УДК 004.42

О. Ролік, О. Амонс, К. Ульяницька,
В. Хмелюк, С. Цимбал

МОДИФІКОВАНА МІКРОСЕРВІСНА АРХІТЕКТУРА НА ОСНОВІ EVENT SOURCING

Анотація: Стаття присвячена проєктуванню мікросервісної архітектури сучасних інформаційних систем. Проведено аналіз сучасних підходів до побудови мікросервісної архітектури, включаючи шаблони відомої Event-driven архітектури. Запропоновано модифіковану мікросервісну архітектуру на основі шаблону Event sourcing, що входить до складу Event-driven архітектури. В роботі наведена модифікована мікросервісна архітектура з використанням Event Sourcing шаблону та зазначені переваги застосування такого підходу.

Ключові слова: мікросервісна архітектура, мікросервіси, інформаційні технології, інформаційні системи, event sourcing.

Вступ

В наш час найбільш популярною архітектурою інформаційних систем є мікросервісна архітектура і все більше систем переводять з монолітної архітектури на мікросервісну. Така архітектура має багато своїх переваг, таких як покращені можливості до масштабування, використання різних технологій в різних мікросервісах, повторне використання мікросервісів в інших системах та інше. Також слід зазначити, що мікросервісна архітектура дуже добре розгортається на розподілених серверах та в хмарних середовищах. Але одночасно з цим є й свої недоліки у такому підході. Основні з них – зростання обсягу повідомлень між мікросервісами та організація обміну даними між ними. Метою статті є вдосконалення мікросервісної архітектури інформаційних систем, побудованої з використанням Event Sourcing.

Постановка проблеми

Кожен мікросервіс має свою базу даних (БД), з якою він працює. Якщо йому потрібні дані, яких немає в БД, до якої він має доступ, то він звертається до іншого мікросервіса для отримання таких даних. В результаті розбиття монолітної архітектури на мікросервіси, одна або кілька баз даних розбивається на багато невеликих баз даних для різних мікросервісів. При цьому на рівень обміну повідомленнями між сервісами виноситься обмін даними, який раніше виконувався безпосереднім збереженням даних в базу одним процесом і читанням цих даних іншим. Крім того, між сервісами залишається високий рівень зв’язності, що вносить додаткові проблеми для виводу

 ISSN 1560-8956

з обслуговування застарілих версій мікросервісів. Це збільшує обсяг робіт, які виконують програмісти, тестувальники, адміністратори та інженери, що забезпечують автоматизацію життєвого циклу програмного забезпечення (DevOps).

Існуючі підходи до розробки інформаційних систем на основі мікросервісів

В монолітній архітектурі масштабування на рівні баз даних відбувається за рахунок реплікацій, шардінгу [1], горизонтального або функціонального партиціювання [1, 2]. В мікросервісній архітектурі застосовуюся схожі підходи, але на рівні одного мікросервісу. У популярному архітектурному підході CQRS (Command Query Responsibility Segregation) [3] запити на зміну та отримання даних в мікросервісі розділяються між різними підсистемами, які працюють з різними БД. При запиті на зміну дані вносяться або модифікуються в реляційній БД, а ці зміни потім проєктуються на БД для читання, яка може бути як реляційною, так і не реляційною та є оптимізованою для швидкого читання. Окрім того, часто БД для читання дублюється для масштабування. Таким чином додатково балансується навантаження в системі: запитів на запис, як правило, кардинально менше ніж запитів на читання даних. Шаблон CQRS дуже добре працює як для монолітної, так і для мікросервісної архітектури.

Ще одним підходом в сучасній архітектурі розподілених інформаційних систем, який використовується дуже широко, є event-driven architecture [4].

При такому підході основною частиною для організації потоків роботи стає система обміну повідомленнями, наприклад, RabbitMQ [5], Apache Kafka [6] або Amazon SQS [7]. При цьому використовуються різні Enterprise Integration Patterns [8].

Event sourcing [4,8,9] є одним із таких шаблонів event-driven архітектури та часто використовується в мікросервісній архітектурі в комбінації з CQRS. В цьому випадку база даних на запис заміняється на Event Store – базою, яка зберігає послідовність всіх подій, викликаних командами на модифікацію даних. На рис. 1 представлено таку архітектуру.

Проведемо аналіз послідовності операцій при зміні даних користувачем на формі. З користувацького інтерфейсу на підсистему запису в мікросервіс приходить запит на модифікацію даних, який в термінах CQRS називається «командою». При відпрацюванні цього виклику підсистема запису генерує подію на модифікацію даних, яка реєструється в Event Store. Event Store зберігає подію та розповсюджує зміни на всі бази даних для читання. Додатково Event Store передає цю подію на всіх підписників. Надалі, коли з користувацького інтерфейсу прийдуть запити на отримання даних, підсистема читання в мікросервісі генерує запит на вибірку з бази даних читання (Read DB) та повертає ці дані на користувацький інтерфейс.

Рисунок 1. Архітектура мікросервісу
з використанням CQRS разом з Event sourcing

При цьому слід зауважити, що Event Store зберігає всі події в їх послідовності і це дозволяє в подальшому повністю «з нуля» відновити бази читання на основі подій.

Основними перевагами використання Event Sourcing разом з CQRS є:

– швидкі операції читання;

– можливість значного масштабування системи;

– зберігання історії подій, що значно спрощує подальший пошук помилок;

– розділення команд на зміну даних та запитів на читання по різним потокам та модулям;

– можливість підписання всіх зацікавлених в цих подіях підсистем для подальшого реагування.

Також слід зазначити такі недоліки:

– складніша реалізація, ніж при використанні однієї БД, що для простих сервісів буде зайвою;

– складність архітектури вимагає більш кваліфікованих розробників для реалізації базової версії такого мікросервісу;

– в процесі функціонування такий сервіс вимагатиме додаткових ресурсів.

Класичну мікросервісну архітектуру представлено на рис. 2, відповідно до якого клієнтська частина системи взаємодіє з мікросервісами викликами API, але робить це через API Gateway. Мікросервіси, в свою чергу, можуть викликати API

методи інших сервісів для отримання даних або звертатися з запитами на виконання певної роботи.

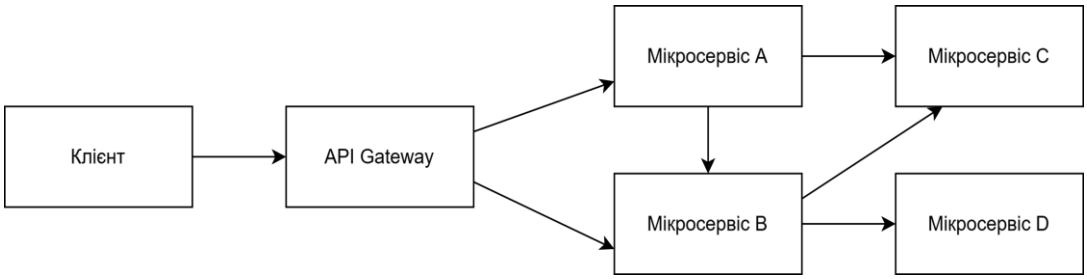

Рисунок 2. Класична мікросервісна архітектура

Використання API Gateway між клієнтом і мікросервісами крім масштабування вирішує таки задачі:

– розриває зв’язок між клієнтом та сервісами;

– дозволяє інтегрувати логування запитів від клієнтів, аутентифікацію та авторизацію;

– визначає, яка версія сервісу потрібна клієнту, і перенаправляє виклики на відповідний сервіс.

Класична мікросервісна архітектура дозволяє спростити розроблення складних систем, але при цьому вона має ряд недоліків, серед яких основними є:

– з розвитком системи зростає кількість запитів між мікросервісами;

– з розвитком системи зростають обсяги додаткових даних, які при відпрацюванні запиту потрібні сервісу від інших сервісів;

– потрібно знати, до якого мікросервісу і якої версії реалізації потрібно звернутися;

– з розвитком системи кількість версій одного мікросервісу, які потрібно підтримувати, збільшується через складність виведення старих версій з експлуатації.

Є два підходи до вирішення цих проблем (повністю або частково) – це використання або шини даних, або шини повідомлень.

При використанні шини даних створюється спеціальна підсистема для викликів API, як це використовується в SOA [10].

В іншому варіанті використовується шина повідомлень, наприклад, Service BUS, Messaging Queue або Eventbus [6, 11]. Такий підхід набуває все більшої популярності.

На рис. 3 наведено приклад реалізації event-driven архітектури з використанням Eventbus.

В архітектурі (див. рис. 3) мікросервіси генерують події (events) визначених типів та передають їх в Eventbus. Кожна подія має зареєстровані в Eventbus тип та

структуру даних, які передаються разом з подією. Інші сервіси реєструються на отримання подій визначеного типу. Коли генерується нова подія, то Eventbus зберігає її в локальній БД та передає далі на зареєстровані на цю подію мікросервіси.

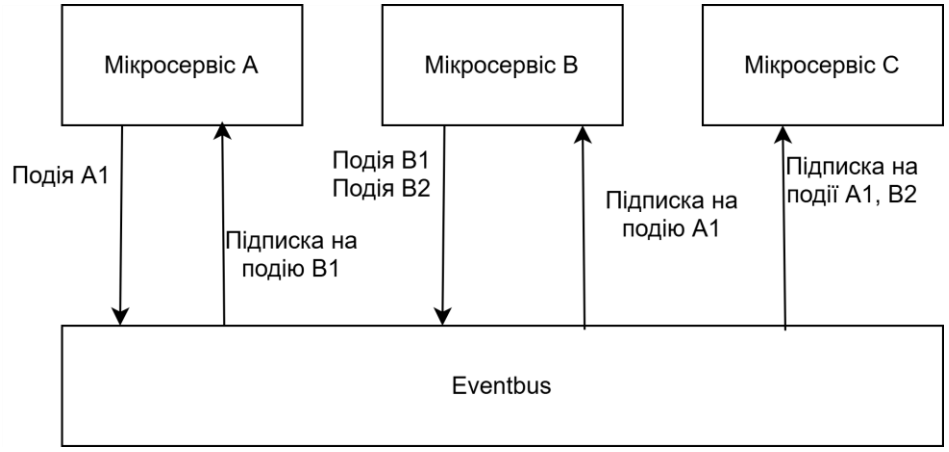

Рисунок 3. Мікросервісна архітектура на основі Eventbus

Як видно з прикладу на рис. 3, мікросервіс А зареєстрований як генератор подій А1, а мікросервіси В та С зареєстровані як отримувачі події А1. В свою чергу мікросервіс В зареєстрований як генератор подій В1 та В2, а також як отримувач події А1, і т.д. Коли мікросервіс А генерує подію А1, то Eventbus передає цю подію разом з даними, які до неї прикріплені, на сервіси В та С.

Такий підхід дозволяє зробити мікросервіси ще більш незалежними: мікросервіси А, В, С не знають нічого один про одного, знають тільки про зареєстровані типи подій та типи даних, які передаються разом з ними. Одну й ту саму подію може обробляти кілька різнотипних мікросервісів, або кілька екземплярів мікросервісів одного типу.

За рахунок такого розриву зв’язків між мікросервісами стає простіше виводити застарілі версії мікросервісів з використання, тому що мікросервіси тепер не знають один про одного і їх спілкування відбувається анонімно лише через події. Крім того, тепер мікросервісам не потрібно знати, до яких мікросервісів звертатися. Таким чином вирішується більша частина із зазначених вище проблем.

Окремо слід розглянути дані, які передаються разом з подією. Є кілька варіантів як це робити, але два з них будуть займати радикально крайні позиції за розміром такого повідомлення.

В першому, простому випадку, передаються лише базові дані для ідентифікації того, що потрібно зробити, наприклад, передача лише ідентифікаторів, та даних, які в БД ще немає. В такому випадку розмір повідомлень буде мінімальним, але мікросервісу прийдеться додатково звернутися до іншого сервісу за даними, які необхідні для виконання задачі.

 ISSN 1560-8956

В іншому, протилежному випадку, передається максимальна кількість даних для того, щоб сервіс, який буде реагувати на подію, отримував всі необхідні дані із повідомлення.

Слід зауважити, що залежності між сервісами залишаються в обох варіантах: в простому повідомленні мікросервіс має знати, до якого іншого мікросервісу звернутися за даними; в іншому випадку, коли сервіс відправляє максимальну кількість даних, сервіс, що відправляє повідомлення, повинен знати, які дані потрібні для подальшого реагування на подію, а в подальшому, коли потрібно буде добавити ще одну реакцію на цю подію, можливо, доведеться правити сервіс-відправник, щоб він передавав більше даних. Крім того, останній варіант збільшує навантаження на Eventbus за рахунок суттєвого збільшення розміру повідомлень, які потрібно зберігати.

Обидва варіанта мають свої недоліки, тому часто обирають проміжний варіант з балансом між кількістю даних, які потрібно передати, та необхідності сервісу в подальшому звертатися до інших сервісів.

Мікросервісна архітектура на основі Event Sourcing

Один із останніх сучасних підходів до побудови мікросервісної архітектури є Event Sourcing підхід, схожий на підхід, який застосовується також на рівні одного мікросервісу [12].

Загальна діаграма такої архітектури представлена на рис. 4, з якого видно, що кожен мікросервіс має ще додатково БД кешування даних з інших сервісів, які потрібні для виконання задач цього сервісу. В системі виділяють спеціалізовані типи подій, які вказують на зміну даних. Мікросервіси можуть підписуватися на такі події, отримувати повідомлення про зміну даних і потім зберігати ці дані в БД кешування. В подальшому, за необхідності, можна базу даних кешування очистити, перезачитати всі події на зміну даних з Event Store і потім послідовно виконати їх на новій БД для отримання останнього актуального стану цих даних. На рис. 4 показано, що для забезпечення цього Eventbus також відіграє роль Event Store і зберігає послідовно всі події про зміну даних.

Перевагою такого підходу є те, що сервіси через події оновлення можуть отримувати всі необхідні дані, які потім будуть використані при виконанні задач мікросервісу. За рахунок цього зв’язки між сервісами ще більше послаблюються.

Основним недоліком такої архітектури є необхідність збереження всіх подій, щоб в подальшому, коли це буде необхідно, мікросервіси могли отримати всі події визначеного типу та оновити собі БД кешування. Це має кілька наслідків: розмір бази даних для Event Store у великих системах буде великий; також через велику кількість подій модифікації даних процес оновлення БД кешування для сервісу триватиме довго і даватиме зайве навантаження на Event Store.

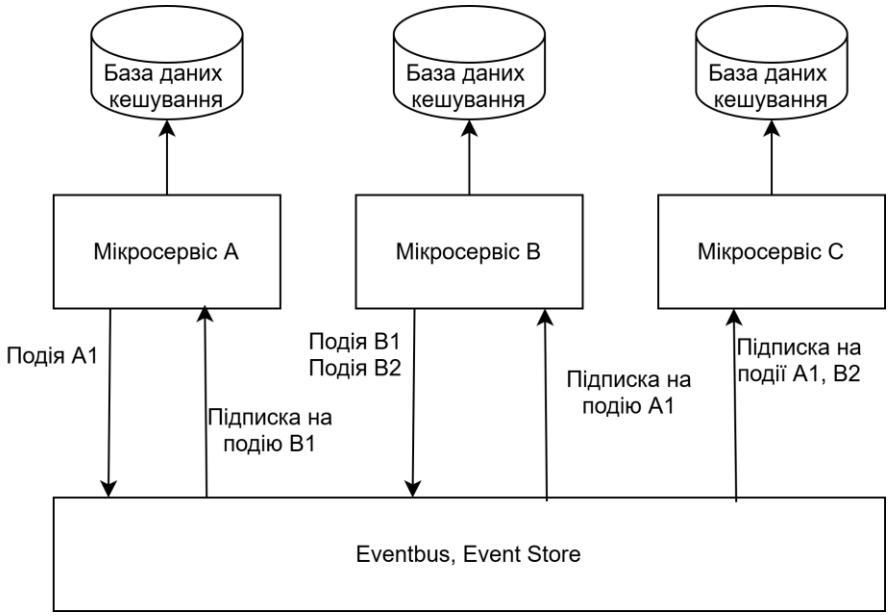

Рисунок 4. Event Sourcing в мікросерверній архітектурі

Модифікована мікросервісна архітектура на основі Event Sourcing

Пропонується модифікація мікросервісної архітектури, заснованої на Event Sourcing. Основною задачею модифікованої архітектури є необхідність зменшити навантаження на зберігання всіх подій для Event Store, але при цьому залишити можливість оновлювати всю базу даних кешу, щоб зберегти незалежність між мікросервісами.

Діаграма цієї архітектури представлена на рис. 5. Пропонується зробити такі зміни:

– кожен мікросервіс має реалізовувати стандартизований API, через який можна буде отримати всі дані в форматі, який використовується цим сервісом при генерації change event. В результаті такої уніфікації сервіс, якому потрібно отримати дані, вже знає про цей формат, а за рахунок стандартизації він не знатиме ніяких специфічних деталей реалізації мікросервіса, що віддає дані. Як наслідок, в систему не додаються сильні зв’язки між сервісами;

– додаємо в систему спеціалізований сервіс, який назвемо Data Source Locator. Всі мікросервіси будуть про нього знати. Основна задача Data Source Locator – реєстрація API методів для вивантаження поточних даних в БД кешування і адресу, за якою цей метод можна викликати;

– при додаванні нового типу події зміни даних потрібно також для неї зареєструвати в Data Source Locator API, яка зможе віддати останні актуальні дані в такому форматі.

Завдяки запропонованим змінам зменшується навантаження на Event Store, а також з’являється можливість зберігати події не за весь час, а лише за визначений

 ISSN 1560-8956

період, наприклад, за останні півроку. Також, при необхідності оновлення бази даних кешування, мікросервіс зможе легко знайти API, викликом якого можна отримати дані у відомому форматі, що використовується для даних, прикріплених до подій модифікації даних.

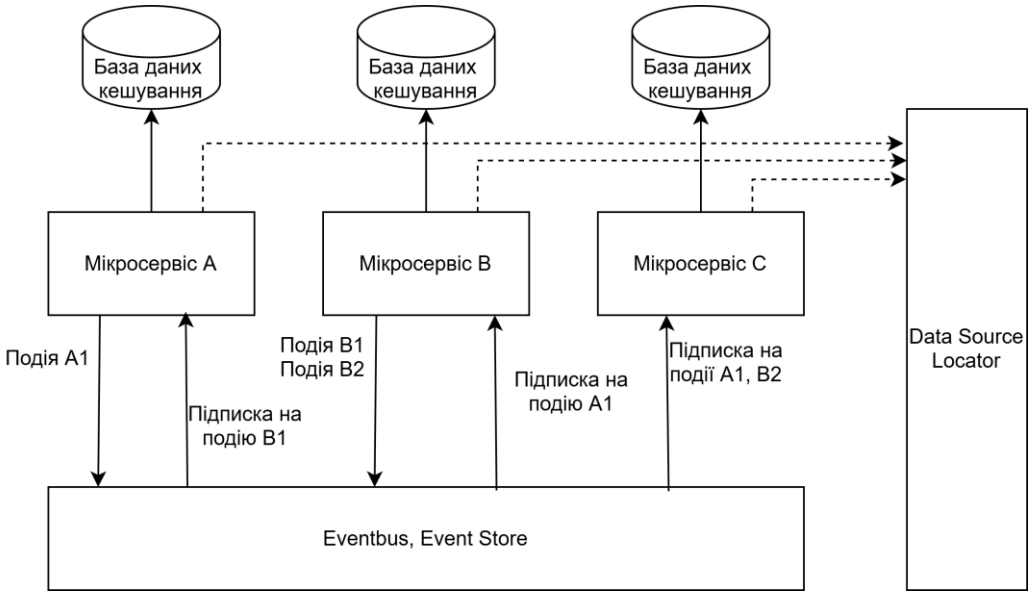

Рисунок 5. Модифікована мікросервісна архітектура з Event Sourcing

Висновок

В статті розглянуто основні виклики, що з’являються при розробці інформаційних систем на основі мікросервісної архітектури. Проаналізовано сучасні архітектурні шаблони створення мікросервісних систем, їх основні переваги та недоліки. Також запропоновано модифікацію до архітектури побудови інформаційних систем на основі мікросервісів з використанням Event Sourcing. Показано, що основними перевагами, які надає даний підхід, є відсутність необхідності зберігати всі події в Event Store весь час, а також зменшення навантаження на Event Store при повному оновлені бази даних кешування. В подальшому плануємо застосувати запропоновану архітектуру у реальній системі та провести дослідження щодо розподілу прав доступу до даних, що надходять з подіями.

Список використаних джерел

1. Martin Kleppmann. Designing Data-Intensive Applications / Martin Kleppmann, – O'Reilly Media, Inc., – 2017 – 614p.

2. Neal Ford, Mark Richards, Pramod Sadalage, Zhamak Dehghani. Software Architecture: The Hard Parts: Modern Trade-Off Analyses for Distributed Architectures / N. Ford, M. Richards, P. Sadalage, Z. Dehghani, – O'Reilly Media, Inc., – 2021 – 459p.

3. Martin Fowler. CQRS (Command Query Responsibility Segregation) // posted 14 July 2011. URL: https://martinfowler.com/bliki/CQRS.html

4. Martin Fowler. Event Sourcing. // posted 12 December 2005. URL: https://martinfowler.com/eaaDev/EventSourcing.html

5. RabbitMQ 4.1 Documentation. URL: https://www.rabbitmq.com/docs (дата звернення 20.05.2025)

6. Kafka 4.0 Documentation. URL: https://kafka.apache.org/documentation/ (дата звернення 20.05.2025)

7. Amazon Simple Queue Service Documentation. URL: https://docs.aws.amazon.com/sqs/ (дата звернення 20.05.2025)

8. Gregor Hohpe, Bobby Woolf. Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley Professional, – 2003, – 736 p.

9. Greg Young. Event Centric: Finding Simplicity in Complex Systems. // Addison-Wesley Educational Publishers Inc, – 2019. – 560 pp.

10. Schmidt Marc-Thomas, Hutchison B., Lambros P., Phippen Rob. The Enterprise Service Bus: Making service-oriented architecture real / Marc-Thomas Schmidt, B. Hutchison, P. Lambros, Rob Phippen // IBM Systems Journal. – 2005. – 44(4). – P. 781 – 797

11. Jaime Correia, Jorge Cardoso, Fillipe Araujo. Improving observability in Event Sourcing systems. //Journal of Systems and Software, – June 2021. DOI: 10.1016/j.jss.2021.111015

12. Nilesh G Charankar, Dileep Kumar Pandiya. Enhancing Efficiency and Scalability in Microservices Via Event Sourcing. // International Journal of Engineering Research & Technology (IJERT) – Vol. 13 Issue 4, April 2024. ISSN: 2278-0181