Engineer

Мікросервіси

архітектура з малих сервісів, що розгортаються окремо

Що це таке

Мікросервіси — це архітектурний стиль, який розбиває один застосунок на малі сервіси. Кожен сервіс — окрема програма, що працює як самостійний процес, і її можна розгорнути, не чіпаючи решти. Джеймс Льюїс (James Lewis) і Мартін Фаулер (Martin Fowler), чия стаття 2014 року дала стилю найвідоміший опис, кажуть про такі застосунки як про «набори сервісів, які можна розгортати незалежно один від одного». Вони перелічують, що зазвичай об'єднує такі системи: сервіси обмінюються даними через легкі механізми, найчастіше через HTTP API ресурсів; їх будують навколо бізнес-можливостей; розгортання повністю автоматизоване; централізованого керування — мінімум, а самі сервіси можуть бути написані різними мовами програмування й використовувати різні технології зберігання даних. Автори окремо застерігають, що формального визначення немає: є лише риси, які трапляються в більшості систем під цією назвою.

Слово молодше за ідею. Виноска в тій самій статті розповідає, що в травні 2011 року термін обговорювали на семінарі архітекторів програмного забезпечення поблизу Венеції, а в травні 2012 року та сама група зупинилася на назві «microservices». Адріан Кокрофт (Adrian Cockcroft) у Netflix називав цей підхід «fine grained SOA», а Льюїс і Фаулер не стверджують, що стиль новий: його коріння вони знаходять щонайменше в принципах, на яких побудовано Unix.

Протиставляють йому зазвичай моноліт — серверний застосунок, який збирають і розгортають як одне ціле. Уся логіка обробки запиту виконується в одному процесі. Якщо трохи подбати, весь застосунок можна запустити й протестувати на ноутбуці одного розробника, а масштабують його, запускаючи більше копій за балансувальником навантаження. Тертя

з’являється згодом. Невелика зміна вимагає заново зібрати й розгорнути все, добру модульну структуру важко втримати роками, а частину, якій бракує потужності, можна масштабувати лише разом з усім іншим.

Компонентом Льюїс і Фаулер називають частину програмного забезпечення, яку можна замінити й оновити окремо. Бібліотеку підключають до програми і викликають у пам’яті. Сервіс — це компонент поза процесом, до якого звертаються мережею: вебзапитом або віддаленим викликом процедури. Для зв’язності між частинами системи ця різниця важлива. У більшості мов лише документація й дисципліна не дають іншому коду залізти в нутрощі бібліотеки, а до сервісу можна дістатися тільки явними віддаленими викликами, тож його інтерфейс значно важче обійти. Утім, сервіс може складатися й з кількох процесів, наприклад із застосунку та бази даних, якою користується лише він.

Префікс «мікро» каже менше, ніж здається. Льюїс і Фаулер зауважують, що назва привертає увагу саме до розміру й породжує суперечки про те, що вважати «мікро». Найбільші сервіси, про які вони чули, відповідали правилу Amazon про команду на дві піци, тобто не більше дюжини людей, а на іншому кінці шкали вони бачили команду з шести осіб, яка підтримувала шість сервісів.

Застосуйте поняття до рішення

Застосуйте «Що таке мікросервіси?» до конкретного рішення. Ми можемо пов'язати поняття із системою, яку ви створюєте або купуєте. Та частина, якої користувач не бачить: де зберігаються дані, які діють правила і як інші системи можуть із нею взаємодіяти.

Переглянути наші послуги Зв'язатися з нами →

Перевірте припущення за терміном

Використайте пояснення та пов'язані роботи, щоб сформулювати запитання для проєкту. Ми можемо перевірити припущення разом до вибору інструмента. Спроєктувала й підтримувала 6 пайплайнів ETL/ELT — 6 пайплайнів ETL на трьох рушіях баз даних

Спроєктувала й підтримувала 6 пайплайнів ETL/ELT Зв'язатися з нами →

Чому це важливо

Перший виграш — власний графік випусків. Коли застосунок поділено на сервіси, багато змін зачіпає лише один із них, і заново розгорнути треба тільки його, тож команда випускає зміну, не чекаючи на інших. Льюїс і Фаулер уточнюють, що це не абсолютне правило: зміну інтерфейсу сервісу однаково доводиться узгоджувати з тими, хто його викликає.

Другий — система може йти за організацією, а не боротися з нею. 1968 року Мелвін Конвей (Melvin Conway) писав у журналі Datamation, що організації, які проєктують системи, змушені створювати проєкти, що копіюють їхні власні комунікаційні структури. Льюїс і Фаулер застосовують цю думку до звичного поділу великого застосунку за технічними шарами: одна команда відповідає за UI, друга за серверну логіку, третя за базу даних. Тоді навіть проста зміна перетворюється на міжкомандний проєкт, а логіка опиняється там, куди певна команда має доступ. Мікросервісний стиль перевертає цю картину. Кожен сервіс покриває одну бізнес-можливість, як-от замовлення, запаси чи платежі, від інтерфейсу до даних, і належить кросфункціональній команді з усіма потрібними для цього навичками. Ця команда має володіти сервісом протягом усього його життя, а не передавати його далі наприкінці проєкту. В Amazon це правило звучить так: хто збудував сервіс, той і відповідає за нього в продакшені.

Третій — логіка лишається в сервісах. Льюїс і Фаулер підсумовують цей вибір формулою smart endpoints and dumb pipes: розумні кінцеві точки й прості канали між ними. Сервіси спілкуються HTTP-запитами до API ресурсів або через легку шину повідомлень, яка лише маршрутизує повідомлення, а рішення ухвалюють сервіси на обох кінцях. Це свідомий відхід від корпоративної сервісної шини (enterprise service bus), яка маршрутизує, перетворює повідомлення й застосовує бізнес-правила посередині мережі, а не на її краях.

Дані ділять за тими самими лініями. Кожен сервіс сам керує своєю базою даних, і різні сервіси можуть брати різні системи керування базами даних, якщо це пасує їхній роботі. Обґрунтування дає предметно-орієнтоване проєктування Еріка Еванса (Eric Evans): велика система містить кілька моделей, і кожній потрібен явно окреслений обмежений контекст (bounded context), заданий організацією команд, частинами застосунку, якими ця модель керує, і фізичними речами на кшталт кодових баз і схем баз даних. Тоді погляд відділу продажів на клієнта й погляд служби підтримки на того самого клієнта можуть різнитися, і нікому не доводиться зводити їх в одній таблиці.

Витрати — частина самого визначення, і Льюїс із Фаулером називають їх прямо. Віддалений виклик коштує набагато дорожче за виклик у пам’яті, тож інтерфейси між сервісами доводиться робити крупнозернистішими. Переносити код через межу сервісу важко: зміни інтерфейсу треба узгоджувати, додавати шари зворотної сумісності, а тестування ускладнюється. Розподілені транзакції важко реалізувати, тому сервіси

координуються без них, погоджуються на те, що узгодженості можна досягти лише з часом, і виправляють помилки компенсувальними операціями. Будь-який виклик може не вдатися. А якщо сервіси не складаються в чисту композицію, складність нікуди не зникає, вона лише переїжджає у зв'язки між ними, де її важче помітити. Льюїс і Фаулер наводять аргумент, який вважають слушним: спершу збудувати моноліт, дати йому чисті модулі й розбити його лише тоді, коли він почне завдавати клопоту. Вони додають застереження: інтерфейс, який добре працює всередині одного процесу, рідко стає добрим інтерфейсом між сервісами.

Як це виглядає на практиці

Нотатка Продати останній квиток рівно один раз береться за окремий сервіс лише тоді, коли однієї бази справді замало, і називає правило, на якому тримається вся конструкція: у кожного товару є рівно один власник його запасу, сервіс запасів, і більше ніхто про запас не вирішує. Цей сервіс шардують за орендарем і товаром, а найбільші орендарі отримують власні комірки. Синхронно відбувається лише резервування. Оплату, замовлення й квитанцію ведуть окремі системи, і спільної транзакції між ними немає. Двофазний коміт нотатка відкидає: він блокує роботу, доки повільний хоч один учасник, а платіжні сервіси в ньому взагалі не беруть участі. Натомість вона обирає сагу — ланцюжок локальних кроків, у кожного з яких є свій крок відкату.

Кожен крок записує своє повідомлення в таблицю outbox у тій самій транзакції, що й зміну, про яку це повідомлення сповіщає. У PostgreSQL інструкція, що змінює дані, всередині `WITH` може передати свій результат вставці, тож сервіс запасів резервує одиницю товару й записує повідомлення однією інструкцією:

```
-- сервіс запасів, зі своєю власною базою даних
WITH held AS (
    UPDATE inventory
    SET available = available - 1
    WHERE tenant_id = 't-7' AND item_id = 'item-881'
      AND available >= 1
    RETURNING item_id
)
INSERT INTO outbox (order_id, item_id, event, units)
SELECT 'A-17', item_id, 'StockReserved', 1 FROM held;
-- розпродано: жоден рядок не підходить, тож повідомлення не записується
```

Далі change data capture публікує рядки з outbox, а кожен споживач ідемпотентний і використовує замовлення як ключ, бо повідомлення доходять щонайменше один раз. У цьому ескізі жоден інший сервіс не торкається таблиці `inventory` : інші дізнаються про запас із цих повідомлень або питають сервіс запасів.

Та сама нотатка показує, яке місце поділ на сервіси посідає в порядку робіт. У її переліку того, з чого почати, окремий сервіс і окремий власник для запасу стоять четвертими: після вимірювання, після того як запис стає однією умовною інструкцією, і після скидання навантаження через залу очікування, шлюз і закешовану відповідь «розпродано». Дешевші виправлення йдуть першими.

Кейс із портфоліо Морська платформа з чітко розділеними шарами фіксує архітектурний внесок засновниці компанії в межах роботи в NextMariner і показує інший бік вибору. Продукт був на ранньому етапі розвитку, його вимоги мали ще суттєво змінюватися, а мета полягала в тому, щоб зміна бізнес-правила не вимагала одночасних правок у фронтенді, API та базі даних. Рішенням стали три шари, кожен з одним завданням: фронтенд на Next.js для відображення, навмисно тонкий Go API, який маршрутизує, перевіряє запити, фільтрує їх з міркувань безпеки й оркеструє, та вся бізнес-логіка у функціях PostgreSQL. Межі записали й забезпечили їх дотримання під час рев'ю. Lewis і Fowler зазначають, що модульні межі всередині одного застосунку вимагають великої дисципліни, а явніше розділення на сервіси полегшує підтримання чітких меж між командами. Ця платформа зберегла цю дисципліну, не розділяючи свою бізнес-логіку на окремі сервіси.

Де виникають проблеми

Ланцюжки синхронних викликів множать простій. Якщо, щоб відповісти на один запит, перший сервіс викликає другий, а той третій, запит вдається лише тоді, коли працюють усі три, і чекає на кожного з них поспіль. Льюїс і Фаулер називають це мультиплікативним ефектом простою і згадують правило Guardian: один синхронний виклик на запит користувача. Тримайте синхронний шлях таким самим коротким, як у нотатці, де відповіді чекає лише резервування, а все після нього стає повідомленням. Кожному виклику, що лишився, дайте тайм-аут і запобіжник (circuit breaker) — патерни, на які Льюїс і Фаулер посилаються з книжки Release It!, — щоб сервіс, який відмовив, відрізали, а не чекали на нього.

Сервіси, які завжди змінюються разом. Поділ не за тими лініями дає сервіси, які неможливо випустити окремо: кожна нова можливість зачіпає три з них, і виходять вони лише разом. Льюїс і Фаулер пропонують простий тест: якщо два сервіси раз у раз доводиться змінювати разом, їх варто об'єднати. Схожа помилка — перетворювати виклики методів на віддалені виклики один до одного: сервіси тоді постійно перемовляються мережею й працюють погано. Проводьте межі навколо бізнес-можливостей і обмежених контекстів, робіть інтерфейси крупнозернистими і проєктуйте кожен сервіс так, щоб він витримував зміни в сервісах, які викликає. Тоді постачальник може розвиватися, не ламаючи своїх споживачів, а версіонування Льюїс і Фаулер радять лишати на крайній випадок.

Два сервіси ділять одну базу даних. Коли сервіс замовлень сам читає й записує таблиці запасів, схема стає інтерфейсом, яким ніхто не володіє: жодна команда не може змінити її сама, а запас отримує другого автора записів, якого перевірки сервісу запасів ніколи не бачать. Нехай кожна частина даних має одного власника, як запас у нотатці, а інші сервіси звертаються до цього власника або читають його повідомлення. Коли оновлення охоплює кілька сервісів, беріть сагу з компенсувальними кроками, а не розподілену транзакцію.

Ніхто не може простежити запит. Запит, що проходить крізь кілька сервісів, лишає уривок своєї історії в логах кожного з них. Льюїс і Фаулер відводять моніторингу в реальному часі центральне місце: технічні показники, як-от кількість запитів до бази даних за секунду, відстежують пліч-о-пліч із бізнесовими, як-от кількістю замовлень за хвилину. Вони також попереджають, що сервіси, які реагують на події один одного, породжують емерджентну поведінку, яку моніторинг має швидко помічати. Трасуйте кожен запит. У моделі OpenTelemetry трейс — це шлях одного запиту крізь систему, складений зі спанів, які мають спільний ідентифікатор трейсу й посилаються на батьківський спан, а поширення контексту дає змогу з'єднати спани з різних сервісів в один трейс. Передавайте контекст трейсу з кожним викликом і кожним повідомленням, зокрема з тими, що йдуть через outbox, інакше трейс обірветься на першому асинхронному кроці.

Що почитати далі

- Microservices — Джеймс Льюїс і Мартін Фаулер, 2014: дев'ять характеристик, історія назви, врізки про розмір і синхронні виклики та аргументи на користь обережності.

- How Do Committees Invent? — стаття Мелвіна Конвея 1968 року з Datamation про те, що структура системи копіює комунікаційну структуру організації, яка її спроєктувала.
- Domain-Driven Design Reference — стислий виклад патернів від Еріка Еванса, зокрема обмеженого контексту й карт зв'язків між контекстами.
- OpenTelemetry: Traces — спани, ідентифікатори трейсів, батьківські спани й поширення контексту: усе, завдяки чому один запит видно в кількох сервісах.
- PostgreSQL: WITH Queries — інструкції, що змінюють дані, всередині `WITH`, завдяки яким одна інструкція змінює рядок і записує цю зміну деінде.

Словник

Обирайте за фактичними умовами

Опишіть робочий процес, наявні інтерфейси та користувачів, яким потрібна зміна. Ми можемо визначити перевірюваний результат і спосіб передачі, який дасть вашій команді змогу підтримувати його.

Дивитися, скільки це коштує Зв'язатися з нами →

Зробіть термінологію практичною

Запитайте, як «Що таке мікросервіси?» стосується вашої організації. Конкретного прикладу з поточної системи достатньо для початку розмови.

Переглянути наші послуги

welcome@engineer.company +45 50 10 61 84 Копенгаген, Данія

Візьміть цю сторінку із собою

Візитівка

Дбайливо зроблено з любов'ю — D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Авторське право © 2025 – дотепер · Engineer ApS · Усі права захищено.

Надруковано з https://engineer.company/uk/glossary/microservices/