнюється реалізація бізнес-транзакцій, що охоплюють кілька сервісів;
- ускладнюється виконання запитів, які об'єднують дані з кількох сервісів;
- підвищується складність управління різними базами даних (SQL, NoSQL).

CAP-теорема (теорема Брюера) стверджує, що для будь-якої розподіленої системи неможливо одночасно забезпечити виконання більше двох із трьох властивостей: узгодженість даних, доступність, стійкість до розділення. Тому при розділенні мережі треба вибирати між узгодженістю та доступністю. Таким чином, розподілених транзакцій краще уникати; ряд сучасних баз даних їх не підтримують (зокрема, NoSQL).

3.2 Шаблон: Спільна база даних (Shared Database)

Альтернативним підходом до зберігання даних в мікросервісному застосунку може стати використання спільної бази даних, до якої мають доступ кілька сервісів. Кожен сервіс може вільно звертатися до даних, що належать іншим сервісам, використовуючи локальні ACID-транзакції.

Це приносить певні зручності:

- спрощена робота з даними – розробники можуть використовувати знайомі ACID-транзакції для забезпечення узгодженості;
- простота операційної підтримки – використання єдиної бази даних спрощує адміністрування.

Але є й суттєві недоліки:

- залежність на рівні розробки – зміну схеми бази даних потрібно узгоджувати між командами, що сповільнює розробку;

- залежність на рівні виконання – при використанні однієї бази даних сервіси потенційно можуть заважати один одному: наприклад, якщо деякий сервіс блокує таблицю в базі даних, інші сервіси можуть бути заблоковані;
- обмеження щодо вибору технологій – єдина база даних може не задовольняти всі вимоги сервісів (наприклад, деякі сервіси можуть потребувати NoSQL).

Слід зазначити, що використання спільної бази даних в контексті мікросервісної архітектури загалом вважається антишаблоном, бо може привнести більше складності і обмежень та в подальшому призвести до втрати переваг мікросервісів.

3.3 Шаблон: Сага (Saga)

Припустимо, було застосовано архітектуру мікросервісів та використано шаблон **Окрема база даних на сервіс (Database per Service)**, і кожен сервіс має власну базу даних. Проте деякі бізнес-транзакції охоплюють кілька сервісів, тому необхідний механізм для їх реалізації. Як можна реалізувати транзакції, що охоплюють кілька сервісів?

Двофазна транзакція (2PC, Two-Phase Commit) не є підходящим варіантом**,** оскільки ускладнює масштабування та може порушувати незалежність сервісів.

В такому випадку кожна бізнес-транзакція реалізується як сага. Сага – це послідовність локальних транзакцій. Кожна локальна транзакція оновлює базу даних і публікує подію або повідомлення, яке

запускає наступну транзакцію в сага-процесі. Якщо одна з локальних транзакцій зазнає невдачі, сага виконує компенсуючі транзакції, які скасовують зміни, зроблені попередніми транзакціями.

Існують два способи координації саг:

1. Хореографія (Choreography) – кожна локальна транзакція публікує події, які тригерять наступні локальні транзакції в інших сервісах.
2. Оркестрація (Orchestration) – спеціальний об'єкт-оркестратор керує виконанням локальних транзакцій у сазі.

Перевагами саг на основі хореографії є простота та слабка зв'язаність. Серед недоліків – складність розуміння (реалізація розкидана між сервісами), поява небажаних циклічних залежностей між сервісами, ризик отримати тісну зв'язаність (через необхідність підписки сервісом на всі події, що на нього впливають). Хореографія може добре працювати з простими сагами, але з огляду на її недоліки, для складніших випадків краще використовувати оркестрацію.

Саги на основі оркестрації мають декілька переваг:

- спрощені залежності – оскільки оркестратор викликає учасників, але не навпаки, циклічні залежності не з'являться;
- менша зв'язаність – сервісам не потрібно знати про події, що публікуються іншими учасниками саги;
- покращене розділення відповідальності і спрощення бізнес-логіки – вся логіка координації знаходиться в оркестраторі.

Єдиним недоліком оркестрації є ризик надлишкової централізації бізнес-логіки в оркестраторі. Однак, цього можна уникнути, якщо проєктувати окрестратори так, щоб вони не містили

зайвої бізнес-логіки і відповідали лише за послідовність виконання дій. Використання оркестрації є кращим підходом для загального випадку, окрім найпростіших саг.

Перевагою використання шаблону **Saga** є те, що дані залишаються узгодженими між сервісами без необхідності в розподілених транзакціях.

Недоліками можна назвати:

- відсутність автоматичного відкату – розробник має явно реалізувати компенсуючі транзакції;
- відсутність ізоляції («I» в ACID) – конкурентне виконання саг і транзакцій може призвести до аномалій даних, що вимагає використання спеціальних стратегій контрзаходів для реалізації ізоляції.

Окрім того, є додаткові моменти, які слід врахувати.

Для гарантування надійності, сервіс має атомарно оновлювати свою базу даних та публікувати повідомлення/подію. За відсутності традиційного механізму розподілених транзакцій, має використовуватися один з шаблонів, **Event Sourcing** або **Transactional Outbox**.

Для того, щоб клієнт міг дізнатися результат саги використовуючи синхронний запит, можна обрати одну з опцій: очікування завершення саги, періодичне опитування чи асинхронне сповіщення.

3.4 Шаблон: Репліка на боці команди (Command-side Replica)

Нехай було застосовано архітектуру мікросервісів та використано шаблон **Окрема база даних на сервіс (Database per Service)** і сервіс, що виконує команду, потребує даних від іншого сервісу. Можна кожного разу робити новий запит (синхронний або асинхронний як крок саги), але це має ряд недоліків. Альтернативний підхід – використати репліку бази даних.

Шаблон **Репліка на боці команди (Command-side Replica)** складається з наступних елементів:

- сервіс команди (Command Service) – імплементує команду;
- сервіс постачальника (Provider Service) – має дані, потрібні сервісу команди;
- репліка бази баних (Replica Database) – доступна тільки для читання копія даних сервісу постачальника, яка оновлюється через підписку на доменні події (Domain Event), публіковані сервісом постачальника.

Шаблон має такі переваги:

- репліка бази даних може використовувати інший стек технологій, краще оптимізований для підтримки команди;
- сервіс постачальника більше не викликається командою, тому може не потребувати такої ж продуктивності чи доступності;
- команда спрощується, бо їй більше не потрібно звертатися до сервісу постачальника;
- команда ефективніша, оскільки більше не залежить від взаємодії із сервісом постачальника;

- репліка може містити застарілі дані;
- зменшується зв’язування на етапі виконання, бо команда більше не взаємодіє з сервісом постачальника.

При цьому виникають наступні проблеми:

- ускладнюються сервіси команди та постачальника через необхідність підтримувати репліку і публікувати події;
- через підвищене зв’язування на етапі проєктування розробникам сервісів можливо доведеться координуватися частіше;
- розгортання сервісу команди може стати повільнішим, оскільки потрібно тестувати репліку;
- взаємодії потенційно менш ефективні, якщо сервіс-провайдер публікує великий обсяг подій;
- існує ризик тісного зв’язування на етапі проєктування між сервісами, оскільки сервіс команди має знати структуру реплікованих даних і, можливо, події їх життєвого циклу.

3.5 Шаблон: Композиція API (API Composition)

Припустимо, було застосовано архітектуру мікросервісів та використано шаблон **Окрема база даних на сервіс (Database per Service)**. Як наслідок, реалізація запитів, що об’єднують дані з кількох сервісів, стала складнішою. Як в такому випадку реалізувати запити в архітектурі мікросервісів?

Можна визначити API-композитора, який буде реалізовувати запит, звертаючись до сервісів, які володіють даними, та

виконуватиме об’єднання результатів у пам’яті. API-композитор може бути як клієнтом, так і сервісом.

Перевагою шаблону є простота такого способу виконання запитів у мікросервісній архітектурі. Однак, деякі запити можуть призводити до неефективних об’єднань великих наборів даних у пам’яті.

3.6 Шаблон: Розділення відповідальності командних запитів (Command Query Responsibility Segregation, CQRS)

Припустимо, було застосовано архітектуру мікросервісів та використано шаблон **Окрема база даних на сервіс (Database per Service)**. Як наслідок, реалізація запитів, що об’єднують дані з кількох сервісів, стала складнішою. Якою буде альтернатива у випадку, якщо запит за допомогою композиції API неефективний? Більше того, якщо також використовується породження подій (event sourcing), то дані більше не можна запитувати напряму.

Тоді можна визначити доступну тільки для читання базу даних – репліку, спеціально створену для підтримки конкретного запиту або групи запитів. Актуальність інформації підтримується шляхом підписки на доменні події (Domain Event), що публікуються сервісом, який володіє даними. При цьому схема та тип бази даних оптимізуються для запиту (зазвичай це NoSQL).

Сильні сторони **CQRS**:

- підтримка денормалізованих представлень, які легко масштабуються та працюють ефективно;
- краща розділеність обов'язків, що спрощує моделі для команд та запитів;
- необхідний для обробки запитів в архітектурі на основі породження подій (Event Sourcing).

Недоліки підходу:

- збільшення складності через підтримку додаткових баз даних;
- можливе дублювання коду;
- затримка реплікації: представлення даних не завжди актуальне (узгодженість в кінцевому рахунку – eventual consistency).

3.7 Шаблон: Доменна подія (Domain Event)

Часто інші сторони взаємодії зацікавлені в інформації про зміни в певному сервісі: він повинен публікувати події, як тільки оновлює свої дані. Яким чином сервіс може публікувати події щоразу, коли змінює свої дані?

Бізнес-логіка сервісу організовується як набір агрегатів (кластери об’єктів, з якими можна поводитися як з одним цілим). Коли їх створюють або оновлюють, агрегати генерують доменні події. Сервіс публікує ці події, щоб інші сервіси могли їх обробити.

3.8 Шаблон: Породження подій (Event Sourcing)

Команда сервісу зазвичай має створювати, оновлювати або видаляти агрегати в базі даних та надсилати події або повідомлення в брокер повідомлень. Аналогічно, сервіс, який публікує доменну подію (Domain Event), має оновити агрегат й опублікувати подію.

Проблема: як атомарно оновити базу даних і надіслати повідомлення в брокер повідомлень?

Двофазна транзакція (2PC, Two-Phase Commit) не є підходящим варіантом: база даних і брокер можуть його не підтримувати, також це створює часом небажану залежність між сервісом, базою даних і брокером.

Окрім того, потрібно, щоб повідомлення надсилалися до брокера у тому порядку, в якому вони були надіслані сервісом; цей порядок має зберігатися між різними екземплярами сервісу, що оновлюють один і той самий агрегат.

Правильним рішенням проблеми було б використання породження подій (event sourcing). Замість безпосередньої зміни стану, стан бізнес-об’єкту зберігається як послідовність подій:

- кожна зміна стану – це новий запис у сховищі подій (event store);
- збереження події – атомарна за своєю природою операція;
- стан об’єкта відновлюється шляхом програвання (replaying) всіх подій;
- сховище подій діє як брокер повідомлень, дозволяючи підписникам отримувати події;

- якщо агрегат містить багато подій, можна періодично зберігати знімки поточного стану (snapshots); щоб відновити стан, переглядаються лише події після найостаннішого знімка.

Породження подій має кілька переваг:

- розв'язується одна з ключових проблем реалізації подіє-орієнтованої архітектури та забезпечує надійну публікацію подій щоразу, коли змінюється стан;
- оскільки зберігаються події, а не об'єкти предметної області, це переважно дозволяє уникнути проблеми об'єктно-реляційного імпедансного розриву;
- забезпечує 100% достовірний журнал змін (audit log) стану бізнес-об'єкта;
- дає змогу реалізовувати часові запити, які визначають стан об'єкта у будь-який момент часу;
- бізнес-логіка, побудована на породженні подій, складається зі слабко зв'язаних бізнес-об'єктів, які взаємодіють через події, що значно полегшує перехід від монолітного застосунку до мікросервісної архітектури.

Проте, присутні і недоліки:

- це незвичний стиль програмування, тому потребує часу на навчання;
- до сховища подій складно робити запити, оскільки для звичайних запитів потрібно реконструювати стан об'єктів, що може бути складним і неефективним; через це для реалізації запитів слід використовувати **CQRS**, що означає роботу з даними, які узгоджуються не миттєво (eventual consistency).

3.9 Висновки

Отже, для реалізації операцій, які охоплюють кілька сервісів можна використати один з наступних підходів:

- окрема база даних на сервіс (Database per Service) – кожен сервіс має власну приватну базу даних;
- спільна база даних (Shared Database) – сервіси використовують одну базу даних;
- сага (Saga) – підтримання узгодженості даних за допомогою послідовності локальних транзакцій;
- репліка на боці команди (Command-side Replica) – створення копії даних у сервісі, який виконує команди;
- композиція API (API Composition) – виконання запитів до різних сервісів та об’єднання результатів у пам’яті;
- розділення відповідальності командних запитів (CQRS – Command Query Responsibility Segregation) – окремі структури для запису та читання даних;
- доменна подія (Domain Event) – публікація подій щоразу, коли змінюються дані;
- породження подій (Event Sourcing) – збереження змін у вигляді послідовності подій.

РЕКОМЕНДОВАНІ ДЖЕРЕЛА

1. Chris Richardson. Microservices Patterns. – Manning, 2018. – 520 p.

2. A pattern language for microservices (by Chris Richardson). *Microservices.io*. URL: https://microservices.io/patterns/index.html

3. Sam Newman. Building Microservices, 2nd Edition. – O'Reilly Media, 2021. – 612 p.

4. Sam Newman. Monolith to Microservices. – O'Reilly Media, 2019. – 270 p.

5. Роберт Мартін. Чиста архітектура. – Фабула, 2019. – 368 с.

6. Eric Evans. Domain-Driven Design: Tackling Complexity in the Heart of Software. – Addison-Wesley Professional, 2003. – 560 p.