УДК 004.41 http://doi.org/10.15407/pp2024.01.64

А. М. Глибовець, Т. А. Чернова, М. М. Глибовець

РОЗРОБКА МЕТОДОЛОГІЇ ІМПЛЕМЕНТАЦІЇ ТРАНЗАКЦІЙ В РОЗПОДІЛЕНИХ СИСТЕМАХ З МІКРОСЕРВІСНОЮ АРХІТЕКТУРОЮ

У роботі описано аналіз проблематики використання мікросервісної архітектури в розподілених системах. Наголос зроблено на гнучкості у виборі технологій, масштабованості та організації команд, які працюють над заданими мікросервісами, технічних і доменних проблемах реалізації транзакцій у порівнянні з монолітною системою. Основну увагу приділено транзакціям, оскільки вони забезпечують дотримання атомарності, консистентності, ізольованості та стійкості над декількома сервісами.

У процесі аналізу сучасних підходів та рішень для роботи з транзакціями в розподілених системах було виявлено, що одним з ефективних рішень є використання патерну Transactional Outbox. Представлено його реалізацію у вигляді Spring starter. Останній додається до системи, конфігурується та полегшує використання транзакцій і публікацію подій, які є частинами транзакції у мікросервісній архітектурі.

Ключові слова: розподілена система, розподілені транзакції, мікросервісна архітектура, патерн Transactional Outbox, асинхронне спілкування, Kafka, Debezium.

Вступ

Наразі для розробки складних програмних систем дедалі частіше використовується мікросервісна архітектура (МА). Це рішення дозволяє розбивати систему на невеликі незалежні компоненти (застосунки) та отримувати швидку розгортку, масштабованість та гнучкість. Однак в таких розподілених системах (РС) виникають певні труднощі керування транзакціями, оскільки вони вимагають забезпечення атомарності, консистентності, ізольованості та стійкості над декількома сервісами [1-3].

Транзакції у базі даних – робоча операційна одиниця [2] для роботи із базою даних, певна послідовність змін, що виконуються в логічному порядку користувачем або програмою, яка працює із базою даних. Якщо відбувається будь-яка операція з інтерфейсом користувача, у базі даних виконується транзакція. Основні концепції транзакцій описуються абревіатурою ACID - Atomicity, Consistency, Isolation, Durability (Атомарність, Узгодженість, Ізольованість, Довговічність).

Атомарність гарантує, що будь-яка транзакція буде зафіксована тільки цілком і кожен крок транзакції виконається повністю. Якщо ж одна з операцій в послідовності завершиться із помилкою, то вся транзакція буде скасована. Існує поняття «відкату змін» (rollback), за якого всі зміни у зворотному порядку скасовуватимуться. У разі виникнення помилки під час проходження транзакції, користувач не побачить ніяких змін.

Узгодженість (консистентність) забезпечує те, що будь-яка завершена транзакція фіксує тільки допустимі результати. Під час виконання принципу узгодженості, база даних повинна завжди переходити із одного несуперечливого стану в інший несуперечливий стан. Умова узгодженості є необхідною для підтримки четвертої властивості транзакції - довговічності.

Ізольованість визначає те, що результат кожної транзакції не повинен залежати від виконання інших паралельних транзакцій. На практиці повна ізольованість важко досяжна. Тому вводиться поняття «рівні ізольованості». Послаблення ізольованості транзакцій призводить до відомих проблем: «брудне читання» (dirty read), «втрачене оновлення» (lost update), «брудний запис» (dirty write), «невідтворюване читання» (nonrepeatable read), «фантомне читання» (phantom read), «перекошене читання або запис» (read and write skew) [2]. У системах із МА, де застосовуються транзакції, розподілені між кількома сервісами, проблема ізольованості постає особливо гостро.

РС - це сукупність незалежних між собою компонентів, які взаємодіють та координують свої дії для досягнення спільної мети [1]. Ці компоненти можуть бути апаратними або програмними, можуть знаходитись на різних комп'ютерах та бути пов'язані спільною мережею. Компоненти взаємодіють між собою, передаючи повідомлення, використовуючи віддалені виклики процедур або інші форми комунікації. Цей принцип взаємодії дозволяє кожному компоненту працювати незалежно і спільно. Управління такою системою є складним процесом, оскільки вимагає вирішення таких проблем, як багатопоточність, консистентність даних, стійкість до помилок, збереження цілісності даних та масштабованості системи.

У МА застосунок – це набір невеликих, незалежних сервісів, кожен з яких має певну бізнес-функціональність. Ці сервіси можуть бути розроблені, розгорнуті та масштабовані незалежно один від одного, що дозволяє досягти більшої гнучкості та динамічності порівняно з традиційними монолітними архітектурними рішеннями [2].

Спілкування між сервісами в МА відбувається за використанням чітко визначених API, застосовуючи протоколи комунікації, такі як REST, черги повідомлень. Це дозволяє системі легше розвиватися та масштабуватися, оскільки зміни в одному сервісі не обов'язково потребують змін у всій системі. Однак архітектура мікросервісів також вносить власний набір викликів, включаючи управління консистентністю даних, балансування навантаження та моніторинг, трейсинг та логування [4]. Ці виклики потрібно ретельно досліджувати для забезпечення стійкості та масштабованості системи.

Тому виникає необхідність ефективних рішень для роботи з транзакціями в РС. Виникли спеціалізовані патерни та методології, які пропонують використання окремих підходів до реалізації такої роботи. Наприклад, патерн "Transactional Outbox (TO)" розроблений для забезпечення атомарності операцій та публікації подій як частини транзакційного процесу.

Метою цієї статті є опис аналізу існуючих загальних рішень імплементації транзакцій в РС мікросервісного типу та створеної методології реалізації розподілених транзакцій на базі черг повідомлень. Було виявлено, що одним з ефективних рішень є використання патерну ТО. Представлено розроблену нами його реалізацію у вигляді Spring starter. Останній додається до системи, конфігурується та полегшує використання транзакцій і публікацію подій, які є частинами транзакції в МА.

1. Проблеми в мікросервісній архітектурі

Під час проєктування комунікації між мікросервісами потрібно враховувати: налаштування правильної комунікації між сервісами, можливі затримки в мережі, проблеми з доступністю серверів та затримки між запитами [5]. У разі налаштування комунікації між мікросервісами важливою складовою є забезпечення відмовостійкості. Потрібно враховувати і консистентність даних. До прикладу, атомарне оновлення даних (або всі операції успішні, або жодна не успішна) є не простим викликом у побудові МА.

Управління консистентністю даних включає розв'язання двох основних проблем: роботу з багатопоточністю та відмовостійкістю. В багатопоточному середовищі, коли кілька процесів або потоків намагаються одночасно отримувати доступ та змінювати ті самі дані, може виникати неконсистентність даних, оскільки різні вузли можуть мати різні варіації тих самих даних. Відмова ж може відбуватися, коли вузол або мережеве з'єднання зазнають пошкоджень. Наприклад, сервіс в поточний момент часу не відповідає або повертає помилку, що призводить до часткових або неконсистентних оновлень даних.

Існує кілька підходів до управління консистентністю даних. Один із підходів - використання розподілених транзакцій (РТ), які забезпечують виконання групи операцій на кількох вузлах атомарно. РТ - це транзакції, що охоплюють кілька баз даних або систем, які можуть знаходитися на різних серверах або навіть у різних географічних місцях.

У РТ координатор транзакції відповідає за координацію дій усіх вузлів системи. Координатор спочатку запускає транзакцію, а потім надсилає запити до учасників систем для виконання потрібних операцій. Якщо всі вузли успішно виконують свої операції, координатор підтверджує транзакцію. Однак, якщо будь-яка з систем не виконує свою операцію, то координатор скасовує всю транзакцію, скасовуючи водночас усі зміни, зроблені учасниками систем.

Хоча розподілені транзакції можуть бути корисні для керування узгодженістю даних у МА, вони також можуть створювати власний набір проблем та труднощів, таких як додаткове навантаження на продуктивність та операційну складність.

2. Патерни, що полегшують роботу з розподіленими транзакціями

Розглянемо деякі відомі мікросервісні патерни, що вирішують проблеми, пов'язані з РТ та їх реалізації [6-7].

Шаблон двофазного коміту (2PC) є класичним розподіленим транзакційним шаблоном, який використовується для координації транзакцій між декількома системами. У цьому патерні існує координатор, що ініціює транзакцію, й інші учасники (мікросервіси, вузли системи), що отримують повідомлення від координатора та зберігають транзакцію [8]. Координатор визначає готовність кожного вузла зберегти транзакцію і, у випадку готовності усіх сервісів, відправляє друге повідомлення, аби вказати всім сервісам зберегти транзакцію. Якщо якийсь сервіс не готовий зберегти транзакцію, координатор відправляє повідомлення про відмову від транзакції. Цей патерн вирішує проблему консистентності даних в РС забезпечуючи те, що всі сервіси здійснюють або скасовують транзакцію у координований спосіб (за допомогою сервісу координатора). Він уможливлює забезпечення атомарності та консистентності між кількома системами, але може спричиняти проблеми, пов'язані з масштабованістю, доступністю та продуктивністю.

Серед відомих реалізацій цього патерну можна виділити наступні: Atomikos (система керування транзакціями, заснована на Java), Bitronix (менеджер транзакцій, заснований на Java), NServiceBus (платформа для передачі повідомлень та інтеграції на основі .NET) .

Шаблон Saga є альтернативою шаблону 2PC. Принцип роботи шаблону Saga полягає у тому, що після виконання локальної транзакції база даних оновлюється, а повідомлення про подію публікується для ініціювання наступної локальної транзакції до баз даних. У Saga використовуються такі терміни, як: compensating transaction (транзакція, що скасовує зміни, виконані попередньою транзакцією), countermeasure (дизайн техніка, що використовується для компенсації нестачі рівня ізоляції), compensatable transaction (транзакція, яку можна компенсувати), pivot transaction (транзакція, що є маркером для саги: якщо транзакція успішна, то сага продовжуватиме ряд дій для завершення послідовності транзакцій) та retriable transaction (повторна транзакція). Цей шаблон дозволяє забезпечити правильність виконання розподіленої між мікросервісами транзакції, а кожну локальну транзакцію можна компенсувати за допомогою її компенсуючої транзакції. Saga також гарантує, що всі операції будуть завершені успішно або відповідні компенсаційні транзакції будуть запущені для скасування раніше виконаної роботи. Є два види координації транзакцій у Saga: хореографія та оркестрація.

Заслуговують на увагу такі реалізації цього патерну: Uber Cadence (РС оркестрування робочих процесів, яка містить підтримку як саг на основі хореографії, так і саг на основі оркестрації), AxonIQ (відкрита платформа для створення мікросервісів, заснованих на подіях, яка включає підтримку саг), Lightbend Akka (набір інструментів та середовище виконання для побудови високопродуктивних, розподілених та відмовостійких систем, яка містить підтримку саг).

Окремо стоїть модель Eventual consistency. Це спосіб управління консистентністю даних в РС без використання традиційних транзакцій. У моделі дозволяється сервісам незалежно оновлювати свої локальні сховища даних, а послідовна консистентність досягається за допомогою

асинхронних оновлень та вирішення конфліктів. Хоча цей підхід може бути більш масштабованим та гнучким, ніж традиційні транзакції, він потребує обережного керування конфліктами даних та може призводити до тимчасових неузгодженостей.

Компенсаційний патерн - це спосіб скасувати наслідки невдалої транзакції без потреби повного відкату системи. У цьому патерні виконується компенсуюча транзакція для скасування наслідків початкової транзакції. Цей патерн часто використовується в поєднанні з патерном саги для керування локальними транзакціями в межах сервісу.

Патерн TO надає спосіб забезпечення консистентності даних у PC, коли сервіс повинен публікувати події як частину транзакції. Це легкий та ефективний спосіб керування координацією подій між декількома системами, не вводячи складності традиційних механізмів PT. Він дозволяє сервісу публікувати події у спосіб, який консистентний з локальною базою даних, не потребуючи механізмів 2PC або інших складних механізмів, як от saga, чи компенсаційний патерн. Це легкий і ефективний спосіб керування координацією подій між декількома системами.

Патерн часто реалізують на базі фреймворку Spring Cloud Stream. Він призначений для створення мікросервісних застосунків, які працюють на основі подій. Він дозволяє публікувати та отримувати події за допомогою різних систем повідомлень, таких як Apache Kafka, RabbitMQ та Google Pub/Sub. Його модель програмування для реалізації патерну TO базується на API Spring Transaction.

Говорячи про відомі реалізацій даних мікросервісних патернів варто зазначити, що їх можна реалізовувати різноманітними способами, підлаштовуючись під клієнтські запити та технічну базу застосунку. Наприклад, Eventuate.io є платформою для створення та управління мікросервісами, що працюють за принципом event-driven applications. Вона включає й підтримку шаблону TO, надає набір клієнтських бібліотек для Java та Spring, які спрощують інтеграцію шаблону TO в застосунки Spring Boot.

3. Transactional Outbox як асинхронний спосіб вирішення проблеми обміну даними в розподілених системах

Під час використання MA поширеним є послуговування своїми локальними сховищами даних - database per service.

Наприклад, сервіс, що відповідає за створення замовлень, зберігає замовлення в реляційній базі даних. Водночас сервіс може бажати надіслати подію про нове замовлення до Apache Kafka, щоб поширити цю інформацію на інші зацікавлені сервіси. Виконання цих двох дій може призвести до неузгодженості даних: нове замовлення буде збережене в локальній базі даних, але відповідного повідомлення до Kafka не буде надіслано (наприклад, через проблему з мережею). Або навпаки, ми можемо надіслати повідомлення до Kafka, але не зможемо зберегти замовлення в локальній базі даних. Обидві ситуації є небажаними та можуть призвести до того, що нібито успішно створене замовлення не буде відправлене. Або буде створений запит на відправку, але в самому сервісі замовлень не буде відомостей про відповідне замовлення.

Оптимальним рішенням цієї проблеми була б зміна лише одного ресурсу: або ж збереження даних на сервері, або ж відправка повідомлення і в подальшому оновлювати інший ресурс на основі успішності першого. Якщо розглянути спершу зміну ресурсу Apache Kafka, то послідовність подій виглядатиме наступним чином: сервіс, що відповідає за збереження замовлень, не буде зберігати замовлення в базі, натомість відправить подію в брокер повідомлень та підпишеться на цей же топік для отримання результату про успішну або неуспішну відправку. Отож, змінюється лише один ресурс за раз, і, якщо щось піде не так, ми одразу дізнаємося про це та повідомимо замовнику, що запит не вдалося виконати. Оскільки сервіс підписується на топік в Kafka, він буде отримувати сповіщення про доставку нового повідомлення в цей же топік і зможе зберегти нове замовлення на закупівлю у своїй базі даних.

Чим погане таке рішення? Якщо в системі виникатиме потреба додати функ-

ціонал на зчитування усіх замовлень в базі, то наш сервіс не зможе читати свої власні записи (read your own write semantic). Припустимо, що сервіс замовлень також має API для пошуку всіх замовлень певного клієнта. У разі виклику цього API безпосередньо після розміщення нового замовлення через асинхронну обробку повідомлень з топіка Kafka може статися так, що замовлення на закупівлю ще не було збережено в базі даних сервісу і, отже, його не буде повернуто цим топіком. Це може призвести до плутанини. Наприклад, користувачі можуть пропустити новостворені замовлення в своїй історії покупок та загалом втратити вкрай важливу інформацію про замовлення.

Існують способи впоратися з цією ситуацією. Так, сервіс може зберігати нові замовлення на закупівлю в оперативній пам'яті та відповідати на наступні запити на основі цих даних. Однак це швидко стає непростою задачею у разі реалізації складніших запитів або врахуванні того, що сервіс замовлень може складатися з кількох вузлів у кластерному середовищі, що потребуватиме передачі цих даних в межах кластера. Більш слушним рішенням буде зміна спершу стану бази, а потім відправка повідомлення, базуючись на зміні цього ресурсу. Дане рішення якраз таки можна імплементувати, використовуючи патерн ТО [9].

Патерн ТО вирішує завдання збереження консистентності та надійності даних у розподілених транзакційних середовищах шляхом збереження даних та подій, пов’язаних з ними в межах однієї бази даних, та гарантує публікацію збережених подій. Публіковані події відображають зміни стану системи, які потрібно повідомити іншим вузлам розподіленої системи. Додаючи події в таблицю "outbox", патерн забезпечує їх надійне зберігання в межах тієї самої атомарної транзакції разом із основною операцією бізнес логіки. Цей підхід надає гарантії консистентності даних, оскільки або всі зміни комітяться разом, або жодна з них не виконується (із скасуванням однієї скасовуються усі). Існує й окремий фоновий процес, відомий як "паблішер" подій або диспетчер подій. Він читає події з таблиці "outbox" та публікує їх у відповідний посередник повідомлень або систему обміну повідомленнями – це може бути Apache Kafka, RabbitMQ та інші. Шляхом розподілення процесу публікації подій від основної транзакції, шаблон ТО допомагає покращити продуктивність, масштабованість та стійкість системи.

Один із базових підходів реалізації патерну ТО є розробка та використання власних сервісів, що будуть зчитувати події з таблиці Outbox, менеджити їхній подальший життєвий цикл та керувати обробкою різноманітних помилкових сценаріїв. Основна перевага такого підходу є його гнучкість, оскільки він дозволяє налаштувати логіку обробки повідомлень відповідно до конкретних вимог.

У патерні кожен мікросервіс підтримує таблицю "outbox" у власній локальній базі даних. Ця таблиця є своєрідним буфером, де зберігаються події та повідомлення до їх поширення у зовнішні системи (наприклад, до публікування у меседж брокер).

Структура таблиці outbox table зазвичай залежить від конкретної реалізації та вимог МА. Однак є кілька загальних атрибутів, які часто присутні в таблиці вихідної скриньки, такі як message_id, message_payload, status, timestamp та інші. Конкретна структура та додаткові поля можуть варіюватися залежно від деталей реалізації та вимог архітектури мікросервісів.

У такій реалізації патерну існують спеціальні консьюмери, які відповідають за періодичні запити до таблиці "outbox" для отримання не переданих та не зчитаних раніше повідомлень. Цими консьюмерами можуть бути окремі мікросервіси або окремі модулі власного застосунку. Після отримання повідомлення з таблиці "outbox" спеціальний консьюмер обробляє його відповідно до визначеної бізнес-логіки. Цей процес може включати форматування повідомлення, виклик API або сервісів та обробку можливих помилок чи повторних спроб. У розробці такого консьюмера варто зважати на необхідність надання гарантованого читання, відправки та підтвердження оброблених повідомлень. Підтвердження читання, до прикладу, можна реалізовувати оновлюючи статус у таблиці "outbox" .

Підхід із використанням спеціальних консьюмерів надає деталізований контроль і налаштування споживання та обробки повідомлень. Він дозволяє адаптувати логіку обробки під задоволення конкретних вимог, таких як перетворення даних, перевірка безпеки або правил валідації.

Хоча даний варіант реалізації патерну сприяє гнучкості і можливості налаштування, він також має кілька потенційних недоліків, які варто врахувати. Серед вагомих недоліків можна вважати збільшений обсяг розробки та потреби підтримки в експлуатації. Необхідно проєктувати, реалізовувати, тестувати та підтримувати компоненти власних консьюмерів. А це збільшує загальний обсяг роботи з розробки та потребує більше зусиль для їхньої підтримки. Масштабування власних споживачів може також викликати певні складнощі, особливо у роботі з великою кількістю повідомлень та подій або одночасним доступом до таблиці outbox. Потрібно використовувати відповідні механізми балансування навантаження та дбати про ефективну та масштабовану обробку повідомлень.

4. Поняття WAL та CDC

Change Data Capture (CDC) та Write-Ahead Log (WAL) є пов'язаними концепціями в контексті систем баз даних, але вони служать різним цілям.

"Write-Ahead Log" (WAL) є технікою, яка часто використовується в системах баз даних для забезпечення стійкості даних та підтримки консистентності транзакцій. Вона передбачає запис змін, пов'язаних із транзакціями, у журнальний файл перед їхнім застосуванням до фактичних файлів даних бази даних. WAL виступає як послідовний журнал усіх змін, зроблених у базі даних, як успішно здійснених, так і незавершених транзакцій [10].

Коли транзакція змінює дані в базі даних, відповідні зміни спочатку записуються в WAL. Це гарантує безпечне зберігання журналу на надійному носії, перш ніж модифікації застосовуються до самої бази даних. Завдяки цьому підходу система бази даних гарантує, що в разі збою або аварії системи зміни, записані в WAL, можуть бути відтворені та застосовані для відновлення бази даних у консистентний стан.

WAL надає кілька переваг. Вона покращує продуктивність бази даних, дозволяючи буферизувати модифікації в пам'яті та записувати їх на диск у ефективнішому послідовному порядку. Вона також забезпечує цілісність даних, надаючи надійний запис транзакцій, що дозволяє відновлення після збоїв та скасування операцій. Крім того, WAL дозволяє реплікацію бази даних та реалізацію резервних серверів, реплікуючи журнал на інші системи для потреб синхронізації.

Загалом використання WAL підвищує стійкість, надійність та можливість відновлення систем баз даних, роблячи його фундаментальною технікою для забезпечення консистентності та доступності даних.

Change Data Capture (CDC) - це техніка, що використовується у галузі інтеграції та синхронізації даних для захоплення та передачі змін даних у реальному часі з баз даних до цільових систем [11]. CDC дозволяє постійно відстежувати та витягувати зміни даних, такі як вставки, оновлення та видалення, у міру їх виникнення в джерелі бази даних. Ці зловлені зміни даних потім перетворюються у формат, який може бути використаний іншими застосунками або реплікований до інших баз даних. CDC дозволяє ефективну та майже в реальному часі реплікацію даних, синхронізацію та інтеграцію між різноманітними системами, забезпечуючи консистентність та актуальність даних у екосистемі. Це широко використовується в сценаріях, де своєчасна та точна передача даних є критичною.

Зв'язок між CDC та WAL полягає в тому, що CDC часто використовує інформацію, записану в WAL, для захоплення та відстеження змін. Оскільки WAL містить послідовний журнал усіх модифікацій, зроблених у базі даних, його можна використовувати як надійне джерело для видобутку окремих змін та їх передачі іншим системам або споживачам.

Аналізуючи WAL, механізми CDC можуть ідентифікувати конкретні зміни, що сталися, включаючи редаговані рядки, стовпці та значення. Потім вони можуть зафіксувати ці зміни та перетворити їх у фор-

мат, придатний для передачі, як наприклад, створення подій зміни або оновлення реплік бази даних.

5. Debezium як спосіб імплементації транзакцій

Debezium - це розподілена платформа з відкритим кодом, яка використовує техніку Capture Data Capture (CDC) для отримання та передачі змін даних у реальному часі із журналів транзакцій бази даних (використовуючи WAL) [12]. Вона інтегрується з різноманітними популярними базами даних, такими як MySQL, PostgreSQL, MongoDB та інші за рахунок конекторів до баз даних. А також надає незамінні функції, такі, як здатність захоплювати зміни в усіх аспектах обробки даних включно із вставками, оновленням та видаленням. Під час налаштування для роботи з певною базою даних Debezium підключається до журналу транзакцій бази даних, який, як правило, реалізується за допомогою WAL. WAL реєструє всі модифікації, зроблені в базі даних, включаючи вставки, оновлення та видалення, в послідовному та стійкому журнальному файлі. Debezium зчитує зміни, записані в WAL, та перетворює їх у потік подій змін. Ці події змін потім перетворюються в стандартизований формат, такий як JSON або Avro, і стають доступними для споживання застосунками або системами вниз по ланцюжку.

Завдяки використанню підходу CDC на основі WAL, Debezium забезпечує надійний та мінімальний вплив захоплення даних, не навантажуючи додатково джерело бази даних [13]. Така ефективність доповнюється легко розширюваною архітектурою, яка може пристосовуватися до ситуацій стійкості до відмов, сприяючи простій синхронізації та інтеграції з численними платформами. Тому внесок Debezium у створення процесів стрімінгу в реальному часі, разом із наданням масштабованих реактивних потоків для сучасних розподілених фреймворків, неможливо недооцінити. Цей підхід мінімізує вплив на продуктивність джерела бази даних та забезпечує майже реальний час та точну реплікацію та інтеграцію даних між різними системами.

Debezium побудований на основі Apache Kafka. На відміну від підходу, заснованого на пулінгу даних, Debezium отримує події з дуже низьким навантаженням на систему та практично в режимі реального часу [12]. Debezium поставляється з CDC-конекторами для кількох баз даних, таких як MySQL, Postgres та SQL Server і після отримання даних може передавати їх іншим сервісам, зокрема, може публікувати подію в меседж брокер. Надійність та запобігання втрати даних є важливими характеристиками платформи.

Debezium надає пріоритет надійності, використовуючи журнали транзакцій, такі як Write-Ahead Log, що надаються системами баз даних, з якими він інтегрується. Ці журнали є надійним та міцним джерелом для отримання інформації про зміни даних. Debezium підключається до журналу транзакцій і зчитує записані в ньому зміни, гарантуючи, що жодні зміни не пропускаються або не втрачаються. Під час захоплення змін з журналу транзакцій Debezium запам’ятовує свою останню оброблену подію, щоб у разі перезапуску або відмови системи, Debezium міг продовжити читати події з того місця, де він зупинився, і продовжити читання змін без дублювання, тобто не читати повторно одні й ті ж дані. Цей механізм забезпечує цілісність даних та запобігає втраті даних у разі будь-яких збоїв або відмов.

Apache Kafka, брокер повідомлень, який використовується разом з Debezium, також відіграє важливу роль у забезпеченні надійності та стійкості даних. Він реплікує дані на кількох брокерах у кластері, забезпечуючи їхню обробку в разі відмов. Кожне повідомлення або подія, яка публікується в Kafka, зберігається у сховищі, яке називається "топіками". Топіки розбиваються на частини, і кожна частина реплікується на кількох брокерах. Цей механізм реплікації забезпечує ситуацію, коли навіть у разі відмови деяких брокерів або вузлів, дані все ще доступні і можуть споживатися користувачами.

Debezium використовує ці та інші можливості надійності Kafka, публікуючи отримані з таблиці outbox зміни подій в топіки Kafka. Після публікації Kafka відповідає за забезпечення їхньої міцності та дос-

тупності. Події можуть бути споживані консьюмерами або оброблені іншими застосунками.

6. Spring Boot Starter та його роль в реалізації транзакцій

Spring Boot Starter - є ключовою складовою частиною фреймворку Spring Boot, що надає спрощений спосіб налаштування та запуску застосунків через об'єднання залежностей та налаштувань для конкретних функціональностей або компонентів. Зазвичай Spring Boot Starter за замовчуванням включає підібраний набір бібліотек, класів автоконфігурації та налаштувань, необхідних для включення певної функції або інтеграції з певною технологією [14].

За допомогою Spring Boot Starter розробники можуть швидко додавати необхідні можливості та функціонал до своїх застосунків, не потребуючи ручного налаштування залежностей або написання стандартного коду. Ці стартери упаковують необхідні залежності та конфігурацію, що дозволяє розробникам зосередитися на написанні бізнес-логіки, а не гаяти час на складнощі налаштування та інтеграції різних компонентів.

Spring Boot Starter надає широкий спектр функціоналу включно із підключенням до баз даних, веб-розробкою, безпекою, обміном повідомленнями, кешуванням тощо. Кожен стартер дотримується певного підходу і надає послідовний спосіб інтеграції пов'язаних технологій у застосунки Spring Boot. На прикладі стартеру "spring-boot-starter-web" можна побачити залежності та конфігурації, необхідні для розробки веб-застосунків з використанням Spring MVC, який включає в себе цей стартер. При доданні spring-boot-starter-web у проєкт, автоматично імпортується кілька інших залежностей, таких як:

- Spring MVC: основний компонент фреймворку Spring для створення веб-застосунків, що надає архітектуру MVC та обробляє відображення маршрутів запитів, обробку запитів та генерацію відповідей.
- Embedded Servlet Container: вбудований контейнер сервлетів (наприклад, Tomcat, Jetty або Undertow), що дозволяє запускати веб-застосунки без потреби в зовнішній конфігурації сервера.
- Spring Web: модуль, що надає додаткові функції та утиліти для роботи з вебом, включно із обробкою HTTP-запитів та відповідей, обробкою форм, зв'язуванням даних, валідацією та обробкою помилок.
- Jackson JSON: бібліотека включена для підтримки серіалізації та десеріалізації даних у форматі JSON (JavaScript Object Notation). Вона дозволяє легко конвертувати об'єкти Java в JSON та навпаки.

Крім наданих стартерів, розробники також можуть створювати власні стартери для упакування уже використовуваних компонентів або бібліотек, специфічних для їхніх застосунків чи домену. Це сприяє модульності коду, повторному використанню та стандартизації в проєктах організації.

7. Методологія імплементації транзакцій

Проаналізувавши проблематику транзакцій в РС, ми розробили власну методологію імплементації транзакцій на базі патерну ТО та черг повідомлень, яка спрямована на їх вирішення з наголосом на забезпеченні послідовності, консистентності та надійності даних у множинних сервісах. На рисунку 1 наведена структура запропонованої методології на прикладі розробленої системи для предметної області – замовлень товарів.

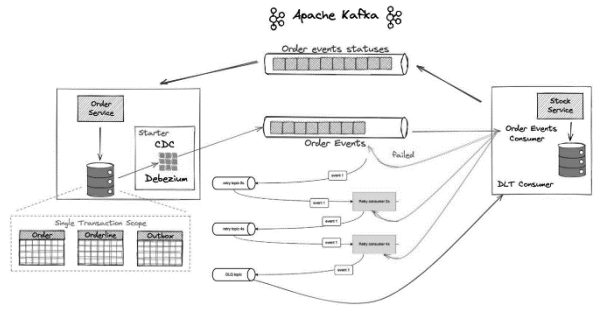

Рис. 1. Методологія імплементації транзакцій

Нехай система замовлень реалізована у вигляді двох мікросервісів: Order Service, що отримує замовлення товарів, та Stock Service, який на базі цих замовлень здійснює певні перевірки товару та інші ма-

ніпуляції для успішного завершення життєвого циклу замовлення. Обидва мікросервіси реалізовані на Java. Вони використовують Spring Boot Framework, JPA/Hibernate для доступу до своїх баз даних, які реалізовані на MySQL. Проте для нашої методології немає чіткої прив'язки до бази даних чи то фреймфорку, оскільки вона дозволяє реалізацію з використанням різноманітних технологій, зокрема, таких як: CDI, PostgreSQL, WildFly та інших.

Сервіс замовлень надає простий REST API для створення замовлень. Сервіс складу за допомогою Apache Kafka отримує події, експортовані сервісом замовлень та створює відповідні записи у власній базі даних MySQL.

Для експортування подій у Apache Kafka сервіс замовлень використовує розроблений Spring Boot Starter, що дбає про налаштування таблиці outbox. Запис подій до таблиці відбувається в межах тієї ж транзакції, що і збереження замовлення в базі даних. Стартер самостійно дбає про моніторинг та зчитування даних таблиці outbox, використовуючи WAL, CDC, Debezium, та публікацію зчитаних даних до Apache Kafka.

Після зчитування подій з Apache Kafka, сток сервіс сповіщає про вдале або провальне зчитування та обробку даних, використовуючи Apache Kafka. Ці події пізніше сервіс замовлень читає та змінює статус замовлення з pending на closed / canceled. Також цей сервіс дбає про відмовостійкість завдяки повторному читанню даних, гарантує обробку даних та відповідь у вигляді публікації статусу обробки у відповідний топік в Apache Kafka.

Дотримуючись цієї методології, можна забезпечити надійний обмін даними між мікросервісами, обробку операцій у межах транзакції та відповідних подій, відправлених іншим сервісам, атомарно та послідовно. Навіть у разі виникнення проблеми під час надсилання подій, їх можна спробувати повторно відправити, або відтворити з таблиці outbox, забезпечуючи потрібну послідовність в РС. Цим можна скористатися і для збереження цілісності та послідовності даних, одночасно дозволяючи асинхронну комунікацію між сервісами.

8. Практична реалізація

На основі запропонованої методології був розроблений Spring boot Starter, який полегшує конфігурацію роботи з транзакціями в РС, що публікують події як частину транзакції. Він має вигляд спеціальної бібліотеки для полегшення впровадження патерну Transactional Outbox в застосунок, побудований на основі Java. Основною метою стартера є спрощення процесу збереження та публікації подій у РС.

Стартер включає необхідні залежності та конфігурації, а також надає зручні методи для збереження подій у відведеній таблиці (outbox) і передачі їх у Debezium для подальшого розповсюдження. Використовуючи цей стартер, розробники можуть ефективно впроваджувати патерн TO, покладаючись на потужність Debezium для зчитування та розповсюдження подій у РС, а також на гарантовану доставку та цілісність даних за рахунок використання Apache Kafka як базової технологій для Debezium.

Інтеграція застосунку зі стартером розпочинається з його підключення у вигляді залежності (Рис. 2.):

```
<dependency>
    <groupId>com.chernova</groupId>
    <artifactId>transactional-outbox-starter</artifactId>
    <version>0.0.1-SNAPSHOT</version>
</dependency>
```

Рис. 2. Підключення стартеру до клієнтського застосунку

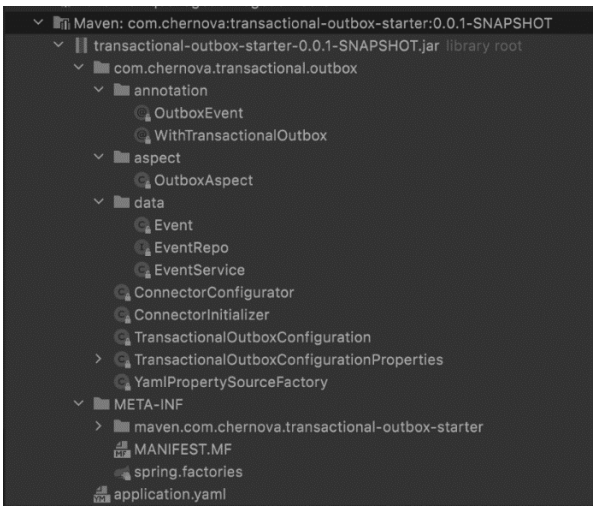

Рис. 3. – Підключений стартер

Рисунок 3 відображає доданий стартер до застосунку. Для коректної роботи стартеру та підтримки роботи з outbox table та Apache Kafka, необхідно передати наступні конфігурації, що будуть зчитані стартером під час ініціалізації проєкту. Connector.class - вказує клас конектора, який використовується для підключення до джерела даних (наприклад, значення конектор класу io.debezium.connector.mysql.MySqlConnector, означає підключення до MySQL бази даних). database.hostname - вказує ім'я хоста, на якому розташована база даних клієнта. atabase.port – визначає порт, на якому доступна база даних клієнта. database.user - уточнює ім'я користувача для підключення до бази даних клієнта. database.password - вказує пароль для підключення до бази даних клієнта. database.server.id - задає унікальний ідентифікатор сервера, який використовується для розпізнавання транзакцій. database.server.name - визначає унікальне ім'я сервера, яке застосовується для ідентифікації каналу змін даних у Debezium. database.history.kafka.bootstrap.servers - вказує адресу та порт сервера Apache Kafka, який використовується для зберігання історії змін бази даних (до прикладу “broker:9092”). database.history.kafka.topic - уточнює тему Kafka, де зберігається історія змін бази даних. key.converter.schema.registry.url: вказує адресу та порт сервера реєстрації схем Avro.

```yaml
transactionaloutbox:
  enabled: true
  connector:
    class: io.debezium.connector.mysql.MySqlConnector
  database:
    hostname: mysql
    port: 3306
    user: debezium
    password: dbz
    server:
      id: 42
      name: asgard
    history:
      kafka:
        topic: dbhistory.demo
        bootstrap:
          servers: broker:9092
    key:
      converter:
        schema:
          registry:
            url: http://schema-registry:8081
    value:
      converter:
        schema:
          registry:
            url: http://schema-registry:8081
```

Рис. 4. Приклад конфігурації стартера в yaml файлі

value.converter.schema.registry.url - визначає адресу та порт сервера реєстрації схем Avro.

Ці конфігурації мають бути надані у yaml файлі й використовуються для налаштування Debezium з метою підключення до бази даних MySQL, зчитування змін у реальному часі та перетворення їх на події, які можна розповсюджувати через Apache Kafka. Приклад конфігурації наведено на рисунку 4.

Для керування підключенням чи відключенням роботи стартера під час запуску застосунку необхідно також задати значення true/false для конфігурації transactionaloutbox.enabled.

Для збереження подій до outbox table, зчитування подій з outbox table та публікацію їх до Apache Kafka, необхідно додати до методу сервісу анотацію @WithTransactionalOutbox (рис.5).

```java
@Transactional
@WithTransactionalOutbox
public PurchaseOrder addOrder(@OutboxEvent PurchaseOrder order) {
    return purchaseOrderRepo.save(order);
}
```

Рис. 5. Використання анотації WithTransactionalOutbox доданого стартера

В цьому прикладі стартер в межах однієї транзакції зі збереженням замовлення до бази збереже подію до таблиці outbox. Для реалізації такого функціоналу стартер використовує можливості Spring, а саме Spring AOP (рис.6).

```java
@Around(value = "@annotation(withTransactionalOutboxAnnotation)")
public Object withTransactionalOutbox(ProceedingJoinPoint joinPoint, WithTransactionalOutbox withTransactionalOutboxAnnotation)
        throws Throwable {
    Object result = joinPoint.proceed();
    String jsonData = "{}";

    MethodSignature signature = (MethodSignature) joinPoint.getSignature();
    Annotation[][] annotations = signature.getMethod().getParameterAnnotations();
    Object[] args = joinPoint.getArgs();

    Object annotatedArgument;
    for (int i = 0; i < args.length; i++) {
        for (Annotation annotation : annotations[i]) {
            if (annotation instanceof OutboxEvent) {
                annotatedArgument = args[i];
                jsonData = objectMapper.writeValueAsString(annotatedArgument);
                break;
            }
        }
    }
    ...

    return result;
}
```

Рис. 6. Spring AOP для реалізації логіки стартера

Застосувавши усі необхідні налаштування стартера, можна інтегрувати засто-

сунок із патерном ТО і таким чином забезпечити атомарність операцій та консистентність даних між різними сервісами. Більше того, додаючи стартер до свого застосунку, можна отримати ряд можливостей, які надаються технологіями Debezium та Apache Kafka.

9. Рекомендації щодо налаштування системи на базі розробленої методології

Під час проектування та реалізації РС, які використовують методологію імплементації транзакцій, варто врахувати кілька рекомендацій.

Насамперед, важливо переконатися, що компоненти системи мають слабку зв'язність та спілкуються за допомогою асинхронного обміну повідомленнями для розділення транзакційної обробки від зовнішніх сервісів. Такий підхід спілкування сприяє стійкості та масштабованості [15-16]. Не менш важливим є використання надійного брокеру повідомлень для комунікації між мікросервісами та гарантованої надійної доставки повідомлень. У ролі такого брокеру можуть бути Kafka або RabbitMQ.

Однією з важливих компонент у розробці дистрибутивної системи є впровадження механізмів обробки помилок та повторного читання даних у разі помилок, управління відмовами та забезпечення консистентності даних. Із цією метою можна використовувати механізми Retryable Topic та Dead Letter Topic [17-18].

Також важливо належну увагу приділити надійності консьюмерів [19]. Оскільки консьюмери виконують обробку повідомлень із черги та виконують певні операції або змінюють стан системи на основі цих повідомлень, важливо забезпечити, щоб ці операції гарантували безпеку та надійність системи, приводили обробку даних до однакового результату незалежно від кількості повторних повідомлень. Це важливо для забезпечення консистентності даних та уникнення небажаних побічних ефектів. Наприклад, якщо консьюмер створює запис у базі даних на основі отриманого повідомлення, то необхідно гарантувати, щоб у базі даних було створено тільки один унікальний запис, навіть якщо повідомлення буде оброблено кілька разів. Також важливим у проєктуванні консьюмерів є використання унікальних ідентифікаторів для оброблених повідомлень або збереження стану процесу обробки, що забезпечить ідемподентну обробку.

10. Тестування

Основними аспектами тестування була перевірка вдалої інтеграції Debezium з існуючими компонентами та технологіями в системі. Це включило перевірку забезпечення правильної конфігурації та налаштування стартера, який базується на імплементації ТО патерну. Також було оцінено інтеграцію з брокером повідомлень Kafka та перевірено публікацію повідомлень до брокеру, наявність повторного читання даних за необхідності.

На нашому прототипі були здійснені перевірки поведінки системи у випадку наступних сценаріїв виникнення помилки:

- у разі спроби збереження замовлення до клієнтської бази та перевірки системи на скасування публікації повідомлення;
- у разі публікації повідомлення до брокера повідомлень та перевірки системи на скасування збереження даних про замовлення до клієнтської бази;
- у разі повідомлення консьюмером у сток сервісі та перевірки системи на повторне читання з використанням retryable механізму;
- після кількох невдаих спроб обробки повідомлень, коли консьюмер в сток сервісі не може обробити повідомлення та перевірити систему на відправку повідомлення до DLT топіку;
- після успішної обробки повідомлення сток сервіс відправив статус успішної обробки до відповідного топіку і сервіс замовлень, базуючись на даній відповіді, змінив статус замовлення на завершений;
- після неуспішної обробки повідомлення сток сервіс відправив статус проблеми обробки до відповідного топіку і сервіс замовлень, базуючись на даній відповіді, змінив статус замовлення на скасований.

Як результат оцінювання система коректно відпрацювала в обох успішних та помилкових сценаріях. Варто зазначити, що даний прототип має правильно налаштовані механізми повторної спроби читання та обробки даних, оскільки у випадку виникнення помилки, спроєктований механізм повторної спроби автоматично повторював операцію, для подолання тимчасових проблем, таких як проблеми з мережею або тимчасова недоступність ресурсів. Повторні спроби обробки виконувались із використанням стратегій публікацій подій до DLT топіку, щоб уникнути перевантаження системи та забезпечити затримку між послідовними спробами.

Висновки

У процесі аналізу сучасних підходів та рішень для роботи з транзакціями в РСах було виявлено, що одним із ефективних рішень є використання патерну Transactional outbox. Він дозволяє зберігати події, пов'язані з транзакціями, в окремій таблиці, що уможливлює їхню подальшу інтеграцію з іншими сервісами через застосунки або механізми, як-от, через брокер повідомлень Apache Kafka.

Проведений аналіз дозволив розробити методологію імплементації транзакцій в РС, а також створити стартер для полегшення такої роботи з розподіленими транзакціями та публікацією подій, що є частинами транзакції в чергу повідомлень. Цей стартер надає зручний і стандартизований спосіб інтеграції патерну ТО в Spring Boot застосунки, забезпечуючи автоматичне керування транзакціями та публікацію подій.

Оцінювання прототипу показало ефективність впровадження даного підходу в РС. Розроблена методологія та стартер можуть допомогти забезпечити надійну обробку розподілених транзакцій та публікацію подій.

Література

1. Maarten van Steen, Andrew S. Tanenbaum. "A brief introduction to distributed systems." Web. 2016. https://www.distributed-systems.net/my-data/papers/2016.computing.pdf
2. Fowler M. Microservices [Електронний ресурс] / M. Fowler, J. Lewis. – 2014. – Режим доступу до ресурсу: https://martinfowler.com/articles/microservices.html.
3. Яшина О.М., Кравчук О.А. Дослідження мікросервісної архітектури, архітектурний стиль REST та їх сучасна реалізація на Java. Вісник ХНУ: технічні науки, Номер: №5, 2020, с. 106-114.
4. Trzaska, Mariusz. “Technical challenges of creating an IT system in microservices architectural style using cloud services” Web. 2022 https://users.pja.edu.pl/~mtrzaska/Files/PraceMagisterskie/220217-Grabowski.pdf
5. Newman S. Monolith To Microservices / Sam Newman., 2019. – 301 с. – (2).
6. Fowler, Martin. “How to break a Monolith into Microservices” Web. 2018 https://martinfowler.com/articles/break-monolith-into-microservices.html
7. Richardson C. Microservices Patterns With examples in Java / Chris Richardson., 2018. – 520 с.
8. Fowler, Martin “Two Phase Commit” Web. https://martinfowler.com/articles/patterns-of-distributed-systems/two-phase-commit.html
9. Transactional Outbox Pattern https://www.linkedin.com/pulse/transactional-outbox-pattern-distributed-pratik-pandey (дата звернення 08.07.23)
10. Chola, Abhinal “Understanding Write Ahead Logging: 4 Comprehensive Aspects” Web. https://hevodata.com/learn/write-ahead-logging/
11. Spring Blog. “Case Study: Change Data Capture (CDC) Analysis with CDC Debezium source and Analytics sink in Real-Time” Web. https://spring.io/blog/2020/12/14/case-study-change-data-capture-cdc-analysis-with-cdc-debezium-source-and-analytics-sink-in-real-time
12. Debezium Documentation. Web. https://debezium.io/documentation/reference/stable/index.html
13. Hevodata. “3 Easy Steps to Decode CDC Using Debezium Spring Boot” Web. https://hevodata.com/learn/debezium-spring-boot/
14. Spring Boot Starters documentation. Web https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using.build-systems.starters

15. Rajabi M. How to Avoid Coupling in Microservices Design [Електронний ресурс] / Mariam Rajabi. – 2020. – URL: https://www.capitalone.com/tech/softwareengineering/how-to-avoid-loose-coupled-microservices/. (дата звернення 15.09.21)
16. Walpita P. Coupling and Cohesion in Microservices [Електронний ресурс] / Priyal Walpita. – 2020. – URL: https://priyalwalpita.medium.com/coupling-and-cohesion-inmicroservices-235ed9203843. (дата звернення 18.09.21)
17. Baeldung Blog "Implementing Retry in Kafka Consumer" Web. https://www.baeldung.com/spring-retry-kafka-consumer
18. Medium. "Dead Letter Queue (DLQ) in Kafka" Web. https://towardsdatascience.com/dead-letter-queue-dlq-in-kafka-29418e0ec6cf
19. Richardson, Chris. "Pattern: Idempotent Consumer" Web. https://microservices.io/patterns/communication-style/idempotent-consumer.html

Одержано: 01.02.2024

Про авторів:

Глибовець Андрій Миколайович,
доктор технічних наук, професор,
декан факультету інформатики
Національного університету
«Києво-Могилянська академія».
Кількість наукових публікацій
в українських виданнях – понад 40.
Кількість наукових публікацій
в зарубіжних виданнях – 8.
Індекс Хірша – 3.
https://orcid.org/0000-0003-4282-481X,

Глибовець Микола Миколайович,
доктор фізико-математичних наук,
професор факультету інформатики
Національного університету
«Києво-Могилянська академія».
Кількість наукових публікацій
в українських виданнях – понад 200.
Кількість наукових публікацій
в зарубіжних виданнях – більше 10.
Індекс Хірша – 12, https://orcid.org/0000-0002-3853-2171

Чернова Тетяна Андріївна,
магістр факультету інформатики
Національного університету
«Києво-Могилянська академія»

Місце роботи авторів:

Національний університет
«Києво-Могилянська академія»,
04070, м. Київ, вул. Сковороди 2.
E-mail: a.glybovets@ukma.edu.ua,
glib@ukma.edu.ua, t.chernova@ukma.edu.ua