Лабораторна робота №3. Забезпечення транзакційності у взаємодії інтелектуальних сервісів

Тема: Розподілені транзакції через шаблон SAGA
Задача: на основі прикладу, описаного у Лабораторній роботі №2 (6), реалізувати шаблони проєктування transactional outbox та saga для надійних розподілених транзакцій.

Завдання

1. Ознайомитися з патерном transactional outbox.
2. Реалізувати цей патерн для запису повідомлень в чергу.
3. Для розподіленої транзакції (event sourcing з п. 6 лр. 2.) реалізувати SAGA. Запис в сервіс-консьюмер повинен кидати помилку. Сервіс продьюсер має відкатити зміни по таймауту або повідомленню в DLQ.

Контрольні питання

1. Що таке CQRS і які задачі вирішує даний шаблон
2. Оркестровка і хореографія
3. Опишіть як працюють брокери повідомлень, що таке visibility timeout
4. Які бувають гарантії доставки повідомлень
5. Optimistic vs pessimistic locking
6. Transactional outbox - що це і навіщо
7. SAGA

Патерн проектування “Transactional Outbox & Inbox”

З існуючих методів та алгоритмів для забезпечення транзакційності в розподілених системах, найкраще себе показують ті методи, що не стараються побудувати ефективні розподілені транзакції, а ті, що алгоритмічним шляхом обробляють локальні транзакції мікросервісів і синхронізують дані між ними. Але в такому випадку, коли один мікросервіс зберігає результати виконання запиту в базу даних, та після транзакції викликає локальну транзакцію іншого сервісу, в процесі виклику наступного сервісу послідовності може статись збій, і повідомлення не будуть надіслані. А в цей момент транзакцію, що завершилась з успіхом, потрібно відкотити, що буде неможливо. Більш того, навіть якщо

викликати їй компенсаційну транзакцію, може статись збій в роботі бази даних, і компенсаційна транзакція не буде виконана. Або ж у випадку протилежної ситуації: коли перед збереженням в базу даних результатів, було викликано логіку наступного сервісу, однак на моменті збереження даних і коміту транзакції відбувається помилка [24]. Патерн проектування Transactional Outbox - це патерн, націлений на вирішення проблеми одночасного і якісного збереження даних в базу даних на рівні мікросервіса, та публікацією повідомлення на те, щоб продовжити послідовність з локальних транзакцій [25].

Класичним підходом до реалізації цього патерну, є використання додаткової таблиці в базі даних. В цю таблицю буде збережено повідомлення, яке необхідно надіслати до наступного мікросервісу, ідентифікатор повідомлення, та додаткові для обробки дані, що буде отримано системою, що буде надсилати повідомлення наступному сервісу з цієї таблиці, зазвичай брокером повідомлень [26]. Збереження цього повідомлення в таблицю відбувається в рамках однієї і тієї ж транзакції, що і з даними, що треба зберегти в результаті виконання логіки транзакції. Тому, у випадку, коли відбудеться помилка на етапі завершення транзакції, тобто процесу коміту, ні результат роботи транзакції, ні повідомлення до наступного мікросервісу не буде записано в базу даних. Приклад такого процесу зображено на рис. 2.9:

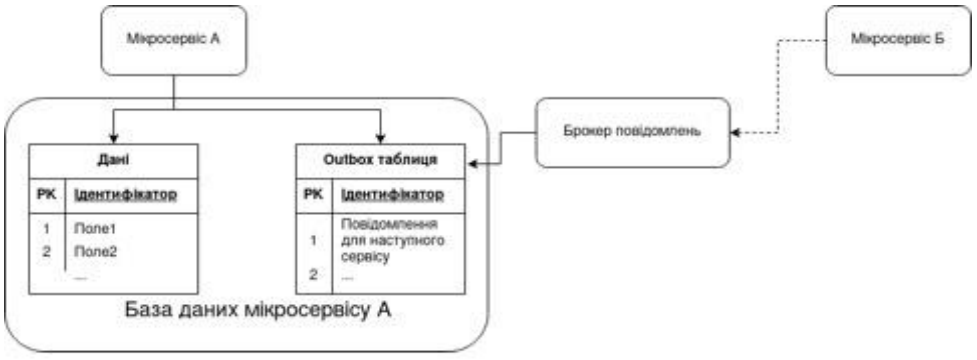

Рисунок 1 – Принцип роботи Transactional Outbox

Коли транзакція завершується успішно, в роботу вступає брокер повідомлень, який буде зчитувати таблицю для повідомлень, та використовувати ці дані для пересилань повідомлень наступним мікросервісам. Для того, щоб

отримати дані з таблиці для повідомлень може бути використано дуже багато методів, зокрема можна імплементувати додатковий сервіс, що буде зчитувати постійно дані з цієї таблиці і надсилати повідомлення в брокер повідомлень, а можна налаштувати брокер повідомлень, щоб отримувати ці дані напряму із таблиці, без сервісів-посередників, як ретранслятор цих повідомлень.

За допомогою даного патерну можна впевнитись в тому, що локальні транзакції мікросервісів, та продовження послідовності цих транзакцій, спрацюють або одночасно, або одночасно не спрацюють. Однак є проблема з процесом надсилання повідомлень іншим сервісам. З використанням патерну Transactional Outbox, взаємодія між мікросервісами є асинхронною. Більш того, самі мікросервіси не беруть участі у надсиланні повідомлень - повідомлення зчитуються брокером повідомлень в певний момент часу після їх збереження, та будуть отримані пізніше іншими мікросервісами. Однак потрібно усвідомлювати, що мережева взаємодія не завжди може бути стабільною. І якщо брокер повідомлень якийсь час не зчитає дані з outbox-таблиці - їх обробка відбудеться пізніше (“в кінцевому рахунку”, англ. - “*eventually*”). Але якщо повідомлення уже зчитано, і другий мікросервіс не був доступний через мережу, можуть бути проблеми з доставкою повідомлення.

Загалом, розрізняють два основних типи доставки повідомлення з допомогою брокерів повідомлень. Перший - “не більше одного разу” (англ. - “*at most once*”), другий - “щонайменше один раз” (англ. - “*at least once*”). В першому випадку, повідомлення будуть надсилатись мікросервісам брокером повідомлень рівно один раз, не важливо від того, чи отримані були ці повідомлення. В такому випадку, якщо мікросервіс не був доступний в цей момент - цього повідомлення він не отримає. Якщо використати цей спосіб до взаємодії між мікросервісами, дуже високим є ймовірність втратити цілісність даних. Якщо ж розглянути другий спосіб, то тоді, якщо мікросервіс був недоступний під час відправки повідомлення, повідомлення буде надіслано ще раз. Однак, існує ймовірність, що перше повідомлення, перед моментом відключення мікросервісу, було опрацьоване, але не було сповіщено брокера повідомлень про успіх цього опрацювання. Тоді, брокер повідомлень буде надсилати це повідомлення ще раз,

поки не отримає підтвердження про успіх опрацювання. В такому випадку, існує ймовірність про створення дублікатів всередині бази даних мікросервісу, тож API повинно відповідати відношенню ідемпотентності [27]. Оскільки не можна на 100% гарантувати стабільність мережевого з'єднання, ситуація, коли повідомлення буде доставлено від брокера повідомлень до мікросервісу рівно один раз, не більше і не менше - неможлива. Тому, саме для такого роду повідомлень зазвичай використовується спосіб "щонайменше один раз". І тому постає питання про те, як забезпечити унікальність записів в базі даних, якщо повідомлення можуть бути доставлені більше одного разу. Не менш важливим є і те, що буде використано додаткові ресурси системи, апаратного забезпечення, що, очевидно, не бажано.

Для вирішення проблеми унікальності даних використовують Transactional Inbox. Цей патерн є продовженням ідеї Transactional Outbox. Але на відміну від ідеї запису повідомлення в кінці транзакції разом із необхідними даними, суть Transactional Inbox в тому, щоб після отримання повідомлення від брокера повідомлень одразу записати це повідомлення в inbox-таблицю в базі даних. А так як у всіх повідомлень, побудованих в цьому патерні, є певний унікальний ідентифікатор, то можна легко розпізнати дублікат, оскільки тоді ідентифікатор повідомлення уже буде присутнім в даній таблиці [28]. Після чого, через деякий час, мікросервіс буде зчитувати повідомлення з цієї таблиці і обробляти їх, наприклад, за допомогою планувальника задач (рис. 2.10).

Рисунок 2 - Принцип роботи патерну Trasnactional Inbox

Отже, якщо скомбінувати використання патернів Transactional Outbox і Inbox, можна отримати гарантію успішно проведених транзакцій, а також доставки повідомлень наступним мікросервісам в ланцюгу викликів логіки в розподіленій системі, при чому наблизившись до типу доставки повідомлень через брокер повідомлень “рівно раз” (англ. - “*exactly once*”).

Отже, патерни проектування Transactional Outbox та Inbox дозволяють вирішити проблеми синхронізації операцій коміту транзакції на надсилання повідомлень іншим сервісом. Також завдяки ним можна бути впевненим в доставці цих повідомлень, таким чином передбачаючи нестабільність мережевого з’єднання чи/та системних і апаратних несправностей, що змусять мікросервісі в мережі працювати нестабільно, або ж взагалі не працювати. Однак, дана комбінація патернів не передбачає алгоритмів для забезпечення узгодженості даних, у випадку помилок на рівні системи, тобто при проблемах, що можуть виникнути в ході виконання логіки в мікросервісах, а також відкату стану систем, у настанні такого випадку.

При цьому варто зазначити, що використання патерну проектування SAGA разом з патернами Transactional Outbox та Inbox збільшує ефективність їх використання, не заважаючи, а навпаки допомагаючи один одному. Таким чином, за допомогою SAGA можна спроектувати алгоритмічне вирішення проблеми узгодженості даних між мікросервісами, а за допомогою Transactional Outbox та Inbox забезпечити стабільність асинхронної мережевої взаємодії між мікросервісами.

1.1.Патерн проектування “SAGA”

SAGA - це патерн проектування розподілених систем, задля забезпечення коректного виконання послідовності із локальних транзакцій. Тобто, в контексті даного патерну, ідея про імплементацію повноцінних розподілених транзакцій відкидається. Тому, акцент будується на грамотній організації та послідовності роботи локальних транзакцій мікросервісів. Основною побудови патерну SAGA, є відкат стану системи за допомогою компенсаційних транзакцій. В даному випадку, компенсаційна транзакція — це транзакція, що має протилежний ефект

до іншої транзакції, або ж відміняє ефект іншої транзакції. Компенсаційні транзакції можуть бути викликані за умови, що в ході виконання послідовності із транзакцій в різних мікросервісах, всередині процесу виникла помилка, і є необхідність у відкаті стану баз даних інших мікросервісів, що уже завершили свої транзакції. Обов'язковим при визначенні логіки цих компенсаційних транзакцій є те, що вони мають відповідати відношенню ідемпотентності, а також, за умови ідемпотентності, мати змогу повторюватись, у випадку невдачі виконання операції. SAGA патерн дозволяє згрупувати всі транзакції в єдину послідовність і гарантує, що вся ця послідовність або повністю виконається успішно, або ж повністю неуспішно [22].

Існує два основних способи досягнення побудови цього патерну: оркестрація та хореографія.

У випадку хореографії, усі процеси опрацювання послідовності транзакцій відбуваються за асинхронної взаємодії між мікросервісами. Коли до мікросервіса поступив запит на обробку певної логіки, яка вимагає виклику іншого мікросервісу, перший сервіс в ході роботи збіргає новий екземпляр сутності, що потребує виконання SAGA`и, після чого публікує певну подію, або повідомлення, на яке другий сервіс очікує, для продовження виконання всієї логіки в ході послідовності локальних транзакції. Той в свою чергу може також публікувати події, на які інші мікросервіси очікують. У випадку, коли всередині послідовності сталась помилка, і локальна транзакція мікросервісу виконала відкат, в ході цього відкату сервіс викликає подію. Цю подію слухає мікросервіс, що знаходиться в попередній ланці ланцюга з транзакцій. І отримавши цю подію, мікросервіс виконує компенсуючу транзакцію, до тої, що була ним виконана попередньо. Після завершення цієї транзакції, він також публікує подію, для виконання компенсаційної транзакції попередньої ланки ланцюга. І так, поки не буде виконана відміна усіх змін в ході загального процесу виконання розподіленої транзакції (приклад на рис. 2.4).

За даної імплементації, потрібно налаштувати систему так, щоб при виконанні однієї локальної транзакції було обов'язковим ланцюговий відкат усієї

послідовності із локальних транзакцій. І це додає складності в імплементації патерну, хоча і не надлишково.

Цей процес можна спростити в інший спосіб реалізації хореографії. Відмінність його від попереднього в тому, що вводиться додатковий сервіс, в якому відбувається моніторинг виконання локальних транзакцій мікросервісів. Цей сервіс відповідальний за збереження послідовного списку із викликів транзакцій, що також можна назвати журналом SAGA (англ. - "*Saga log"*) [18]. Цей сервіс прийнято називати координатором виконань SAGA (англ. - *"Saga execution coordinator - "**SEC**""*). Процес взаємодії мікросервісів із сервісом координатором SAGA зображено на рис. 2.5:

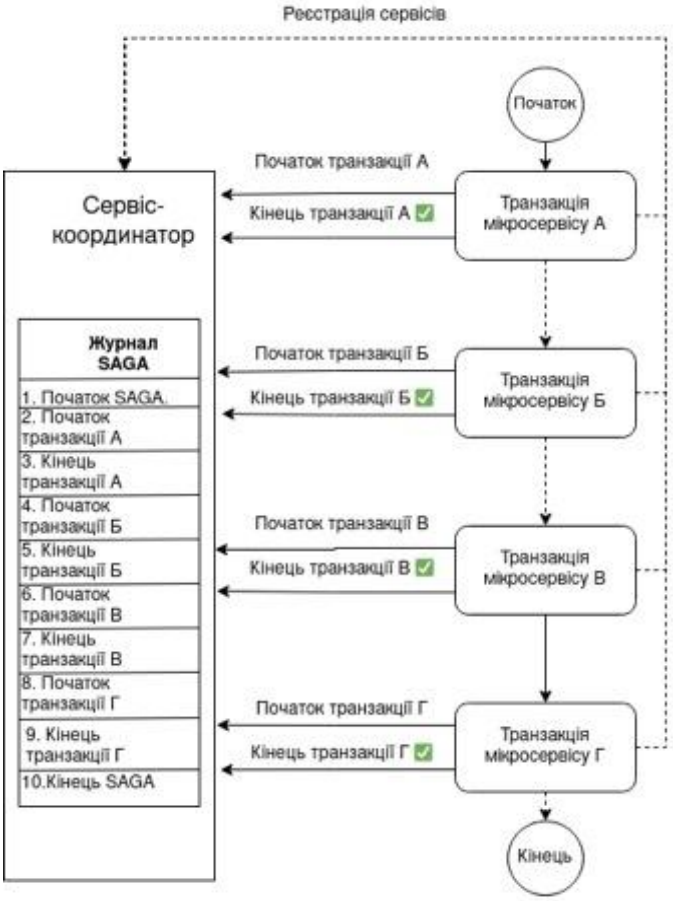

Рисунок 2.8– Приклад роботи хореографії SAGA з сервісом-координатором

У випадку, якщо одна із локальних транзакцій мікросервісів завершиться з помилкою, в рамках відповідальності сервіса-координатора буде виклик усіх

необхідних компенсаційних транзакцій у зворотному порядку, у відповідності до журналу SAGA, як це зображено на рис. 2.6:

Рисунок 2.9. – Приклад роботи хореографії SAGA з сервісом-координатором, з викликом компенсаційних транзакцій

При цьому, у випадку невдачі виклику компенсаційних транзакцій, вони будуть повторені, допоки не виконаються. Саме тому є вимога до компенсаційних транзакцій на те, щоб задовольняти можливості повторення та ідемпотентності.

Тепер розглянемо інший спосіб до побудови SAGA - оркестрація. В попередньому випадку, для реалізації хореографії, можна ввести сервіс-координатор, в журнал SAGA якого буде додаватись послідовність із виконаних локальних транзакцій самими мікросервісами. І цей сервіс може скористатись цим журналом для відкату у зворотному порядку усіх локальних транзакцій мікросервісів, що були виконані до цього. Ідея оркестрації полягає в

тому, щоб використати сервіс-координатор не просто як полегшення для послідовного відкату, як загального регулювальника процесу викликів усіх локальних транзакцій.

На початку виконання SAGA, в джерелі виникнення запиту створюється запит на виконання SAGA, зазвичай створюється екземпляр сутності тієї чи іншої SAGA`и. Принцип асинхронного способу взаємодії мікросервісів залишається, тому далі ця SAGA буде виконуватись в асинхронному режимі. Після збереження першої сутності SAGA`и, публікується повідомлення, яке слухає інший сервіс, що є наступною ланкою в послідовності транзакцій цієї SAGA. Однак після успішного завершення цієї локальної транзакції, мікросервіс публікує повідомлення, яке слухає сервіс-координатор. Отримавши це повідомлення сервіс координатор зберігає інформацію про успішне виконання даного кроку SAGA, після чого публікує повідомлення для другого мікросервіса. І так далі, поки не закінчиться виконання SAGA, як показано на рис. 2.7:

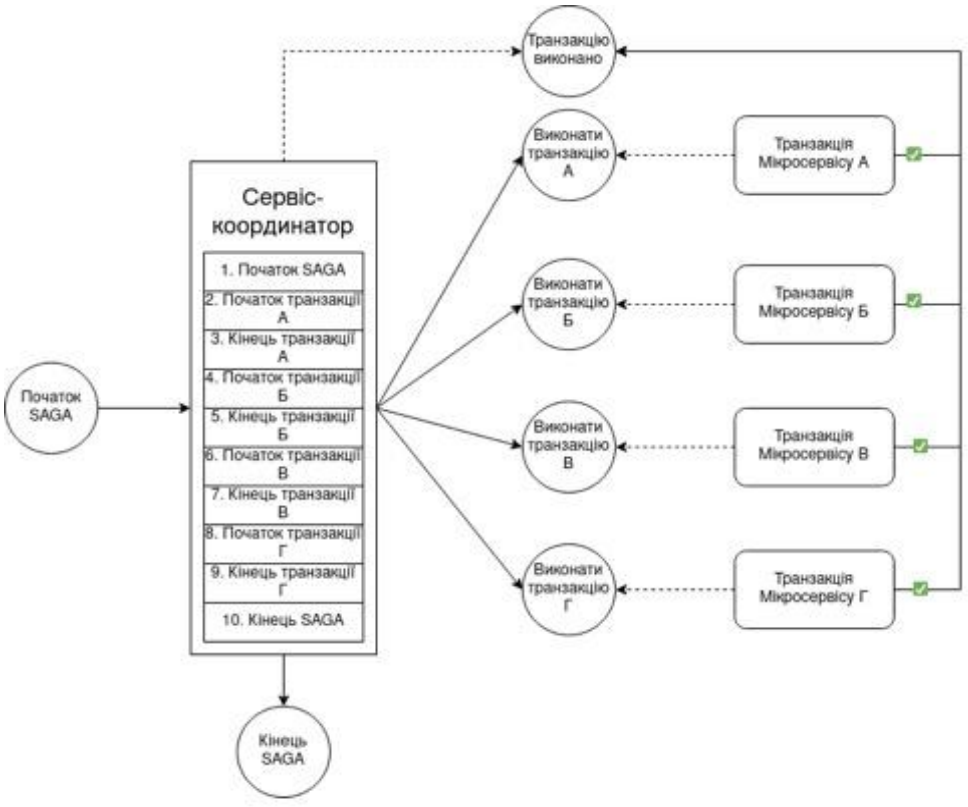

Рисунок 2.10– Приклад роботи оркестрації SAGA

Якщо одна із транзакцій цієї послідовності завершується з помилкою, то цей сервіс-координатор буде відповідальний за виклик компенсаційної транзакції, як це показано на рис. 2.8:

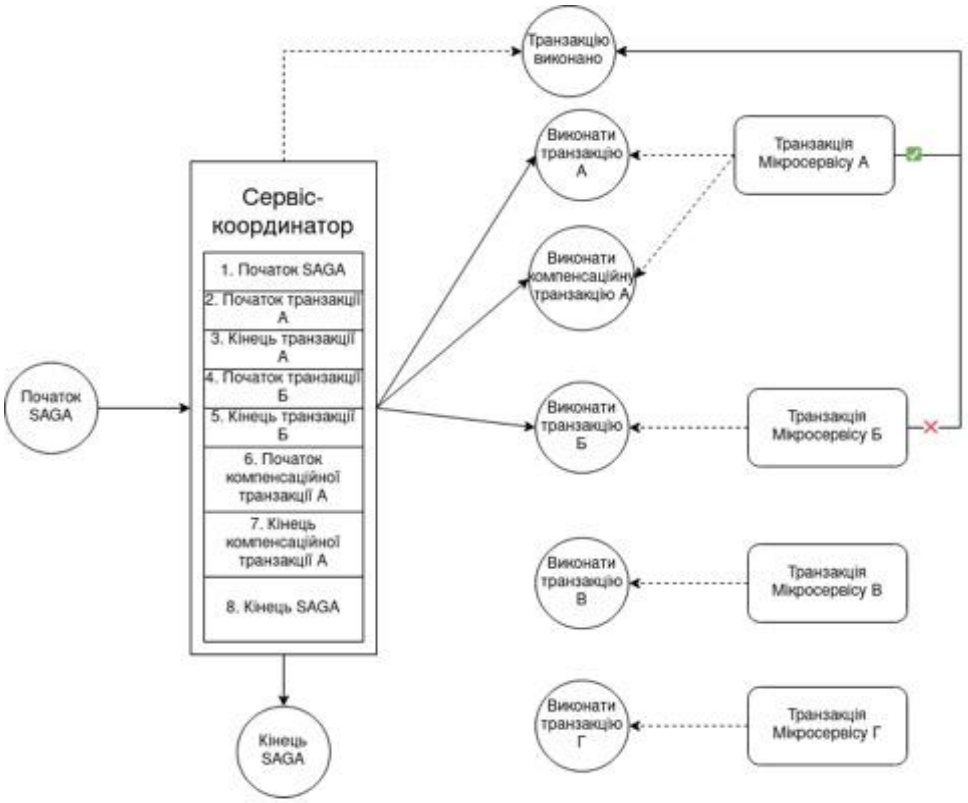

Рисунок 2.11- Приклад роботи оркестрації SAGA, з викликом компенсаційних транзакцій

За допомогою патерну проектування SAGA, та його головних концепцій в управління локальними транзакціями можна вирішити проблему узгодженості даних в мікросервісній архітектурі, коли локальні транзакції мікросервісів можуть завершуватись із помилкою. За рахунок координації в рамках асинхронної взаємодії між сервісами, а також виконанням компенсаційних транзакцій до тих, що були завершені із помилкою, можна досягти консистентності в даних між мікросервісами, але не в умовах реального часу, а згодом, при виконанні транзакцій мікросервісами, що спрацювали після отримання повідомлення, або при публікації події (залежно від способу імплементації синхронної взаємодії між мікросервісами).

Даний патерн проектування має багато переваг, окрім зазначеного вище. Так, за допомогою обох реалізацій SAGA, можна досягти низького рівня прив’язки між мікросервісами системи, за рахунок того, що всі мікросервіси в системі знають лише про події/повідомлення, що стимулюють виконання їх локальних транзакцій, а також про події/повідомлення, які мають бути після виконання їх локальних транзакцій: у випадку оркестрації - це буде подія/повідомлення про успіх транзакції до головного сервіса-координатора. У випадку хореографії - подія/повідомлення про потребу виконання наступної локальної транзакції іншим сервісом, що очікуватиме на цю подію/повідомлення.

Для хореографії без сервіса-координатора буде проблемою комплексність послідовності асинхронних викликів локальних транзакцій між мікросервісами, однак це зменшить складність в імплементації патерну. Якщо використовувати сервіс-координатор для імплементації SAGA, до логіка викликів локальних транзакцій мікросервісів стає легшою для розуміння, однак логіка в самому сервісі-координаторі буде доволі складною як для підтримки, так і для розуміння. Окрім цього, сервіс-координатор, як і у випадку з двофазними транзакціями, стає місцем ризику усієї системи, адже коли він вийде з ладу, процес синхронізації даних між собою буде втрачено. Також варто зазначити, що через те, що в ході SAGA потрібно викликати декілька різних локальних транзакцій в середині мережі мікросервісів, які зазвичай мають власні окремі бази даних, втрачається ізоляція транзакцій на рівні всієї послідовності викликів.

Однак, на жаль реалізація даного патерну не захищає від нестабільності мережевого з’єднання, і в разі настання такого, можна буде очікувати неузгодженості даних між сервісами.