УДК 004.42:004.75
DOI https://doi.org/10.32782/tnv-tech.2023.4.4

РЕАЛІЗАЦІЯ SAGA З ВИКОРИСТАННЯМ ШАБЛОНУ OUTBOX

Бугаєва І. Г. *– кандидат технічних наук,*
доцент кафедри технічної кібернетики й інформаційних технологій
імені професора Р. В. Меркта
Одеського національного морського університету
ORCID ID: 0000-0002-2839-9266

Мікросервісна архітектура надає ряд переваг, до яких можна віднести гнучкість, стійкість до відмов, можливість повторного використання функціональності. Але розподілений характер таких систем призводить до складності управління взаємодією та узгодженістю даних між різними сервісами. В додатках необхідно виконувати транзакції, які охоплюють кілька сервісів, кожен з яких має власну базу даних і є ізольованим. Для управління транзакціями в системах з мікросервісною архітектурою використовується шаблон проєктування Saga, який розподілену транзакцію представляє як послідовність локальних транзакцій. Зазвичай локальна транзакція оновлює базу даних сервісу та публікує повідомлення, щоб ініціювати наступну локальну транзакцію у сазі. Оновлення бази даних та публікація повідомлення повинні виконуватися атомарно, інакше система може перейти у неузгоджений стан. Це може статися, якщо, наприклад, після фіксації змін у базі даних повідомлення не буде надіслано через збій в мережі. Для забезпечення узгодженості даних можна використовувати шаблони Transitional Outbox або Event Sourcing. Стаття присвячена дослідженню особливостей застосування цих шаблонів при реалізації розподілених транзакцій за допомогою Saga, виявленню переваг та недоліків кожного з підходів. Event Sourcing дозволяє зберігати історію зміни стану об'єкта, відтворювати події для відновлення його стану на певний момент часу. Недоліком цього шаблону є те, що він складний у вивченні. У разі використання складних запитів до бази даних у поєднанні з Event Sourcing слід використовувати патерн CQRS. В роботі представлена архітектура додатку на основі мікросервісів, побудована з використанням шаблонів проєктування Saga та Outbox. Сервіси взаємодіють між собою асинхронно за допомогою Apache Kafka та конекторів Debezium, які відстежують зміни в базах даних та надсилають повідомлення в брокер. Сервіси отримують повідомлення з брокеру відразу після публікації.

Ключові слова: *мікросервісна архітектура, розподілені транзакції, узгодженість даних, шаблон проєктування Saga, Transitional Outbox, Event Sourcing.*

Buhaieva I. H. Saga implementation using the Outbox pattern

Microservice architecture provides a number of advantages, which include flexibility, fault tolerance, and the ability to reuse functionality. But the distributed nature of such systems makes it difficult to manage interactions and data consistency between different services. Applications need to perform transactions spanning multiple services, each with its own database and isolated. To manage transactions in systems with a microservice architecture, the Saga design pattern is used, which represents a distributed transaction as a sequence of local transactions. Typically, a local transaction updates the service database and publishes a message to initiate the next local transaction in the saga. Updating the database and publishing a message must be done atomically, otherwise the system may end up in an inconsistent state. This can happen if, for example, after committing changes to the database, the message is not sent due to network failure. The Transitional Outbox or Event Sourcing pattern can be used to ensure data consistency. The article is devoted to research the features of using these patterns when implementing distributed transactions using Saga, identifying the advantages and disadvantages of each approach. Event Sourcing allows you to save the history of changes in the state of an object, play back events to restore its state at a certain point in time. The disadvantage of this pattern is that it is difficult to learn. When using complex database queries, in combination with Event Sourcing you should use the CQRS pattern. The paper presents a microservices-based application architecture built using the Saga and Outbox design patterns. Services interact with each other asynchronously using Apache Kafka and Debezium connectors, which monitor changes in the databases and transmit messages to the broker. Services receive messages from the broker immediately after publication.

Key words: *microservice architecture, distributed transactions, data consistency, Saga design pattern, Transitional Outbox, Event Sourcing.*

Вступ. У багатьох додатках з мікросервісною архітектурою необхідно здійснювати транзакції, що охоплюють декілька сервісів. Для управління такими транзакціями застосовується шаблон проєктування Saga [1; 2]. Він забезпечує виконання або всіх локальних транзакцій, з яких складається сага, або невиконання жодної, якщо якась транзакція не може бути реалізована з якоїсь причини.

Якщо при виконанні локальної транзакції сервіс записує дані до своєї бази даних, а потім відправляє повідомлення в чергу для іншої служби, що бере участь у сазі, то існує ймовірність того, що додаток не буде працювати надійно, оскільки після фіксації в базі даних може статися збій програми, повідомлення в інший сервіс не буде надіслано та система залишиться у неузгодженому стані. Надсилання повідомлення може завершитися збоєм через різні причини: помилка мережі, недоступність служби повідомлень, збій вузла. Якщо повідомлення надсилається раніше запису у базу даних, існує можливість збою оновлення бази даних.

Для забезпечення узгодженості даних у системі необхідно виконувати операції оновлення локальної бази даних та надсилання повідомлень іншим сервісам як атомарну операцію. Вирішенням цієї проблеми може бути використання шаблону Transactional Outbox або Event Sourcing.

Метою статті є дослідження особливостей реалізації Saga із застосуванням шаблонів Outbox та Event Sourcing. На етапі проєктування програмної системи з мікросервісною архітектурою важливо знати переваги та недоліки кожного з цих підходів.

Аналіз досліджень і публікацій. Одним із способів надійної публікації повідомлень є використання шаблону Transactional Outbox [3]. Згідно з ним сервіс, що використовує реляційну СУБД, вставляє повідомлення в таблицю outbox у рамках локальної транзакції, що оновлює базу даних. Окремий процес ретрансляції повідомлень читає цю таблицю та передає повідомлення брокеру. Якщо сервіс використовує NoSQL СУБД, при оновленні бізнес-об'єкта повідомлення додається до атрибуту відповідного запису.

Для доставки повідомлень від бази даних до брокера зазвичай використовується відстеження транзакційного журналу бази даних. У ньому фіксуються всі оновлення, що виконані додатком. Ретранслятор читає цей журнал і публікує кожну зміну як повідомлення для брокера.

Одним з прикладів реалізації цього підходу є використання Debezium – розподіленої платформи, яка працює на основі функцій системи відстеження змінених даних у базі даних.

Debezium побудований поверх Apache Kafka та надає конектори, які публікують у брокері Kafka зміни бази даних, що вносяться додатком. Інші сервіси можуть зчитувати результуючі записи подій з тем Kafka. Кожен з конекторів працює з певною СУБД.

Необхідно також враховувати те, що брокер зазвичай гарантує хоча б одну доставку. Обробники подій, які не є ідемпотентними, повинні самостійно виявити і відкинути повідомлення, що повторюються.

Ще один архітектурний шаблон, який можна використовувати для надійної публікації повідомлень – Event Sourcing [4]. Відповідно до нього дані зберігаються у вигляді подій у спеціальному сховищі подій. У традиційному підході до баз даних зберігається кінцевий стан бізнес-об'єкта. При використанні Event Sourcing зберігаються усі проміжні стани у вигляді послідовності подій. Кінцевий стан одержують послідовним застосуванням усіх проміжних станів.

Сховище подій являє собою гібрид бази даних та брокера повідомлень. Воно є базою даних, тому що має API для додавання та зчитування подій за допомогою первинного ключа, але воно також, як і брокер повідомлень, має API для підписки на події.

Існує кілька різних способів реалізації сховища подій. Можна застосовувати для цього реляційну СУБД. Це простий, хоч і низькопродуктивний спосіб публікації подій. Передплатники періодично опитують таблицю events для отримання нових даних. Інший варіант – використовувати спеціалізоване сховище подій, яке, як правило, надає багатий набір функцій, більш високу продуктивність та масштабованість. Прикладом такого сховища подій є Event Store.

При використанні Event Sourcing зберігається історія зміни стану об'єкта, яка може стати в нагоді для завдань аудиту або моніторингу. Але цей шаблон має й недоліки. Він складний у вивченні. Схема подій може розвиватися з часом і для відновлення стану об'єкта сервісу може знадобитися обробити події, що відповідають кільком різним версіям схеми [5; 6]. Крім того, при зверненні до сховища подій можуть виникати труднощі під час використання складних запитів. Тоді слід реалізовувати запити за допомогою шаблону Command Query Responsibility Segregation (CQRS), який розділяє операції читання та оновлення сховища даних.

Для виконання проєкту був обраний шаблон Outbox, як більш простий у розробці та надійний у застосуванні.

Виклад основного матеріалу. Розроблено додаток з мікросервісною архітектурою, що забезпечує взаємодію трьох служб, які розгортаються незалежно та мають свої бази даних: order – обробляє запити на замовлення подорожі, hotel – бронювання готелю, avia – бронювання авіаквитків. Оскільки виконання замовлення можливе лише при успішному бронюванні готелю та авіаквитків для перельоту, то при його обробці має виконуватися транзакція, що охоплює сервіси hotel та avia. Для здійснення розподіленої транзакції використовувався шаблон Saga. Для забезпечення узгодженості даних при оновленні баз даних та публікації

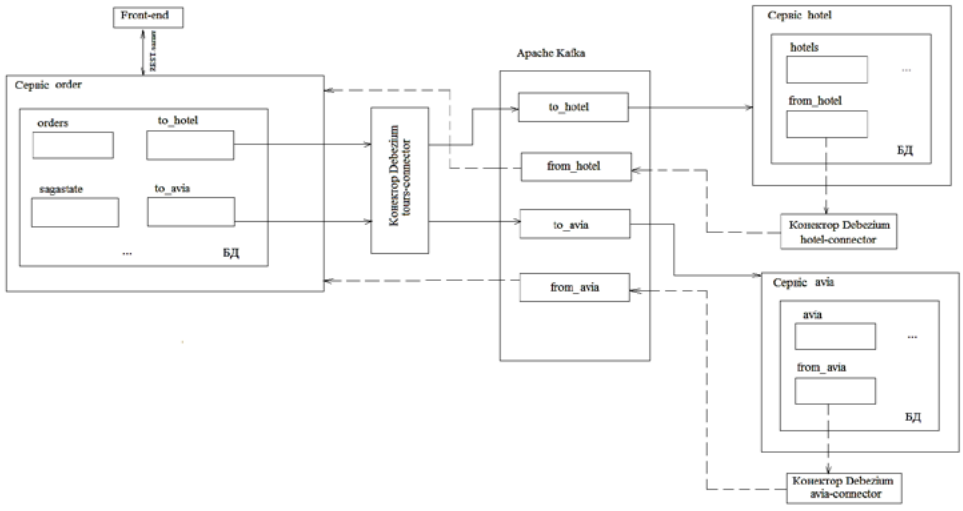

Рис. 1. Архітектура додатку

повідомлень застосовувався шаблон Outbox. Конектори Debezium зчитували зміни, які вносилися у бази даних сервісів, та публікували повідомлення у брокері Apache Kafka [7; 8]. Архітектура додатку наведена на рис. 1.

Проєкт було реалізовано у середовищі node.js. Як база даних застосовувалася СУБД MySQL. Для запуску необхідних служб використовувалися образи контейнерів Docker. Для роботи з Debezium потрібні три окремі служби: ZooKeeper, Kafka Connect та служба конектора Debezium. Для зв'язку зі службою Kafka Connect застосовувалися команди curl для надсилання запитів API на порт 8083 вузла Docker, який було зіставлено з портом 8083 у контейнері connect під час запуску Kafka Connect.

Після початку роботи служб Debezium та MySQL реєструвалися конектори Debezium MySQL з іменами "tours-connector", "hotel-connector", "avia-connector", щоб вони могли розпочати моніторинг відповідних таблиць баз даних MySQL.

Після реєстрації конектори починають відстежувати файли binlog сервера бази даних і генерують події зміни для кожного рядка, що змінюється, таблиць баз даних, які визначаються в параметрах конфігурації конекторів. Параметр "tasks. max" зі значенням "1" вказує, що у кожний момент часу має виконуватися лише одне завдання, що забезпечує правильний порядок та обробку подій. Для відстеження бази даних MySQL використовується клас MySqlConnector.

Задаються також ім'я хоста сервера бази даних, яке є назвою контейнера Docker з запущеним сервером MySQL, номер порту сервера бази даних, ім'я користувача БД і пароль. У параметрах "database.include.list" й "table.whitelist" вказуються відповідно ім'я бази даних та назви таблиць, зміни у яких необхідно відстежувати.

Шаблон Saga може бути реалізований на основі хореографії чи оркестрації. У сазі на основі хореографії відсутня центральна точка управління. Кожна локальна транзакція у службі публікує події, які запускають локальні транзакції в інших сервісах. Сага на основі оркестрації передбачає наявність координатора, який відповідає за керування загальним станом транзакції. У проєкті реалізовано шаблон Saga на основі оркестрації.

Після отримання REST-запиту від клієнта сервіс order викликає функцію, яка є оркестратором саги та повідомляє сервісам, що беруть участь у сазі, які локальні транзакції їм слід виконувати.

Оркестратор реалізований у вигляді кінцевого автомата, що складається з набору станів і переходів між ними, які ініціюються за допомогою подій. Кожен перехід має певну дію, яка означає виклик наступного учасника саги. Наступний перехід та дія, які потрібно виконати, визначаються поточним станом.

На рис. 2 представлена модель кінцевого автомата для саги з бронювання подорожі, яка включає наступні стани:

0 – створення замовлення;

1 – бронювання готелю;

2 – бронювання авіаквитків;

3 – замовлення підтверджено;

4 – компенсаційні дії у сервісі hotel;

5 – замовлення на бронювання поїздки відхилено.

Кінцевий автомат визначає множину переходів станів. Після створення замовлення (стан 0) запускається виконання першого кроку саги – бронювання готелю, і відбувається перехід у стан 1. При успішному бронюванні готелю ініціюється наступний крок саги – бронювання авіаквитків, і відбувається перехід у стан 2. В іншому випадку – перехід у стан 5 «Замовлення відхилено».

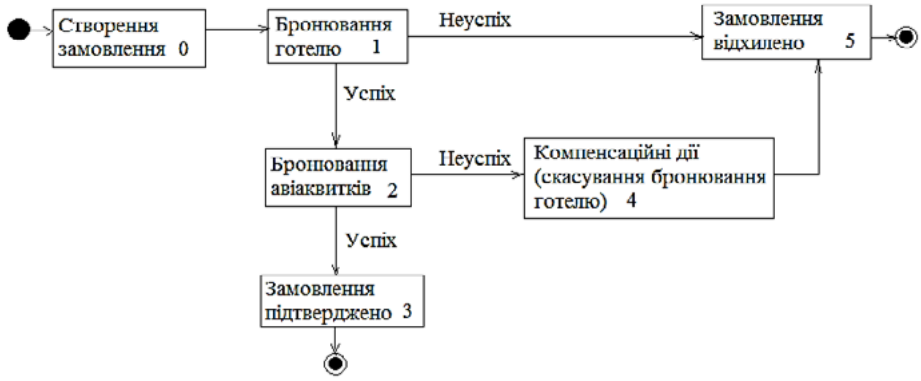

Рис. 2. Модель кінцевого автомата для саги «Бронювання подорожі»

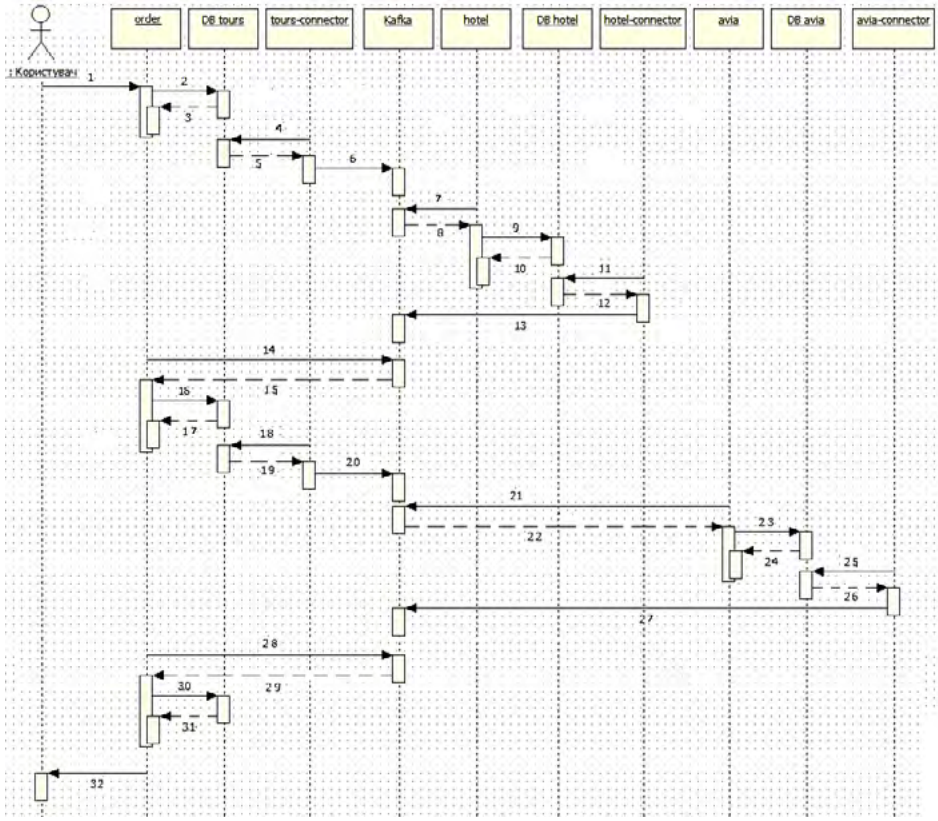

Рис. 3. Успішне виконання саги з використанням Outbox

При успішному бронюванні авіаквитків здійснюється перехід у стан 3 «Замовлення підтверджено». Якщо забронювати квитки не вдалося, то необхідно провести компенсаційні дії в сервісі hotel зі скасування бронювання готелю. Відбувається перехід у стан 4. Після виконання компенсаційних дій здійснюється перехід у стан 5 «Замовлення відхилено».

На діаграмі послідовності (рис. 3) представлено успішне виконання саги з бронювання подорожі з використанням шаблону Outbox.

При отриманні запиту від клієнта на замовлення подорожі (крок 1) оркестратор зберігає дані в базі даних сервіса order (кроки 2, 3). В рамках однієї транзакції поновлення бази даних інформація додається в таблицю to_hotel, яка містить поля: id_order – номер замовлення, state – номер стану, statedescript – опис стану, payload – об'єкт типу json з даними для бронювання готелю. Поле state отримує значення 1, що визначає номер поточного стану, поле statedescript – рядок "hotel_started".

При зміні даних у таблиці to_hotel конектор Debezium з ім'ям tours-connector зчитує запис з цієї таблиці з оновленими даними (кроки 4, 5) і публікує повідомлення в брокері Kafka в темі з назвою, що відповідає назві таблиці (крок 6).

Сервіс hotel підписаний на повідомлення з теми "to_hotel" брокера. Після публікації даних він отримує їх (кроки 7, 8) і здійснює запит до своєї бази даних на бронювання готелю (кроки 9, 10). Як складова цієї транзакції виконується оновлення таблиці from_hotel, яка має таку ж структуру, що і таблиця to_hotel. Поле state отримує значення 2, що вказує номер наступного стану системи в моделі кінцевого автомата, наведеного вище. В поле statedescript записується рядок "hotel_successed", в payload зберігається об'єкт з даними про заброньований готель.

Якби запит про бронювання готелю закінчився неуспішно, то поле state отримало б значення 5, statedescript – "hotel_failed", payload – порожній об'єкт.

В таблиці hotelstate сервісу hotel зберігається інформація про оброблені повідомлення. Для запобігання одержанню дубльованих даних з брокера виконується перевірка: якщо повідомлення з такими номерами замовлення та стану вже оброблялося, воно буде проігноровано.

При зміні даних в таблиці from_hotel виконується зчитування оновлених даних з неї за допомогою конектора "hotel-connector" (кроки 11, 12) та публікація повідомлення у відповідній темі брокера (крок 13).

Оркестратор підписаний на теми "from_hotel" і "from_avia". Отримавши повідомлення з теми from_hotel (кроки 14, 15), він оновлює базу даних tours і в рамках однієї транзакції додає до таблиці to_avia (кроки 16, 17) інформацію, необхідну для бронювання авіаквитків, тим самим ініціюючи виконання наступного кроку саги. Таблиця to_avia має таку ж структуру, що і таблиця to_hotel. Поле state отримує значення 2, statedescript – рядок "avia_started", payload – об'єкт з даними для сервісу avia. При зміні даних у таблиці to_avia конектор tours-connector зчитує з неї оновлений запис (кроки 18, 19) і публікує повідомлення в темі to_avia (крок 20). Сервіс avia отримує дані з брокера (кроки 21, 22) і виконує запит до своєї бази даних на бронювання авіаквитків (кроки 23, 24). При обробці запиту оновлюються дані таблиці from_avia. Поле state отримує значення 3, поле statedescript – рядок "avia_successed", payload – об'єкт з даними про заброньовані квитки.

Якби не вдалося забронювати квитки, поле state таблиці from_avia отримало б значення 4, statedescript – "avia_failed", payload – порожній об'єкт. Значення 4 поля state вказувало б на необхідність переходу до виконання компенсаційних дій у сервісі hotel.

При оновленні таблиці from_avia дані зчитуються конектором “avia-connector” (кроки 25, 26) та публікуються (крок 27). Оркестратор отримує їх (кроки 28, 29) та оновлює локальну базу даних (кроки 30, 31), також зберігаючи у полі orderstate відповідного запису таблиці orders значення “success”, що є показником того, що сага завершилася успішно.

Якби запит до сервісу avia був невдалим, оркестратор додав би у таблицю to_hotel запис зі значеннями полів state = 4 і statedescript = ”hotel_compensating”, що запустило б виконання компенсаційних дій сервісом hotel. Служба hotel, отримавши повідомлення, внесла б зміни до своєї бази даних щодо скасування бронювання готелю і оновила б таблицю from_hotel, записавши значення полів: state = 5, statedescript = “hotel_compensated”.

Сервіс order періодично опитує таблицю orders локальної бази даних. Коли у записі поточного замовлення поле orderstate набуває значення ”success”, клієнту надсилається відповідь з даними щодо бронювання поїздки (крок 32).

Висновки. У статті розглядаються особливості реалізації шаблону Saga, який використовується для управління транзакціями у додатках з мікросервісною архітектурою. Досліджуються два підходи для забезпечення узгодженості даних у таких системах – застосування шаблонів Transitional Outbox та Event Sourcing.

Event Sourcing є потужним архітектурним шаблоном, який слід використовувати, коли необхідно зберігати історію зміни стану об’єкта, наприклад, для завдань аудиту або моніторингу. Він дозволяє відтворювати події для відновлення стану об’єкта на певний момент часу.

Недоліком цього шаблону є те, що він складний у вивченні. У разі використання складних запитів до бази даних у поєднанні з Event Sourcing слід використовувати шаблон CQRS, що може ускладнити розробку додатку.

Обидва шаблони Outbox та Event Sourcing дозволяють надійно публікувати повідомлення/події у брокері повідомлень. Для проєкту було обрано більш простий у використанні шаблон Outbox. При реалізації цього підходу використовувалися конектори Debezium для відстеження змін в базах даних сервісів. Застосування шаблону Outbox та конекторів дозволяє оновлювати базу даних та публікувати повідомлення атомарно, тим самим забезпечуючи узгодженість даних у мікросервісах.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ:

1. Richardson Ch. *Microservices Patterns.* Manning Publications, 2019. 520 p.

2. Бугаєва І.Г., Розум М.В., Ларін Д.Г., Ткаченко М.Г. Управління транзакціями в мікросервісній архітектурі. *Таврійський науковий вісник. Серія: Технічні науки*. 2023. № 2. С. 3–13.

3. Richardson, Ch. Pattern: Transactional outbox. URL: https:// microservices.io/patterns/data/transactional-outbox.html (дата звернення: 05.06.2023).

4. Fowler M. Event Sourcing. 2005. URL: https://martinfowler.com/eaaDev/EventSourcing.html (дата звернення: 20.06.2023).

5. Overeem, M., Spoor, M., Jansen, S. The dark side of event sourcing: Managing data conversion. *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering.* 2017. DOI:10.1109/SANER. 2017.7884621

6. Overeem M., Spoor M., Jansen S., Brinkkemper S. An empirical characterization of event sourced systems and their schema evolution – Lessons from industry. *The Journal of Systems and Software*. 2021. Vol. 178. DOI:10.1016/j.jss.2021.110970.

7. Scott D., Gamov V., Klein D. Kafka in Action. Manning Publications, 2022. 272 p.
8. Debezium. URL: https://debezium.io/ (дата звернення: 11.07.2023).

REFERENCES:

1. Richardson, Ch. (2019). *Microservices Patterns.* Manning Publications.
2. Buhaieva, I.H., Rozum, M.V., Larin, D.H., Tkachenko, M.H. (2023). Upravlinnia tranzaktsiiamy v mikroservisnii arkhitekturi [Transaction management in microservice architecture]. *Tavriiskyi naukovyi visnyk. Seriia: Tekhnichni nauky*, 2, 3–13 [in Ukrainian].
3. Richardson, Ch. Pattern: Transactional outbox. Retrieved from https://microservices.io/patterns/data/transactional-outbox.html.
4. Fowler, M. (2005). Event Sourcing. Retrieved from https://martinfowler.com/eaaDev/EventSourcing.html.
5. Overeem, M., Spoor, M., Jansen, S. (2017). The dark side of event sourcing: Managing data conversion. *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering.* DOI:10.1109/SANER.2017.7884621
6. Overeem, M., Spoor, M., Jansen, S., Brinkkemper, S. (2021). An empirical characterization of event sourced systems and their schema evolution – Lessons from industry. The Journal of Systems and Software, 178. DOI:10.1016/j.jss.2021.110970
7. Scott, D., Gamov, V., Klein, D. (2022). *Kafka in Action*. Manning Publications.
8. Debezium. Retrieved from https://debezium.io/.