Міністерство освіти і науки України
Національний університет «Києво-Могилянська академія»
Факультет інформатики
Кафедра мережних технологій

Магістерська робота

освітній ступінь – магістр

на тему: «**Розробка бібліотеки на .NET для роботи зі знімками (snapshots) для EventStoreDB**»

Виконав: студент 2-го року навчання,
Спеціальності
121 Інженерія Програмного Забезпечення
Осадчук Володимир Ігорович
Керівник Франчук О.В.
доцент, к.т.н.

Рецензент _____
(прізвище та ініціали)

Магістерська робота захищена
з оцінкою _____
Секретар ЕК С.А. Мелещенко_____
«_____» _____ 2023 р.

Київ – 2023

Міністерство освіти і науки України

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра мережних технологій

ЗАТВЕРДЖУЮ
Зав. кафедри мережних технологій,
професор, д.ф.-м.н.
_____ Г. І. Малашонок
(підпис)
“____” _____ 2023 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на магістерську роботу

студенту 2-го р.н. магістерської програми Інженерія Програмного Забезпечення
Осадчуку Володимиру Ігоровичу

Тема: Розробка бібліотеки на .NET для роботи зі знімками (snapshots) для EventStoreDB

Зміст текстової частини до магістерської роботи:

1. Зміст
2. Анотація
3. Вступ
4. Теоритична частина
5. Створення та аналіз клієнтських бібліотек для збережень знімків
6. Застосування EventStore та створеної бібліотеки до веб-застосунку
7. Висновки
8. Список використаної літератури
9. Додатки

Дата видачі “____” _____ 2023 р.

Керівник _____

Завдання отримано _____

Тема: «Розробка бібліотеки на .NET для роботи зі знімками (snapshots) для EventStoreDB»

Календарний план виконання роботи:

№ п/п	Назва етапу дипломного проекту (роботи)	Термін виконання етапу	Примітка
1.	Отримання завдання на магістерську роботу.	жовтень 2022 р.	
2.	Огляд технічної літератури за темою роботи.	жовтень-листопад 2022 р.	
3.	Виконати аналіз клієнтських бібліотек на .NET для роботи з знімками.	листопад - січень 2023 р.	
4.	Написання теоретичної частини курсової роботи.	січень - квітень 2023 р.	
5.	Реалізація практичної частини роботи.	лютий - квітень 2023 р.	
6.	Надання роботи керівнику для перевірки, демонстрація практики.	квітень 2023 р.	
7.	Корегування роботи за результатами перевірки керівником.	травень 2023 р.	
8.	Остаточне оформлення теоретичної частини та слайдів доповіді.	травень 2023 р.	
9.	Попередній захист магістерської роботи.	травень 2023 р.	
10.	Захист магістерської роботи.	червень 2023 р.	

Студент **Осадчук Володимир Ігорович**

Керівник **Франчук Олег Васильович**

“____” _____ 2023 р.

ЗМІСТ

АНОТАЦІЯ

Робота присвячена базі даних, яка спеціалізується на збереженні подій (events) та їх історії в EventStore, а також патернам, які вона реалізує – Event Sourcing та CQRS (Command Query Responsibility Segregation) в контексті мікросервісної архітектури.

Були розглянуті недоліки EventStore та необхідність збереження проміжних подій. Був проведений аналіз існуючих бібліотек на .NET для вирішення цих проблем, а також розроблена власна бібліотека, яка вирішує проблеми EventStore. Проведено порівняння написаної бібліотеки з існуючими та виміряна продуктивність. Розроблену бібліотеку, разом з EventStore, було використано у наявному веб-застосунку для оголошень з прокату паперових книг.

ВСТУП

Все більшої популярності набуває архітектурний стиль під назвою «Мікросервісна архітектура». Це підхід, коли застосунок складається з невеликих, незалежних, самодостатніх сервісів, що можуть працювати разом. Кожен сервіс відповідає за конкретну функціональність програмного продукту і може бути розроблений, розгорнутий та масштабований окремо від інших сервісів.

У такій архітектурі кожен сервіс може бути розгорнутий на окремому сервері або контейнері, що дозволяє незалежно масштабувати окремі компоненти системи в залежності від навантаження. Кожен сервіс може мати свою власну базу даних та інші залежності, що дозволяє розробляти та випускати компоненти системи незалежно один від одного.

Існує багато підходів та патернів для розробки мікросервісних архітектур, кожен з яких має свої переваги та недоліки. Основний патерн, що буде розглянутий, та який буде використано це Event Sourcing – підхід, при якому стан системи зберігається як послідовність подій (events), які відбулися в системі. Кожна подія є записом у журналі, який зберігається у довільному місці, наприклад, у базі даних або у спеціальному системному журналі. Це дозволяє відновити стан системи в будь-який момент часу шляхом відтворення послідовності подій, що відбулися в системі.

Одна з основних технологій для реалізації патернів CQRS (Command Query Responsibility Segregation) та Event Sourcing є EventStore. Це база даних, яка спеціалізується на збереженні подій (events) та їх історії. У мікросервісній архітектурі EventStore може бути використаний для зберігання подій, які стосуються різних сервісів. Одним із головних переваг використання EventStore в мікросервісній архітектурі є можливість повторного відтворення подій, які відбулися в системі. Це дозволяє діагностику помилок та знайти проблемні місця в системі.

Метою даної роботи є розгляд EventStore в контексті мікросервісної архітектури, розбір його проблем та способи їх вирішення. Розглянути існуючі бібліотеки на .NET для збереження проміжних подій, проаналізувати їх проблеми та створити власну бібліотеку як альтернативу, порівняти результати роботи та заміряти продуктивність. Другою частиною даної роботи є застосувати та протестувати створену бібліотеку до вже існуючого веб-застосунку для оголошень з прокату паперових книг, який написаний за домомогою мікросервісної архітектури на фреймворках ASP.NET Core для бекенду та Angular для фронтенду.

РОЗДІЛ 1. ТЕОРЕТИЧНА ЧАСТИНА

В теоретичній частині розглянуто поняття та контекст мікросервісної мікросервісною архітектури, її переваги та недоліки, а також основні підходи у взаємодії між сервісами. Далі будуть розглянуті основні архітектурні патерни мікросервісів. Вкінці розділу буде надано опис EventStore – бази даних для збереження подій (events).

1.1 Мікросервісна Архітектура

Мікросервісна архітектура (Microservices architecture) – це підхід до розробки програмного забезпечення, в якому програма складається з багатьох невеликих, незалежних, самодостатніх сервісів, що можуть функціонувати окремо із мінімальною залежністю один від одного.

Кожен сервіс відповідає за виконання конкретної функції, наприклад, обробка замовлень, авторизація користувачів, розрахунок вартості замовлення, аналіз даних тощо. Сервіси мають відкритий інтерфейс, який може бути використаний користувачами, або ж іншими сервісами для звернення до нього і отримання потрібної інформації або виконання конкретних дій.

Кожен сервіс можна розробити, відтестувати та розгорнути окремо від інших сервісів, що дозволяє швидко вносити зміни в код і покращити функціональність сервісу, не впливаючи на роботу інших компонентів програми. Більше того, такий підхід дозволяє зменшити ризик збоїв у програмі, оскільки в разі виникнення проблеми в одному сервісі, решта компонентів застосунку продовжують працювати без перебоїв.

Така архітектура дозволяє підвищити гнучкість, масштабованість та надійність програмного забезпечення, зменшити час розгортання та впровадження нових функцій, а також знизити витрати на обслуговування і

підтримку програми. Нижче наведено основні переваги даного архітектурного стилю:

1. Гнучкість: дозволяє розробникам швидко вносити зміни в код, що дозволяє підтримувати високу швидкість розробки та релізів. Кожен сервіс можна розробляти, тестувати, моніторити та масштабувати окремо від інших сервісів. Також надо кожним сервісом може працювати окрема команда.
2. Масштабованість: дозволяє легко масштабувати окремі сервіси залежно від навантаження. Тобто, якщо збільшується кількість запитів до сервісу, можна легко збільшити його потужність, не впливаючи на роботу інших сервісів.
3. Надійність: зменшує ризик збоїв у програмі, оскільки в разі виникнення проблеми в одному сервісі, решта компонентів застосунку продовжують працювати без перебоїв.
4. Швидкість розгортання: дозволяє швидко відтворювати середовище для тестування та розгортання окремих сервісів.
5. Розширюваність: дозволяє додавати нові функції до сервісу, не змінюючи код інших сервісів.

Незважаючи на чимало переваг, така архітектура має й свої недоліки, основні з яких:

1. Складність: розробка та підтримка мікросервісів вимагає високого рівня експертизи та досвіду, оскільки програма складається з багатьох окремих сервісів, які повинні взаємодіяти між собою.
2. Витрати на інфраструктуру: вимагає більш складної інфраструктури, оскільки кожен сервіс повинен бути розгорнутий окремо, що може вимагати більшої кількості серверів та засобів моніторингу.
3. Проблеми зі збіркою та інтеграцією: оскільки кожен сервіс може бути розроблений з використанням різних технологій та мов програмування, збірка та інтеграція окремих сервісів може бути нелегким завданням.

4. Складність тестування: кожен сервіс повинен бути протестований окремо, а також потрібно враховувати взаємодію з іншими сервісами.
5. Проблеми зі зберіганням даних: оскільки кожен сервіс може використовувати власну базу даних, це може призводити до проблем з консистентністю даних та взаємодією між різними сервісами.

Але, незважаючи на вищеперераховані недоліки, мікросервісна архітектура є ефективним рішенням для розробки та підтримки різних додатків, якщо вона правильно керована і реалізована.

Також існує багато різноманітних інструментів та технологій для розробки, розгортання, керування та моніторингу сервісів в мікросервісній архітектурі, популярні з яких:

1. Docker є одним з найпопулярніших інструментів для контейнеризації сервісів. Він дозволяє упаковувати сервіси та їх залежності в контейнери, що забезпечує їх ізольованість та переносимість між різними середовищами.
2. Kubernetes є потужною платформою для оркестрації та керування контейнерами. Він дозволяє автоматизувати розгортання, масштабування та керування контейнеризованими сервісами.
3. NGINX, Kong – API Gateway, що допомагає керувати та контролювати доступ до сервісів. Він дозволяє об'єднати різні мікросервіси під одним API, забезпечуючи безпеку, автентифікацію та моніторинг.
4. Prometheus, Grafana, ELK Stack (Elasticsearch, Logstash, Kibana), Splunk, Datadog допомагають відстежувати та моніторити роботу сервісів, збирати та аналізувати логи для виявлення проблем та вдосконалення продуктивності.
5. Ansible, Chef та Puppet дозволяють автоматизувати конфігурацію та розгортання сервісів у різних середовищах. Вони дозволяють описувати інфраструктуру та конфігураційні файли у вигляді коду, що спрощує процес розгортання та забезпечує консистентність середовищ.

6. Istio або Linkerd надають додаткові можливості для моніторингу, безпеки та керування взаємодією між сервісами. Вони додають проміжний шар (sidecar) до кожного контейнера, що забезпечує багатофункціональність, таку як маршрутизація, балансування навантаження, кешування та шифрування.
7. Інструменти для безперервної інтеграції та безперервної доставки, такі як Jenkins, GitLab CI/CD, CircleCI, допомагають автоматизувати процеси збирання, тестування та розгортання мікросервісів. Вони дозволяють швидко і безпечно впроваджувати зміни та випускати нові версії сервісів.
8. New Relic, Dynatrace і AppDynamics дозволяють відстежувати продуктивність мікросервісів, аналізувати метрики продуктивності, виявляти проблеми та оптимізувати швидкість та ефективність сервісів.
9. Elasticsearch, Logstash, Kibana (ELK Stack), Splunk допомагають збирати, аналізувати та візуалізувати логи з різних сервісів у мікросервісній архітектурі. За їх допомогою можна виявляти проблеми, відстежувати діагностику та виконувати аналіз помилок.
10. JUnit, Selenium, Apache JMeter допомагають автоматизувати тестування функціональності, продуктивності та навантаження сервісів.
11. Healthchecks.io та Consul дозволяють відстежувати стан та доступність мікросервісів. Вони можуть надсилати повідомлення про відмови та відновлення сервісів, а також забезпечувати реакцію на аварійні ситуації.

Варто зазначити, що це лише кілька прикладів інструментів, які можуть використовуватися в даній архітектурі. Вибір інструментів залежить від самого проекту, масштабу, бюджету та інших факторів.

1.2 Способи комунікації між сервісами

Розрізняють два основні типи комунікації між компонентами системи (сервісами) в мікросервісній архітектурі – синхронна та асинхронна.

1.2.1 Синхронна комунікація

Це коли один компонент відправляє запит до іншого компонента, очікуючи на відповідь перед продовженням виконання наступних операцій. Тобто, відправник блокується, поки не отримає відповідь від отримувача. Це може викликати затримки, особливо якщо відповідь займає багато часу або якщо отримувач недоступний. Така комунікація часто використовується в класичних монолітних системах, де компоненти взаємодіють безпосередньо.

До неї належать такі типи:

1. HTTP/REST: Це найпоширеніший тип комунікації між мікросервісами. Використання HTTP протоколу і стандартних методів (GET, POST, PUT, DELETE) дозволяє взаємодіяти з сервісами за допомогою RESTful API. Сервіси взаємодіють між собою, надсилаючи HTTP запити та отримуючи HTTP відповіді.
2. RPC (Remote Procedure Call): RPC є протоколом, що дозволяє сервісам викликати методи або функції в інших сервісах. Він дозволяє здійснювати виклики методів через мережу, ніби вони виконуються локально. RPC може використовувати різні протоколи транспорту, такі як gRPC (за допомогою Protocol Buffers), Thrift або RESTful RPC.

1.2.2 Асинхронна комунікація

У випадку асинхронної комунікації, відправник не очікує безпосередньої відповіді від отримувача після відправки запиту. Замість цього, відправник продовжує свою роботу, а отримувач оброблює запит і може відправити відповідь пізніше, коли це буде можливо, або ж не відправити нічого в

залежності від архітектури. Це дозволяє продовжувати виконання операцій без блокування. Асинхронна комунікація часто використовується в мікросервісних системах, де сервіси можуть взаємодіяти асинхронно через повідомлення або події.

Найпопулярнішими підходами такої комунікації є:

1. Message Queue: У цьому типі комунікації сервіси взаємодіють через повідомлення, які відправляються до черги повідомлень (message queue) або брокера повідомлень (message broker). Сервіси можуть відправляти повідомлення до черги, а інші сервіси можуть очікувати повідомлення та прочитувати їх. Цей підхід надає велику гнучкість та масштабованість.
2. Event-driven: В цьому підході сервіси спілкуються через події (events). Коли в певному сервісі відбувається певна подія, він генерує та публікує її, а інші сервіси, які підписалися на ці події, отримують їх і певним чином реагують ни це. Цей підхід дозволяє розрізняти виробників подій та споживачів, що забезпечує розскладку та розширює можливості асинхронної комунікації та реакції на події у розподіленій системі.

Саме event-driven підходу буде приділено найбільше увазі в даній роботі.

1.3 Архітектурні патерни в мікросервісній архітектурі

Архітектурні патерни в мікросервісах використовуються для стандартизації та розподілу відповідальностей між різними компонентами системи. Основна мета використання патернів полягає у полегшенні розробки, супроводу та масштабування мікросервісних архітектур. До причин, чому патерни є важливими в контексті мікросервісної архітектури, можна віднести стандартизацію, розподіл відповідальностей, узгодженість, масштабованість, тестування, супровід, реюзабельність та забезпечення якості.

Крім того, патерни можуть допомогти забезпечити більшу стабільність та масштабованість системи, допомагаючи вирішувати типові проблеми, які зазвичай виникають в процесі розробки та експлуатації.

Серед основних патернів мікросервісної архітектури можна виділити наступні:

1. Service Registry (Реєстр сервісів): використовується для реєстрації та виявлення сервісів в мережевому середовищі. Він дозволяє сервісам зареєструватись у централізованому реєстрі та знаходити інші зареєстровані сервіси, з якими вони можуть взаємодіяти.
2. Circuit Breaker (Переривач ланцюга): використовується для захисту системи від збоїв у взаємодії між сервісами. Він дозволяє обробляти помилки та збої швидко, переключаючись на альтернативні шляхи або надавши збережені дані, замість блокування або збою системи.
3. API Gateway (Шлюз API): використовується як централізована вхідна точка до мікросервісної системи. Він надає єдину точку доступу для клієнтів та забезпечує додаткові функції, напр. аутентифікація, авторизація, маршрутизація і кешування.
4. Database Per Service (База даних на сервіс): розділення бази даних на окремі мікросервіси. Патерн передбачає, що кожен сервіс, який працює з даними, має мати власну базу даних, що сприяє незалежності та ізоляції даних сервісів, а також полегшує масштабування та розвиток.
5. Event Sourcing (Запис подій): використовується для зберігання та обробки подій як основного джерела даних в системі. Кожна дія або зміна в системі представляється як подія, яка зберігається в журналі або базі подій, що дозволяє зберігати повний життєвий цикл даних і стану системи та відтворювати його в будь-який момент часу. Він також дозволяє здійснювати аудит даних, вирішувати конфлікти та робити аналітику на основі потоків подій.
6. Command and Query Responsibility Segregation (CQRS – Розділення обов'язків команд та запитів): розділяє операції запису та читання в

системі. Команди (зміна даних) та запити (отримання даних) обробляються окремо, що дозволяє оптимізувати шляхи доступу до даних та підлаштовувати логіку обробки під потреби конкретного виду операцій.

7. Distributed Messaging (Розподілене повідомлення): використовується в асинхронній комунікації між сервісами через розподілену систему повідомлень. Це дозволяє знизити залежність між сервісами, полегшує масштабування та забезпечує гнучкість взаємодії.
8. Sequentially Aggregated activities for a long-lived process (SAGA): використовується для керування транзакціями та збереження консистентності даних між багатьма сервісами. Це розбиває довгу транзакцію на кілька кроків, що можуть виконуватись асинхронно. Якщо стається помилка, патерн виконує зворотні дії для скасування вже внесених змін.
9. Strangler Pattern (Патерн задушника): використовується для поетапного перенесення функціональності з монолітного додатку до мікросервісної архітектури. Він передбачає поступове заміщення окремих компонентів або модулів моноліту новими мікросервісами, зберігаючи при цьому інтеграцію з рештою системи.
10. Gateway Aggregation (Агрегація шлюзів): використовується для забезпечення агрегації даних з різних мікросервісів у єдиний API-виклик. Замість того, щоб клієнтам звертатися до кожного сервісу окремо, вони звертаються до гейтвей, що здійснює необхідні виклики до різних сервісів та об'єднує отримані дані перед поверненням відповіді клієнту.

В цілому, архітектурні патерни в мікросервісах допомагають розробникам створити модульну, гнучку та масштабовану систему. Вони надають стандартизований підхід до проектування та розробки, що полегшує роботу команди та забезпечує високу якість результуючої системи.

1.3.1 Event Sourcing та CQRS

Це цікавий підхід, коли для побудови застосунку одночасно використовується два патерни: Event Sourcing та CQRS(Command and Query Responsibility Segregation).

Event sourcing це архітектурний патерн, у якому всі зміни, що вносяться в систему, зберігаються в тій послідовності, в якій вони були викликані. Ці записи є як джерелом для отримання поточного стану, так і audit-log'ом того, що відбувалося в системі. Event sourcing сприяє децентралізованій зміні та читанню даних. Така архітектура добре масштабується та підходить для систем, які вже працюють із подіями або підходять для міграції на таку архітектуру. Патерн може бути використаний для зберігання подій, які відбуваються в окремих мікросервісах, і для підтримки повної історії змін в системі.

Використання event sourcing може допомогти уникнути проблем, що пов'язані з розподіленою системою, такі як конфлікти збереження даних і проблеми з відновленням стану системи після відмови. Крім того, event sourcing дозволяє зберігати повну історію подій, що сталося в системі, і дозволяє легко відновлювати стан системи в будь-який момент часу.

У мікросервісній архітектурі, кожен мікросервіс може мати власний журнал подій, який зберігає послідовність подій, що відбулися в сервісі, але в даній роботі журнал буде спільним для всього застосунку. Події можуть бути записані в цей журнал після кожного дії, яка змінює стан системи. Ці події можуть бути використані для побудови проекцій стану системи, які можуть бути використані для відображення поточного стану системи або для аналізу історії подій.

Крім того, event sourcing може бути використаний для відслідковування подій між мікросервісами. Кожен мікросервіс може відправляти події до інших мікросервісів, щоб сповістити їх про зміни, які відбулися в системі. Це може допомогти уникнути проблем, пов'язаних з асинхронним спілкуванням між мікросервісами і забезпечити послідовність подій в системі.

Узагалі, для мікросервісної архітектури, event sourcing може бути корисним по наступним причинам:

1. Історична стежка подій: забезпечує збереження повної історії всіх подій, що відбуваються в системі. Це дозволяє переглядати і аналізувати всі зміни стану системи в минулому, а також відновлювати стан системи в будь-який момент часу.
2. Отримання поточного стану: шляхом використання проекцій стану на основі журналу подій, можна побудувати поточний стан системи, враховуючи всі події, що відбулися в мікросервісах. Це дозволяє отримати оновлену інформацію про стан системи безпосередньо з подій, а не збереженого стану.
3. Аудит і відладка: надає можливість проводити аудит і відлагоджувати систему, адже всі дії, які змінюють стан системи, фіксуються в журналі подій. Це дає змогу відстежувати причину помилок, аналізувати проблеми та відновлювати стан системи для дослідження аномалій.
4. Підтримка розподіленості: є особливо корисним при розгортанні мікросервісної архітектури на різних вузлах або в різних місцезнаходженнях. Журнал подій може бути розподіленим між сервісами, дозволяючи їм підтримувати послідовність подій та обмінюватись ними для підтримки консистентності.
5. Легкість міграції і реплікації: дозволяє легко мігрувати та реплікувати дані, оскільки потрібно перенести лише журнал подій, а не всі станові дані. Це спрощує розгортання нових мікросервісів та забезпечує більш гнучке управління системою.
6. Розширюваність та гнучкість: сприяє розширенню та модифікації системи, оскільки події можуть бути оброблені різними споживачами, які можуть мати власні правила обробки подій. Це дозволяє додавати нові функціональні можливості до системи, не змінюючи існуючий код мікросервісів.

7. Відновлення та синхронізація: дозволяє легко відновлювати стан системи після відмови або відновлювати взаємодію між мікросервісами після перерви в комунікації. Завдяки журналу подій можна відтворити всі події, які відбулися від певного моменту часу, тим самим забезпечивши консистентність системи.
8. Можливості аналітики: надає можливості для аналізу та використання історичних даних подій для створення аналітичних звітів, прогнозування трендів, виявлення патернів та інших аналітичних завдань. Це може допомогти приймати кращі рішення на основі історичних даних та становити перевагу для бізнесу.

Загалом, використання патерну event sourcing в мікросервісній архітектурі допомагає забезпечити ефективну обробку подій, збереження стану системи та гнучке управління даними. Він дозволяє створити масштабовану та надійну систему, яка легко адаптується до змін та вимог бізнесу.

З іншої сторони, патерн CQRS (Command Query Responsibility Segregation) передбачає розділення відповідальностей для команд (зміна даних) та запитів (отримання даних) в окремі моделі.

Основними принципами патерну в мікросервісах є:

1. Розділення моделей: замість використання однієї моделі для команд та запитів, патерн CQRS пропонує використовувати окремі моделі для кожного. Це дозволяє оптимізувати моделі для конкретних потреб команд та запитів.
2. Оптимізація для швидкості та продуктивності: оскільки команди часто вимагають інших механізмів (напр. валідації), вони можуть бути оптимізовані для швидкості та продуктивності. Запити можуть бути оптимізовані для швидкого доступу до даних без зайвих перевірок та обробки.

3. Масштабованість: розділення моделей також дозволяє масштабувати систему незалежно для обробки команд та запитів. Можна масштабувати та розгортати окремі сервіси, що обробляють команди та запити, залежно від їх потреб та навантаження.
4. Підтримка доменної моделі: за допомогою цього патерну, доменна модель може бути більш експресивною та фокусуватись на реалізації бізнес-правил. Розділення команд та запитів дозволяє моделі бути спрощеною та оптимізованою для своїх конкретних потреб.
5. Оптимізація читання: оскільки запити на читання відбуваються частіше ніж запити на запис, можна оптимізувати окремі сервіси або репліки бази даних для ефективного обробки цих запитів. Запити на читання можуть використовувати спеціалізовані сховища даних, такі як підсистеми кешування або бази даних для читання, для забезпечення швидкого доступу до інформації.

Схематично роботу клієнта по CQRS моделі можна зобразити наступним чином:

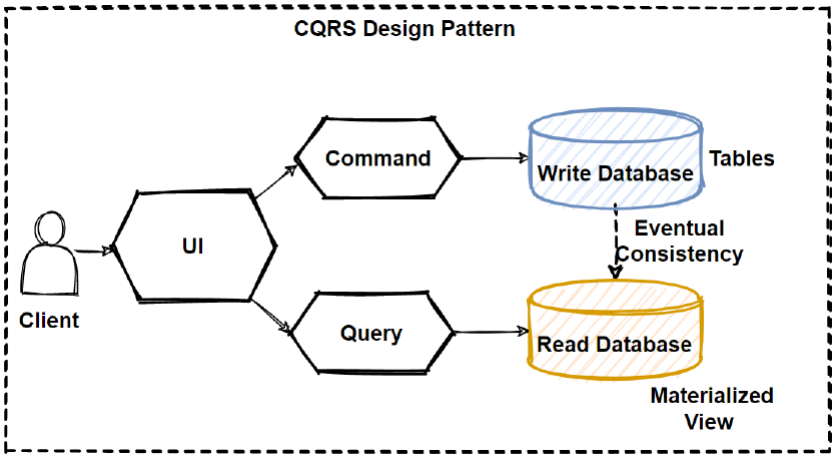


Рисунок 1 Модель патерну CQRS

Використання патерну CQRS разом із підходом Event Sourcing є досить потужною комбінацією для розробки різноманітних додатків і систем. Це цікавий підхід, що має свої переваги. Одне з яких – спрощення розширення

системи у майбутньому, оскільки event log зберігає всі події, їх можна використовувати у зовнішніх системах. Також, досить легко інтегруватися через додавання нових обробників подій.

Повна схема такої комбінації виглядає наступним чином:

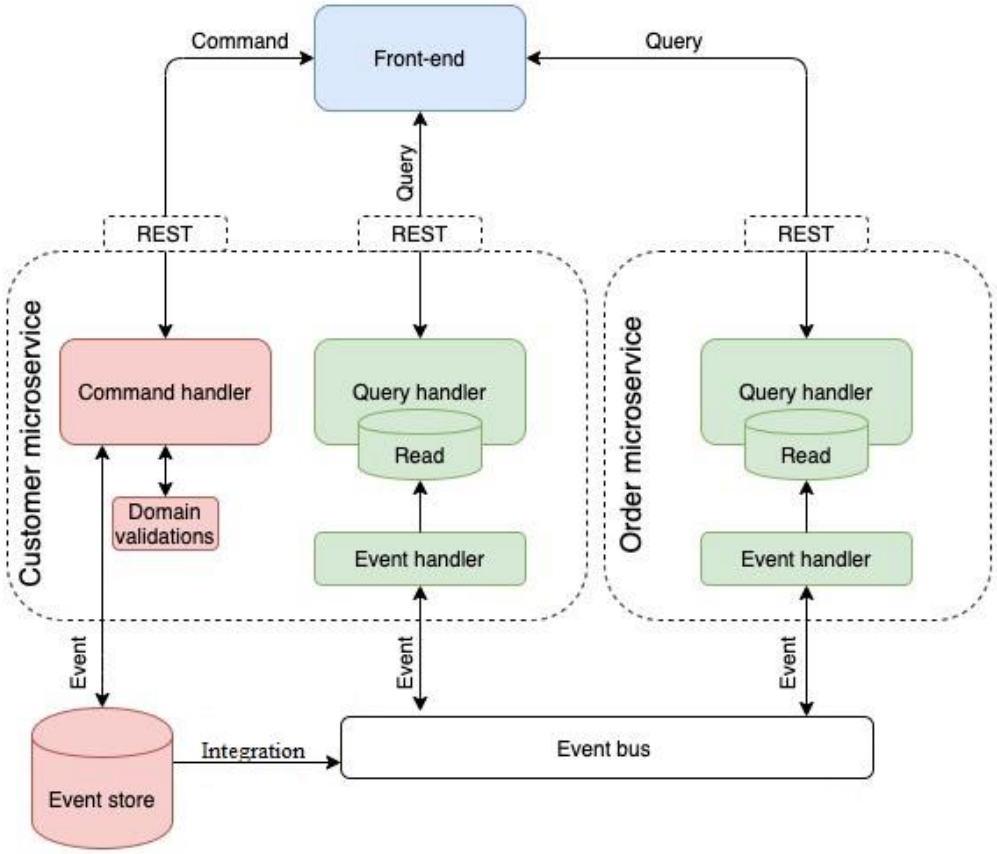


Рисунок 2 Модель патерну CQRS + Event Sourcing

На даній схемі можна побачити, що користувач (UI) відправляє запити на створення/редагування/видалення даних до «Customer» сервісу, запити конвертуються в команди (command), які валідуються та обробляються в хендлерах (Command handler). Хендлери лише створюють події (events) в журналі подій.

На діаграмі можна побачити, що на події підписано два сервіси: «Customer» та «Order». Коли нова подія з’являється в журналі, вона попададає до всіх слухачів (Event handler), які роблять певні дії, наприклад запис в базу даних. Коли ж користувач відправить запит на отримання даних, вони

конвертуватимуться в запити (query), які обробляються в хендлерах (Event handler). Хендлери лише читають дані з баз даних.

Незважаючи на всі вищеперераховані переваги, така комбінація має певні обмеження, пов'язані зі складністю предметної області, вимогами до узгодженості та доступності даних, а також збільшення обсягу даних, що зберігаються, і масштабованість у довгостроковій перспективі. Також важливо приділяти увагу розробникам, які розроблятимуть і підтримуватимуть таку систему протягом усього SDLC. Тому, перш ніж інтегрувати в себе CQRS і Event Sourcing, потрібно визначити, чи дійсно це потрібно, адже принцип KISS ще ніхто не скасовував.

Висновки до розділу 1

В першому розділі було розглянуто мікросервісну архітектуру, її переваги та недоліки, основні способи комунікації між мікросервісами. Також описано основні архітектурні патерни, що використовуються в цій архітектурі, зокрема комбінація патернів Event Sourcing та CQRS, яка буде використана в даній роботі.

РОЗДІЛ 2. СТВОРЕННЯ ТА АНАЛІЗ КЛІЄНТСЬКИХ БІБЛІОТЕК ДЛЯ ЗБЕРЕЖЕНЬ ЗНІМКІВ

Цей розділ буде присвячено базі даних, яка спеціалізується на збереженні подій (events) EventStoreDB, робото з нею а також її перевагам та недолікам. Далі буде розглянуто проблеми що виникають при отриманні даних з EventStoreDB, та бібліотеки для її вирішення. Наостанок буде розроблено власну бібліотеку, та порівняно її з існуючими.

2.1 База даних EventStoreDB

EventStore (або EventStoreDB) є базою даних, спроектованою для зберігання і керування потоком подій (event stream) в розподілених системах. Вона використовується для реалізації патернів Event Sourcing та CQRS (Command Query Responsibility Segregation), про які було розглянуто в теоретичній частині. EventStore надає потужний механізм зберігання та обробки подій, який може бути використаний в різних типах додатків, від мікросервісів до розподілених систем.

Вона забезпечує механізми для збереження, читання та запиту подій, включаючи можливість визначення стрімів (streams), проекцій (projections) і підписок (subscriptions). Вона дозволяє зберігати та розподіляти потоки подій у реальному часі, забезпечуючи гарантію атомарності та послідовності подій.

EventStoreDB є відкритим програмним забезпеченням і має підтримку різних мов програмування та платформ, таких як .NET, Java, Python тощо. Він знаходить своє застосування в багатьох сферах, включаючи фінанси, логістику, громадський транспорт, геймдев та багато інших, де важлива надійність, масштабованість та спроможність до оптимізації роботи з потоками подій.

EventStore пропонує ряд функціональностей, які роблять його потужним інструментом для роботи з потоками подій. Основні з них:

1. Збереження подій: дозволяє зберігати події у послідовних стрімах. Кожен стрім має унікальне ім'я і може містити будь-яку кількість подій. Події

зберігаються в тому ж порядку, в якому вони були записані, і не можуть бути змінені або видалені після того, як вони були записані.

2. Читання подій: надає потужні механізми для читання подій зі стрімів (streams). Можна прочитати всі події зі стріму або вказати певний діапазон чи кількість подій. Крім того, він дозволяє читати події у реальному часі, отримуючи сповіщення про нові події, які додаються до стріму.
3. Проекції (projections): дозволяє визначати проекції, які є похідними від потоків подій. Проекції можуть бути використані для створення агрегованих видів даних або побудови стану системи з подій. Вони оновлюються автоматично, коли нові події додаються до стріму, завдяки цьому легко мати актуальний стан системи в режимі реального часу.
4. Підписки: дозволяє підписатися на потоки подій, щоб отримувати сповіщення про нові події або зміни в існуючих подіях. Це дозволяє миттєво реагувати на зміни в системі в реальному часі і виконувати потрібні дії при надходженні нових подій.
5. Масштабованість: підтримує горизонтальне масштабування, що дозволяє розподіляти навантаження між декількома потоками подій. Це дозволяє розподіленій системі обробляти великі обсяги подій і забезпечувати високу доступність і швидкодію. Можна додавати нові вузли до кластера EventStoreDB для розширення масштабування і обробки більшого навантаження.
6. Транзакційна безпека: гарантує транзакційну безпеку для збереження подій. Кожен запис події в стрім здійснюється в рамках транзакції, що дозволяє забезпечити атомарність та послідовність подій. Це важливо для систем, де консистентність та надійність даних мають велике значення.

Для опису логічної групи подій, що відноситься до певного конкретного об’єкта або сутності в EventStore використовується термін «агрегат» (aggregate). Це основна одиниця зміни стану. Наприклад «UserAggregate» – агрегат що

відповідає користувачу системи, або «StatementAggregate» – агрегат, який відноситься до заяви.

Кожен агрегат має свій унікальний ідентифікатор, за допомогою якого його можна ідентифікувати. Події, пов'язані з конкретним агрегатом, зберігаються та індексуються разом, у вигляді послідовності подій, що називається потоком подій (event stream).

Коли змінюється стан агрегата, нова подія, що відображає цю зміну, додається до потоку подій агрегата. Це забезпечує попередню історію змін стану агрегата та дозволяє відтворювати стан агрегата на будь-який момент часу, використовуючи послідовність подій.

За допомогою агрегатів у EventStoreDB можна забезпечити консистентність та цілісність даних, керувати паралельною обробкою подій та підтримувати оптимістичну блокування для забезпечення цілісності змін стану агрегата.

Важливо пам'ятати, що в EventStoreDB агрегати і їх події зберігаються в незмінній формі, тобто події не можуть бути змінені або видалені після того, як вони були записані. Це дозволяє забезпечити незмінність журналу подій та можливість аудиту та відновлення стану системи на основі цього журналу.

EventStoreDB має різні способи завантаження та використання. Його можна завантажити з офіційного веб-сайту, де доступні пакети для різних операційних систем, таких як Windows, Linux та macOS. Також є готові образи (images) для роботи в контейнеризованому середовищі за допомогою таких інструментів, як Docker або Kubernetes. Однак для продуктового середовища рекомендовано використовувати хмарні ресурси в AWS, Azure або Google Clouds.

Серед користувачів EventStore можна відмітити «SkuVault», «Kallidus», «PwC», «CERN», «Xero», «Arnold Clark», «Wiser Solutions», «Walmart», та багато інших. Українська компанія «Uklon» також активно використовує цю базу даних.

Отже, EventStoreDB є потужним інструментом для роботи з потоками подій і використовується в різних сферах, де важливо зберігати, керувати та аналізувати події у реальному часі. Він дозволяє побудувати подієво-орієнтовані системи, що забезпечують гнучкість, розширюваність та надійність.

2.2 Знімки (Snapshots) в EventStoreDB

EventStoreDB надає можливість зберігати, оновлювати та отримувати дані шляхом запису та читання послідовностей подій. Одна з проблем цієї бази даних полягає в тому, що вона не підтримує традиційні знімки (snapshots), які доступні у деяких інших базах даних, таких як реляційні бази даних.

Замість того, EventStoreDB пропонує механізми для отримання стану агрегату на основі послідовності подій. Події можуть бути записані і відновлені з потоку подій для відтворення стану об'єкта в пам'яті.

Інакше кажучи, в EventStoreDB використовуються самі події як джерело правди та становлять основу для відтворення поточного стану. Тому, для відтворення стану агрегата, потрібно вичитати всі події, що належать до цьому агрегату, та накласти одну на одну в правильній послідовності.

Знімки (snapshots) в EventStoreDB представляють собою стан агрегату в певний момент часу. Замість читання всіх подій з початку стріму (stream), можна скористатися знімком для отримання актуального стану. Зазвичай знімок містить агреговані дані або іншу сумарну інформацію, яка відображає стан стріму подій (агрегату) на певний момент.

Створення знімів в EventStore відбувається за допомогою процесу, відомого як «snapshotting». Щоб створити знімок, потрібно прочитати всі події зі стріму та побудувати знімок на основі цих даних. Це можна зробити використовуючи клієнтські бібліотеки для певної мови програмування.

Після створення знімка, його можна використовувати для прискорення читання даних зі стріму, або ж іншими словами для отримання агрегату.

Замість читання всіх подій з початку, достатньо прочитати знімок та потім прочитати та накласи лише нові події, що не входили в цей знімок.

Основний підхід до реалізації знімків в EventStoreDB включає:

1. Визначення певної точки або умови для створення знімку. Це може бути кількість подій, часовий проміжок або інша логічна умова, яка визначена для певного агрегату.
2. При досягненні цієї точки або виконанні умови створення знімку, можна згенерувати нову подію, яка представлятиме знімок стану агрегату, або краще – записати це в якусь базу даних.
3. При відтворенні стану агрегату, варто спочатку відтворити знімок (якщо такий існує), по версії знімку можна зрозуміти кількість подій що в нього входить, а потім відтворити лише події, які відбулися після знімку. Це дозволить швидше відновити стан агрегату, оскільки не потрібно відтворювати всю історію подій.

Важливо зазначити, що реалізація знімків в EventStoreDB залежить від вимог та особливостей застосунку. Головне налаштувати процес створення знімків так, щоб він відповідав потребам і забезпечував оптимальну продуктивність.

2.3 Клієнтські бібліотеки на .NET для роботи зі знімками (snapshots)

Як уже було сказано, EventStoreDB не має вбудованої підтримки знімків, тому потрібно реалізувати цей механізм самостійно, або ж користуючись готовими клієнтськими бібліотеками.

Для роботи зі знімками на .NET для EventStoreDb знайдено три бібліотеки в NuGet-і (пакетний менеджер, що дозволяє легко встановлювати, оновлювати та видаляти бібліотеки та розширення (пакети) для розробки програмного забезпечення на платформі .NET).

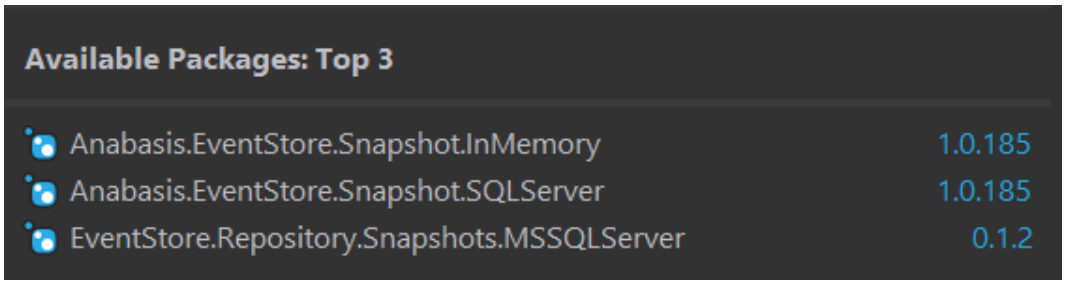


Рисунок 3 Клієнтські бібліотеки на .NET для роботи з EventStoreDb

2.3.1 EventStore.Repository.Snapshots.MSSQLServer

Це пакет NuGet, який надає підтримку для зберігання та відновлення знімків (snapshots) стану агрегатів в базі даних Microsoft SQL Server при роботі з EventStore в .NET-додатках.

Цей пакет є частиною бібліотеки «EventStore.Repository», яка призначена для спрощення роботи з EventStore в сценаріях з використанням підходу Event Sourcing. Вона надає абстракції та реалізації для керування агрегатами, подіями та знімками.

Проблемою цього пакета є сильна зв’язність з бібліотекою «EventStore.Repository». Для роботи з EventStore існує офіційна бібліотека «EventStore.Client» (понад 5 млн завантажень), якою користується більшість розробників, тому використовувати сторонню бібліотеку зовсім недоцільно, а пакет «EventStore.Repository.Snapshots.MSSQLServer» не може бути використано окремо.

Ще одною проблемою є повна відсутність документації. Матеріалу по використанню цієї бібліотеки не було знайдено ні на офіційному веб-сайті бібліотеки, ні на GitHub репозиторії. Спроби інтуїтивно зрозуміти як використовувати, переглянувши вихідний код також не увінчалися успіхом.

Також кількість завантажень бібліотеки (909 разів) свідчить про її не популярність. А остання версія – 0.1.2, про те, що це бета-версія і її не можна використовувати в робочому середовищі. Тому, ця бібліотека не може брати участі в подальшому порівнянні.

2.3.2 Anabasis.EventStore.Snapshot.SQLServer

Ще один пакет NuGet для .NET, що надає можливість зберігання знімків (snapshots) стану агрегатів в базі даних Microsoft SQL Server при роботі з EventStoreDb.

Як і в попередній бібліотеці, документація по використанню відсутністя і на NuGet-сторінці пакету, і в GitHub репозиторії. Але за допомогою unit-тестів, що присутні в репозиторії, бібліотеку було успішно використано та протестовано. Також з кількості завантажень у розмірі 13.5 тис. випливає, що бібліотека відносно популярна, та непогано використовується для роботи зі знімками.

Для роботи з бібліотекою довелось написати певний адаптер AggregateSnapshotDbContext, що наслідує абстрактний клас бібліотеки «Anabasis» BaseAggregateSnapshotDbContext<TAggregateSnapshot>:

```
6 usages  Volodymyr Osadchuk *
public class AggregateSnapshotDbContext : BaseAggregateSnapshotDbContext<AggregateSnapshot>
{
    4 usages  Volodymyr Osadchuk *
    public AggregateSnapshotDbContext(DbContextOptions options)
        : base(options)
    {
    }

    Volodymyr Osadchuk *
    public AggregateSnapshotDbContext(DbContextOptionsBuilder dbContextOptionsBuilder)
        : base(dbContextOptionsBuilder)
    {
    }
}
```

Рисунок 4 Клас AggregateSnapshotDbContext

Окрім відсутності документації, ця бібліотека має ще кілька недоліків:

1. Збереження динамчних об’єктів в реляційну базу даних. Справа в тому, що потрібно зберігати агрегати різних типів, які наперід не відомі, а sql-таблиці для знімків повинні бути створені заздалегідь, тому бібліотека зберігає тіло агрегата у вигляді серіалізованого об’єкту в колонку «SerializedAggregate», що по-перше не читабильно при спробі подивитися в таблицю, а по-друге вимагає серіалізації/десеріалізації на рівні коду.

2. Бібліотека зберігає все в одній таблиці «AggregateSnapshots». Не зважаючи на індекс, коли всі знімки агрегатів різних типів лежать в одній таблиці, це уповільнює запис та читання даних. Можливо, це наслідок того, що в SQL та EntityFramework (бібліотека для роботи з базами даних на .NET) не можна динамічно створювати таблиці. Значно краще було б для кожного агрегату мати свою таблицю.
3. В якості первинного ключа таблиці «AggregateSnapshots» використовується стрічка. Це частоко наслідок попереднього пункту, оскільки знімки різних типів лежать в одній таблиці, потрібно їх якось відрізняти, а ID агрегату вже не достатньо, бо в теорії два різних типи можуть мати однакові ідентифікатари. Тому ключ таблиці має вигляд «[Назва агрегату]-[Ід агрегату]».

2.3.3 Anabasis.EventStore.Snapshot.InMemory

Ще один компонент фреймворку «Anabasis», який забезпечує зберігання знімків (snapshots) в оперативній пам'яті при роботі з EventStore в .NET-додатках. У нього майже 32 тис. завантажень, що робить цей пакет найбільш популярним серед розглянутих.

Завдяки тому, що цей NuGet-пакет працює зі знімками безпосередньо в пам'яті, це значно покращує продуктивність та спрощує процеси відтворення, збереження та роботи з агрегатами. Це основна і, можливо, єдина перевага в такому підході.

Однак, не зважаючи таку перевагу, цей підхід набуває нових недоліків:

1. Обмежена пам’ять мікросервісу. Оперативна пам'ять, виділена на сервіс, обмежена, і збереження в ній знімків агрегатів протягом роботи програми може призвести до вичерпання ресурсів та зниження продуктивності системи.
2. Втрата даних. В оперативній пам'яті знімки зберігаються тільки до зупинки/перезавантаження/оновлення сервісу. Це особливо критично,

якщо на проекті використовується DevOps, і публікація нових змін може відбуватися декілька разів на день. Потрібно буде щоразу відновлювати стан агрегатів з повного потоку подій, що негативно позначиться на продуктивності.

3. Відсутність спільного доступу до даних. Зазвичай у великих системах запущено декілька екземплярів одного сервісу, і завдяки окремому сховищу, коли один мікросервіс зберігає знімок, інші можуть його використовувати. Ця перевага відсутня в даній реалізації.
4. Відсутність зовнішнього доступу до даних. Буває, що для відлагодження та тестування знімків потрібно перевірити, чи вони коректно створюються, але збереження їх у пам'яті унеможливлює цю дію.
5. Неможливість відновлення системи. Ще однією, можливо не самою очевидною перевагою знімків є можливість часткового відновлення даних після непередбачуваного збою системи, але в такій реалізації ця опція неможлива.

2.4 Імплементація бібліотеки на .NET для роботи зі знімками (snapshots)

Вищерозглянуті клієнтські бібліотеки для роботи зі знімками в SQL та оперативній пам’яті мають свої переваги та недоліки. Метою є розробка нової бібліотеки для поєднання найкращих практик для ефективної роботи зі знімками і мінімізувати їх недоліки.

Основна ідея полягає в використанні комбінацію зберігання знімків в базі даних для довгострокового зберігання, а також у пам'яті для підвищення продуктивності та швидкого достуву під час роботи. Такий підхід може поєднати переваги різних реалізацій та надати розширену функціональність для роботи зі знімками.

Додатково, бібліотека може підтримувати механізми версіонування знімків, асинхронного збереження та завантаження знімків, а також може включати надання детальної документації та прикладів використання, що

допоможе розробникам швидко розуміти, як правильно використовувати бібліотеку та впроваджувати її в свої проекти.

Однак, замість використання реляційної бази даних для довгострокового та спільного зберігання знімків, пропонується використати NoSQL базу, наприклад MongoDB.

Таким чином, комбінація зберігання знімків агрегатів у пам’яті та в MongoDB перекриває більшість недоліків існуючих бібліотек, та дає наступні переваги:

1. Висока продуктивність побудови агрегату. Оскільки робота зі знімками вібувається безпосередньо в пам’яті, в першу чергу перевіряється наявність знімку там, що дає можливість не чіпати сторонні сховища. Хоча запис нового знімку потребує збереження і в пам’ять і в MongoDB.
2. Збереження динамчних об’єктів. Завдяки документо-орієнтованій базі даних, можна зберігати об’єкти різних типів не серіалізуючи / десеріалізуючи їх безпосередньо в коді, та роблячи їх читабильнішими при перегляді колекцій.
3. Один агрегат на колекцію. MongoDB дозволяє швидко створювати неіснуючі колекції для знімків нових типів агрегатів. Це дозволяє швидше знаходити потрібний знімок, і трохи пришвидшує вставку документів. Іншою перевагою є те, що тепер ключом таблиці слугує ключ агрегату, а не стрічка як в випадку з MS SQL Server-ом.
4. Відсутність втрати даних. Оскільки окрім пам’яті знімки зберігаються в MongoDB, вони не втрачаються при зупиненні або перезавантаженні сервісу. Після перезапуску при зверненні до агрегату сервіс витягне знімок з бази даних та покладе у пам’ять для швидкого доступу.
5. Спільний доступ до даних. Хоча кожен екземпляр мікросервісу зберігає свої знімки в оперативній пам’яті, MongoDB спільна для всіх них, тому одні сервіси можуть користуватись знімками, зробленими іншими.

Методи роботи зі знімками агрегатів виглядають наступним чином:

```
1+2 usages  Volodymyr Osadchuk *  More...
public async Task<TAggregate> GetByVersionOrLast<TAggregate>(string streamId, long? version = null)
    where TAggregate : Aggregate, new()
{
    if (_allowMemorySnapshots)
    {
        await using var inMemoryContext = new AggregateSnapshotInMemoryContext(_inMemoryContextOptions);

        var inMemorySnapshot = await inMemoryContext.AggregateSnapshots // DbSet<AggregateSnapshotSqlModel>
            .Where(a :AggregateSnapshotSqlModel => a.AggregateKey == Guid.Parse(streamId))
            .Where(a :AggregateSnapshotSqlModel => version == default || a.Version == version) // IQueryable<AggregateSnapshotSq
            .OrderByDescending(a :AggregateSnapshotSqlModel => a.Version) // IOrderedQueryable<AggregateSnapshotSqlModel>
            .FirstOrDefaultAsync(); // Task<AggregateSnapshotSqlModel>
        if (inMemorySnapshot != default)
        {
            var inMemoryAggregate =
                _jsonSerializer.Deserialize(inMemorySnapshot.Payload, typeof(TAggregate)) as TAggregate;
            return inMemoryAggregate;
        }
    }

    var mongoSnapshotsCollection = _mongoDatabase.GetCollection<AggregateSnapshot>(typeof(TAggregate).Name);

    var mongoSnapshot = await mongoSnapshotsCollection // IMongoCollection<AggregateSnapshot>
        .Find( filter: a :AggregateSnapshot => a.AggregateKey == streamId && (version == default || a.Version == version))
        .SortByDescending( field: a :AggregateSnapshot => a.Version) // IOrderedFindFluent<AggregateSnapshot,...>
        .FirstOrDefaultAsync(); // Task<AggregateSnapshot>

    if (mongoSnapshot == default) return default;

    var mongoAggregate = BsonSerializer.Deserialize<TAggregate>(mongoSnapshot.Payload);
    return mongoAggregate;
}
```

Рисунок 5 Метод отримання знімку агрегата за версією, або останньої версії

```
0+2 usages  Volodymyr Osadchuk
public async Task Save<TAggregate>(TAggregate aggregate)
    where TAggregate : Aggregate, new()
{
    var snapshotsCollection = _mongoDatabase.GetCollection<AggregateSnapshot>(typeof(TAggregate).Name);

    var existingSnapshot = await GetByVersionOrLast<TAggregate>(aggregate.Id.ToString(), aggregate.Version);
    if (existingSnapshot != default) return;

    var mongoSnapshot = new AggregateSnapshot
    {
        AggregateKey = aggregate.Id.ToString(),
        Version = aggregate.Version,
        Payload = aggregate.ToBsonDocument()
    };

    await snapshotsCollection.InsertOneAsync(mongoSnapshot);

    if (_allowMemorySnapshots)
    {
        var memorySnapshot = new AggregateSnapshotSqlModel
        {
            AggregateKey = aggregate.Id,
            Version = aggregate.Version,
            Payload = _jsonSerializer.Serialize(aggregate)
        };

        await using var inMemoryContext = new AggregateSnapshotInMemoryContext(_inMemoryContextOptions);
        inMemoryContext.AggregateSnapshots.Add(memorySnapshot);
        await inMemoryContext.SaveChangesAsync();
    }
}
```

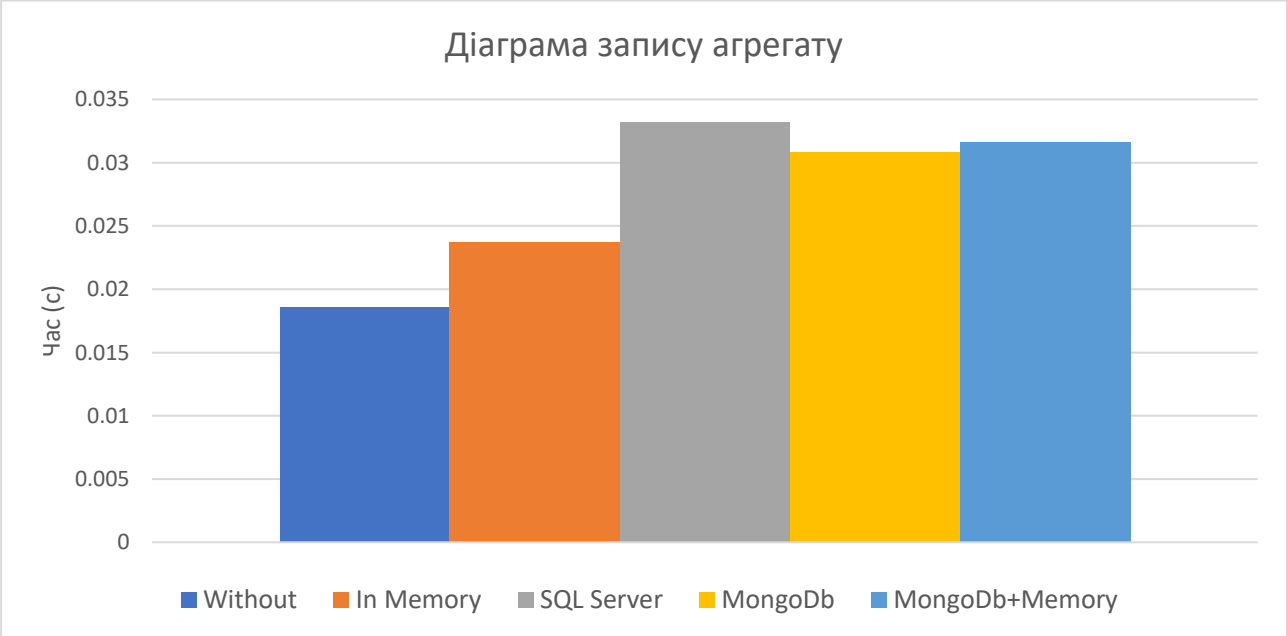
Рисунок 6 Метод збереження нової версії знімку агрегата

Отже, незважаючи на перераховані недоліки такої комбінації MongoDB та оперативна пам’ять, вони зовсім не критичні. Однак, така бібліотека має значно більше переваг та поєднує кращі практики існуючих клієнтських бібліотек, що дозволяє поєднати швидкість та продуктивність оперативної пам'яті з масштабованістю та стійкістю MongoDB.

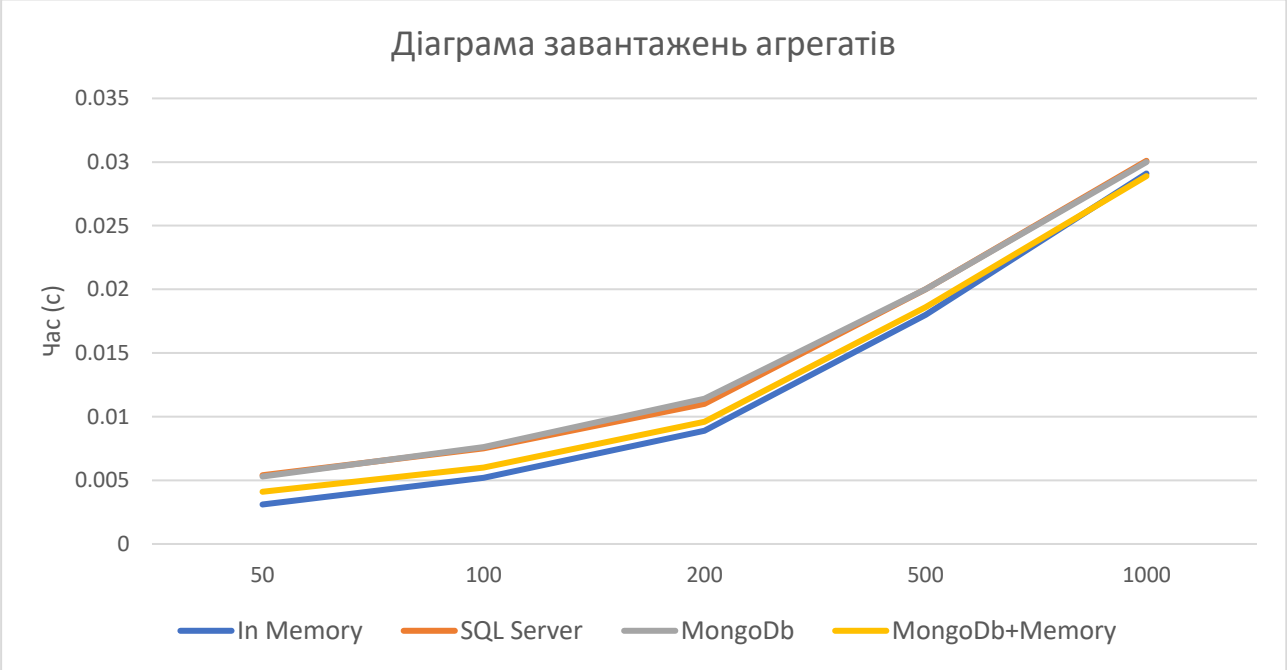
2.5 Порівняння продуктивності бібліотек

Проведено порівняльний аналіз продуктивності клієнтських бібліотек для збереження знімків агрегатів «Anabasis.EventStore.Snapshot.SQLServer» та «Anabasis.EventStore.Snapshot.InMemory» зі створеною бібліотекою, яка зберігає дані в пам’яті та MongoDB (з увімкненою та вимкненою опцією збереження безпосередньо до пам’яті), а також варіант без використання бібліотек та знімків.

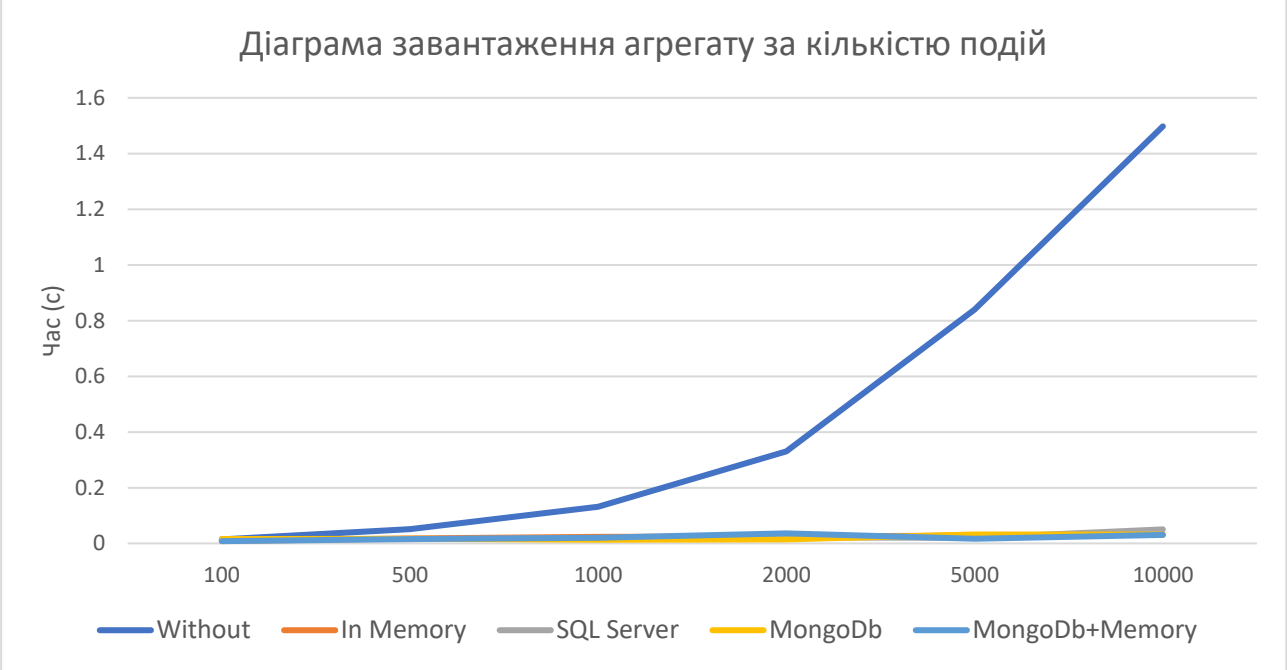
Бібліотеки порівнювались за декількома категоріями: середня швидкість запису та зчитування агрегатів в залежності від частоти збереження знімків та швидкість отримання агрегату з різною кількістю подій: 100, 500, 1000, 2000, 5000 та 10000. Нижче наведені діаграми досліджень:



Діаграма 1 Швидкість запису агрегатів



Діаграма 2 Швидкість зчитування агрегатів



Діаграма 3 Швидкість завантажень агрегатів за кількісю подій

З графіків випливають наступні висновки:

- Середня швидкість запису агрегату без знімків та зі знімками майже не відрізняється. Це означає що наявність бібліотеки майже не впливає на збереження нової версії агрегатів.

- Середня швидкість запису з використанням різних бібліотек займає приблизно однаковий час, однак варто враховувати, що кількість даних в базах даних є приблизно однаковою. Але бібліотека «Anabasis» зберігає всі зімки в одній таблиці, тому якби тестування проводилось на декількох агрегатах, вона б «програла» в швидкості написаній бібліотеці.
- При завантаженні агрегату очевидно швидше працюють бібліотеки, що зберігають знімки безпосередньо в пам’яті, та комбінація пам’ять + MongoDB.
- Завантаження агрегату без використання бібліотек займає значно довше часу, наприклад якщо в агрегаті 10000 подій, це займе приблизно 1.5 сек, в той час як буль-яка бібліотека це зробить за лічені мілісекунди.

Щодо того, як часто робити збереження знімка залежить від конкретного використання та потреб системи. Однак, з графіків видно, що оптимамально це краще робити кожні 100-200 подій в залежності що важливіше – зчитування чи запис.

Висновки до розділу 2

Розглянуто існуючі клієнтські бібліотеки на .NET для роботи зі знімками (snapshots) в EventStoreDB, їхні переваги та недоліки. Створено власну бібліотеку, що зберігає знімки безпосередньо в память та в довгострокову базу MongoDB. Проведено заміри по запису та зчитуванню агрегатів по кожній з бібліотек, та без них, порівняно результати.

РОЗДІЛ 3. ЗАСТОСУВАННЯ EVENTSTORE ТА СТВОРЕНОЇ БІБЛІОТЕКИ ДО ВЕБ-ЗАСТОСУНКУ

В цьому розділі патерн Event Sourcing + CQRS з використанням бази даних EventStoreDB та написаною бібліотекою по збереженню знімків буде застосовано на вже існуючому веб-застосунку з оголошень по прокату паперових книг.

3.1 Існуюча архітектура застосунку та технології

Застосунок написано на .NET та Angular 9. Для зберігання даних використовується база даних MS SQL Server. Взаємодія між сервісами здійснюється за допомогою різних підходів: синхронна комунікація (HTTP), асинхронна комунікація з використанням «MassTransit» та системи повідомлень RabbitMQ, а також взаємодія в реальному часі з використанням бібліотеки «SignalR». Для перевірки стану кожного сервісу та його залежностей використовується бібліотека «AspNetCore.HealthChecks». Збирання і запуск всіх сервісів відбувається в середовищі Docker за допомогою Docker-Compose. Крім цього, в проекті використовуються інші .NET-бібліотеки, такі як «AutoMapper», «Hangfire» і «Refit».

Застосунок складається з семи мікросервісів:

1. Identity. Призначений для реєстрації та перевірки ідентичності користувачів, включаючи як звичайних користувачів, так і адміністраторів. Він використовується в сервісах WebApp та Admin. Аутентифікація здійснюється за допомогою JWT (JSON Web Token). Для реалізації цієї функціональності використовується бібліотека «Microsoft.AspNetCore.Identity». Дані про користувачів зберігаються в MS SQL Server.
2. Publishers. Виконує функції створення, редагування, перегляду та видалення публікацій. Крім того, публікатори можуть переглядати та редагувати свої профілі. Крім основних функцій, сервіс також відповідає

за надсилання сповіщень про створення нової публікації. Для досягнення цього використовується RabbitMQ та бібліотека «MassTransit», за допомогою яких повідомлення відправляються до черги. Крім того, за допомогою бібліотеки «Hangfire», сервіс запускає Cron-джоб, який періодично перевіряє та повторно намагається доставити повідомлення, що не були успішно відправлені з яких-небудь причин. «Hangfire» також використовує базу даних MS SQL Server для збереження проміжних результатів.

3. Statistics. Сервіс, який відстежує створені публікації, та збирає в статистику. Ця статистика доступна як для користувачів (WebApp сервіс), так і для адміністраторів (Admin сервіс). Для отримання інформації про новостворену публікацію, сервіс очікує повідомлення в черзі RabbitMQ. Статистика зберігається в базі даних MS SQL Server.
4. Notifications. Відповідає за надсилання сповіщень про створені публікації. Цей сервіс взаємодіє з WebApp, використовуючи комунікацію в режимі реального часу через WebSocket-и та бібліотеки «SignalR». Наразі сервіс лише сповіщає про створення публікації в консоль, однак його можна розширити для сповіщень на електронну пошту або пуш-повідомлень на мобільні телефони. Подібно до сервісу Statistics, він отримує інформацію про новостворені публікації з черги RabbitMQ за допомогою бібліотеки «MassTransit».
5. WebApp. Це SPA застосунок (Single-page application), розроблений з використанням Angular 9. Він відповідає за відображення інформації та взаємодію з бекенд-сервісами. Застосунок призначений для використання звичайними користувачами. Для авторизації користувачів застосунок використовує Identity сервіс. Для публікації та перегляду оголошень використовується сервіс Publishers. Для відображення статистики щодо опублікованих оголошень використовується сервіс Statistics. Для отримання сповіщень про новостворені оголошення використовується сервіс Notifications. Для запуску та роботи Angular-застосунку в

середовищі Docker спочатку запускається веб-сервер «Nginx», що виконує роль проксі-сервера на операційних системах, які базуються на Unix.

6. Admin. Сервіс з використанням фреймворку ASP.NET Core, який використовується адміністраторами для керування всім застосунком. Цей сервіс включає як фронтенд, так і бекенд-частину, і використовує архітектуру MVC (Model-View-Controller). У сервісі Admin відсутнє власне сховище даних. Замість цього, він використовує Identity, Publishers та Statistics сервіси для отримання та управління інформацією. Комунікація відбувається через REST API за допомогою .NET-бібліотеки «Refit».
7. Watchdog. Цей сервіс здійснює моніторинг роботи інших сервісів та їх залежностькй. Кожен сервіс має API «[service-url]/health», який повертає статус роботи своїх компонентів. Для перевірки роботи MS SQL Server, RabbitMQ та самого сервісу використовується бібліотека «AspNetCore.HealthChecks». Крім того, сервіс періодично кожні 10 секунд перевіряє працездатність інших сервісів та зручно відображає ці дані на сторінці за допомогою бібліотеки «AspNetCore.HealthChecks.UI».

Діаграма існуючої архітектури застосунку виглядає наступним чином:

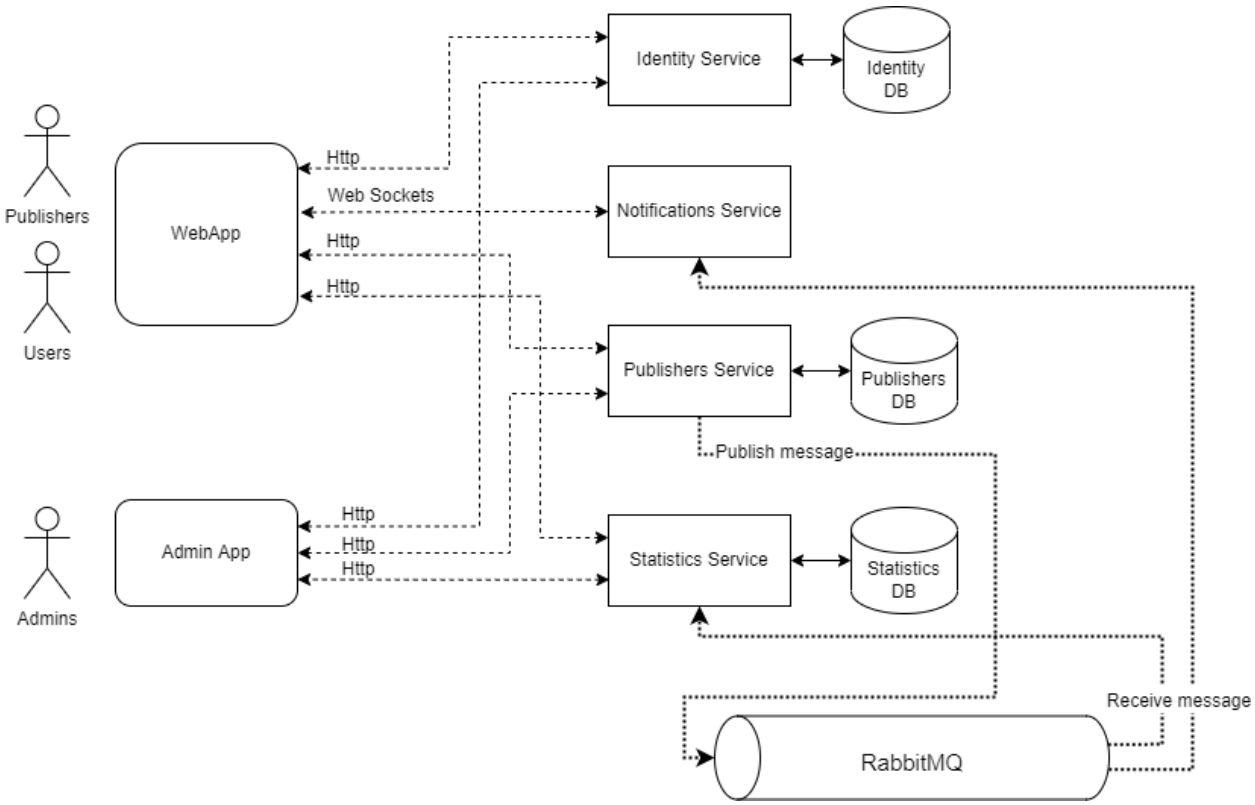


Рисунок 7 Існуюча діаграма застосунку

3.2 Застосування патерну Event Sourcing + CQRS до застосунку

Реалізація вище має певні недоліки, наприклад при створенні нового користувача WebApp відправляє два запити: перший на Identity, другий – на Publishers, хоча достатньо було б наприклад згенерувати подію, на яку підписались ці сервіси.

Отже, визначимо наступні агрегати для збереження в EventStoreDB:

- UserAggregate – агрегат, що відповідає користувачу системи, містить три поля: ім’я (Name), ел. пошта (Email) та телефон (PhoneNumber). А також події:
    - UserCreatedEvent – подія, що створився новий користувач з унікальним ідентифікатором, іменем, ел. поштою та телефоном.

    - UserUpdatedEvent – подія, що користувач змінив ім’я або телефон. (ел. пошту змінювати не можна, тому що вона прив’язана до логіну та входу в систему)
- BookAdAggregate – агрегат, який відповідає одному оголошенню з прокату книги. Він містить повну інформацію про книгу та автора, а також статус (чи активне оголошення) та кількість переглядів. Події, що пов’язані з цим агрегатом:
    - BookAdCreatedEvent – сповіщає про створення нового оголошення з інформацією про книгу та автора.
    - BookAdUpdatedEvent – сповіщає про зміну інформації про книгу чи автора в уже створеному оголошенні.
    - BookAdDeletedEvent – інформує про видалення оголошення публікатором.
    - BookAdAvailabilityChangedEvent – подія про зміну статусу оголошення активна/не активна.
    - BookAdViewedEvent – подія, яка сповіщає що оголошення було переглянуте, містить іфномацію хто переглянув.

Тепер, з використанням EventStoreDB як системи збереження подій, відпадає необхідність використовувати RabbitMQ разом з бібліотекою «MassTransit» як посередниками між сервісами. Адже EventStoreDB самостійно забезпечує можливість публікації та підписки на події.

Замість цього, створено та використано нові мікросервіси, які виконують роль підписників (subscribers) для вищеописаних подій. Ці мікросервіси можуть бути розгорнуті та масштабовані гнучко, залежно від потреб системи. Вони підписуються на потрібні події з EventStoreDB та реагують на них відповідним чином, виконуючи необхідну логіку.

Нижче перераховані нові та змінені сервіси:

1. Identity. Коли користувач реєструється в системі або ж міняє про себе дані, запити надходять в цей сервіс, який генерує події «UserCreatedEvent» та «UserUpdatedEvent».
2. Publishers. Коли користувач системи створює, редагує, видаляє, активує/деактивує свої оголошення, запити відправляються на сервіс Publishers, який генерує відповідні події. Також, коли інші користувачі заходять на чиєсь оголошення з прокату книги, цей сервіс також створює подію «BookAdViewedEvent».
3. Publishers.Projections. Проекція сервісу Publishers, «слухає» події «UserCreatedEvent» та «UserUpdatedEvent» для оновлення інформації про публікаторів оголошень. Також підписаний на події «BookAdCreatedEvent», «BookAdUpdatedEvent», «BookAdDeletedEvent» та «BookAdAvailabilityChangedEvent» для збереження інформації в базу даних MS SQL Server.
4. Statistics. Оскільки сервіс збирає інформацію лише по кількості створених оголошень, він підписаний лише на «BookAdCreatedEvent», і коли подія відбулася, сервіс перевіряє чи це не дублікат та оновлює статистику в базі даних.
5. Notifications. Цей сервіс також інформує лише про те, що створено нове оголошення, тому він «слухає» подію «BookAdCreatedEvent». Коли генерується подія, сервіс сповіщає про це клієнта за допомогою WebSocket-з’єднання. Для забезпечення цього механізму використовується .NET-бібліотека «SignalR», яка дозволяє реалізувати двосторонню комунікацію в режимі реального часу.

Такий підхід дозволяє знизити складність системи, уникнути зайвих шарів посередників та спростити архітектуру, використовуючи EventStoreDB як основне джерело подій та спільний механізм комунікації між сервісами.

3.3 Дігарама застосунку

Оновлена діаграма застосунку виглядає наступним чином:

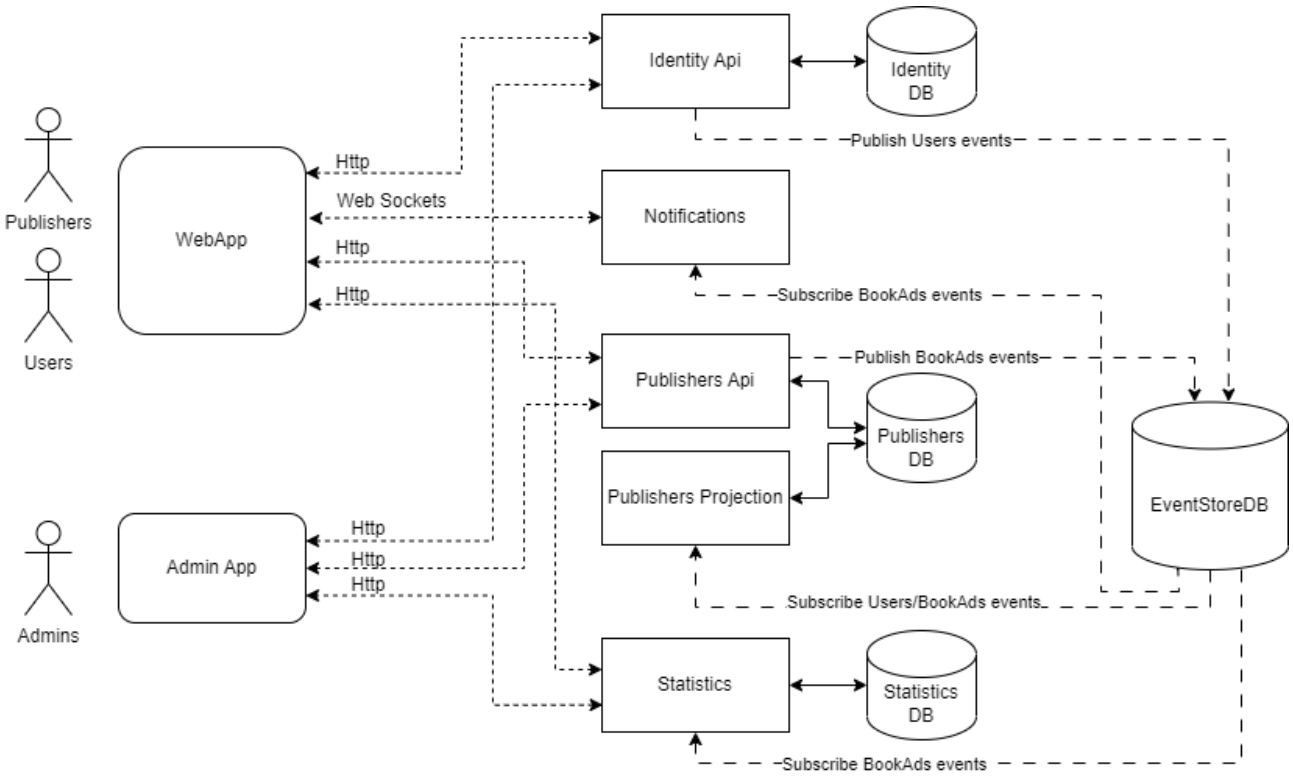


Рисунок 8 Оновлена діаграма застосунку

Висновки до розділу 3

В цьому розділі було розглянуто архітектуру існуючого веб-застосунку з оголошень по прокату паперових книг, переглянуто його мікросервіси. Далі розписано процес застосування патерну Event Sourcing та CQRS разом з базою даних EventStoreDB до цього застосунку. Наведено оновлену діаграму.

ВИСНОВКИ

Мету роботи було досягнуто: розглянуто патерни Event Sourcing та CQRS (Command Query Responsibility Segregation) в контексті мікросервісної архітектури. Розглянуто базу даних для зберігання та керування потоковими подіями EventStoreDB, що допомагає реалізувати даний патерн, а також її основні недоліки.

Розглянуто клієнтські бібліотеки на платформі .NET для збереження знімків (snapshots) від «EventStore.Repository» та «Anabasis». Проаналізовано переваги та недоліки кожної з цих бібліотек. З метою подальшого удосконалення, була розроблена власна бібліотека, яка зберігає знімки агрегатів безпосередньо в пам'яті та в документо-орієнтованій базі даних MongoDB.

Проведено заміри в швидкості роботи бібліотек за різними критеріями. Як результат, створена бібліотека майже не програє в записі агрегату, однак швидше працює витягнення даних, та не втрачаються переваги довгострокового зберігання знімків.

Наостанок, розроблену бібліотеку разом з базою даних EventStoreDB було використано у веб-застосунку для оголошень з прокату паперових книг, який був розроблений з використанням мікросервісної архітектури. Наведено зміни, що вносились до застосунку, та остаточну діаграму.

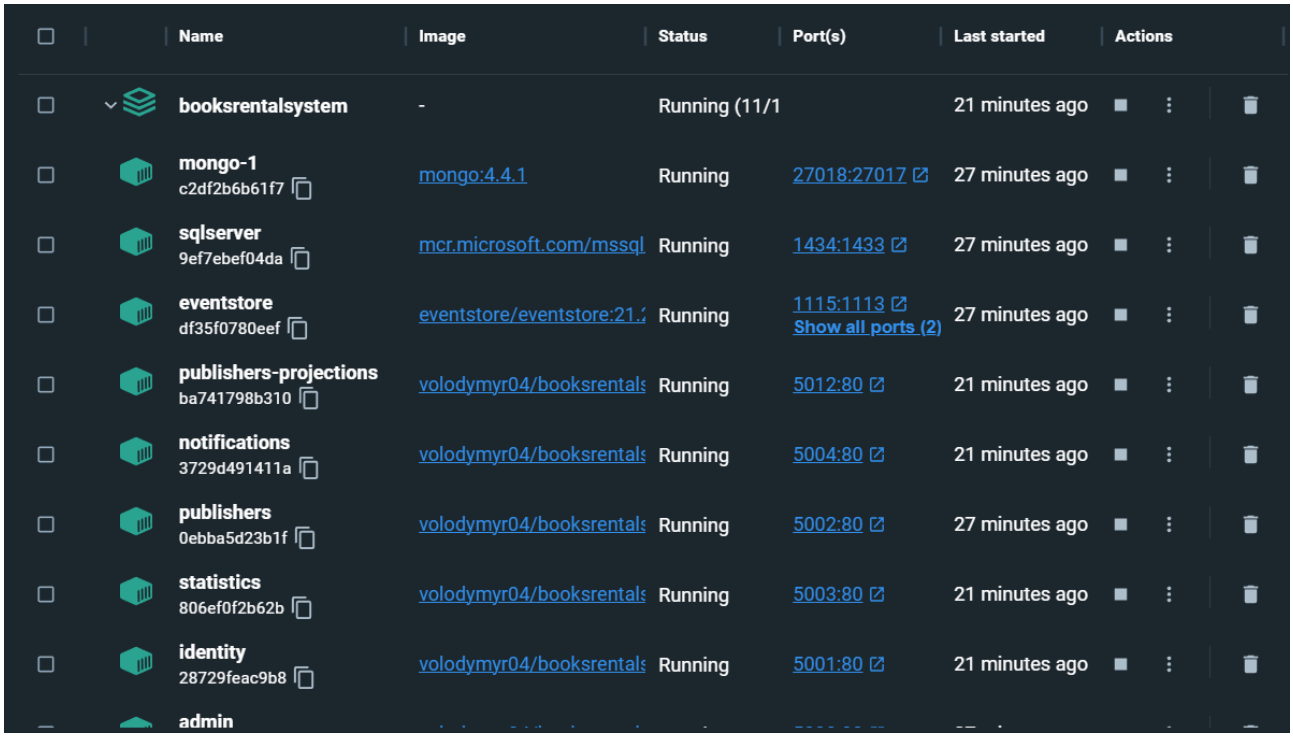
СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Eric Evans, "Domain-Driven Design: Tackling Complexity in the Heart of Software", Addison-Wesley Professional.
2. David Schmitz, "Event Sourcing and CQRS: Applying Domain-Driven Design Patterns", Packt Publishing.
3. Ben Stopford , "Designing Event-Driven Systems: Concepts and Patterns for Stream-Based Applications", O'Reilly Media.
4. Vaughn Vernon, "Implementing Domain-Driven Design", Addison-Wesley Professional.
5. Jonas Bonér, Peter Godman, and Richard Rodger, "Event-Driven Architecture: How Event-Driven Microservices Enable Scalability and Reliability", O'Reilly Media.
6. Greg Young, "Event Sourcing: What it is and why it's awesome", website: eventstore.com/blog, Mar. 21, 2014.
7. Microsoft patterns & practices, "CQRS Journey", Microsoft Corporation, Dec. 2012.
8. Vladimir Khorikov, "Implementing Event Sourcing and CQRS: Lessons Learned", website: enterprisecraftsmanship.com, Feb. 5, 2020.
9. Dennis Doomen, "Event Sourcing and CQRS in Practice", website: infoq.com, Dec. 4, 2017.
10. Rinat Abdullin, "Event Sourcing and CQRS: Two Decades of Practice and Learning", website: slideshare.net, Oct. 18, 2018.
11. Udi Dahan, "CQRS, Race Conditions, and Sagas", website: udidahan.com, Sep. 7, 2010.
12. Mauro Servienti, "Event Sourcing in Practice", Manning Publications, Aug. 2018.
13. Conrad Barski, "CQRS, Event Sourcing, and Stream Processing: An Elixir Case Study", website: lispcast.com, May 17, 2020.

14. Dominic Betts, Julian Dominguez, and Grigori Melnik, "Exploring CQRS and Event Sourcing", Microsoft Corporation, Sep. 2012.
15. Event Store LLP, "EventStoreDB Documentation", website: eventstore.com.
16. Jonathan Worthington, "EventStoreDB in Action", Manning Publications, Feb. 2022.
17. Ben Stopford, "Designing Event-Driven Systems: Concepts and Patterns for Stream-Based Applications", O'Reilly Media.
18. Gary Benner, "EventStoreDB Applied", website: linkedin.com/pulse, Jan. 12, 2021.
19. Mathew McLoughlin, "EventStoreDB: An Introduction", website: dev.to, Apr. 6, 2020.
20. Aaron Palmer, "EventStoreDB: A Distributed Event Store for Modern Applications", website: codeproject.com, Oct. 13, 2020.
21. Tomasz Pluskiewicz, "EventStoreDB as an Event-Sourced Data Store, Journal of Computer Science and Control Systems, vol. 11, no. 1, 2018.

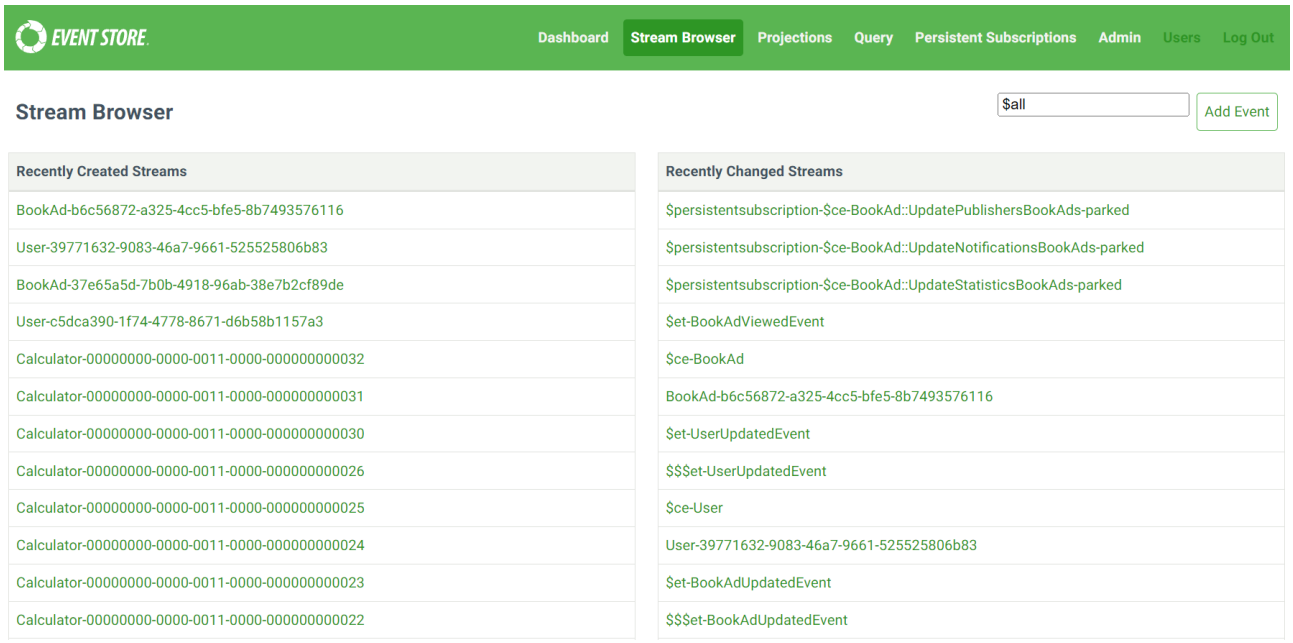
ДОДАТКИ

Запущені сервіси застосунку



Name	Image	Status	Port(s)	Last started	Actions
booksrentalsystem	-	Running (11/1		21 minutes ago	
mongo-1 c2df2b6b61f7	mongo:4.4.1	Running	27018:27017	27 minutes ago	
sqlserver 9ef7ebef04da	mcr.microsoft.com/mssql	Running	1434:1433	27 minutes ago	
eventstore df35f0780eef	eventstore/eventstore:21.1	Running	1115:1113 Show all ports (2)	27 minutes ago	
publishers-projections ba741798b310	volodymyr04/booksrentals	Running	5012:80	21 minutes ago	
notifications 3729d491411a	volodymyr04/booksrentals	Running	5004:80	21 minutes ago	
publishers 0ebba5d23b1f	volodymyr04/booksrentals	Running	5002:80	27 minutes ago	
statistics 806ef0f2b62b	volodymyr04/booksrentals	Running	5003:80	21 minutes ago	
identity 28729feac9b8	volodymyr04/booksrentals	Running	5001:80	21 minutes ago	
admin					

Перегляд новостворених агрегатів застосунку у веб-інтерфейсі EventStore



EVENT STORE

Dashboard | Stream Browser | Projections | Query | Persistent Subscriptions | Admin | Users | Log Out

Stream Browser

$all | Add Event

Recently Created Streams	Recently Changed Streams
BookAd-b6c56872-a325-4cc5-bfe5-8b7493576116	$persistentsubscription-$ce-BookAd::UpdatePublishersBookAds-parked
User-39771632-9083-46a7-9661-525525806b83	$persistentsubscription-$ce-BookAd::UpdateNotificationsBookAds-parked
BookAd-37e65a5d-7b0b-4918-96ab-38e7b2cf89de	$persistentsubscription-$ce-BookAd::UpdateStatisticsBookAds-parked
User-c5dca390-1f74-4778-8671-d6b58b1157a3	$et-BookAdViewedEvent
Calculator-00000000-0000-0011-0000-000000000032	$ce-BookAd
Calculator-00000000-0000-0011-0000-000000000031	BookAd-b6c56872-a325-4cc5-bfe5-8b7493576116
Calculator-00000000-0000-0011-0000-000000000030	$et-UserUpdatedEvent
Calculator-00000000-0000-0011-0000-000000000026	$$$et-UserUpdatedEvent
Calculator-00000000-0000-0011-0000-000000000025	$ce-User
Calculator-00000000-0000-0011-0000-000000000024	User-39771632-9083-46a7-9661-525525806b83
Calculator-00000000-0000-0011-0000-000000000023	$et-BookAdUpdatedEvent
Calculator-00000000-0000-0011-0000-000000000022	$$$et-BookAdUpdatedEvent

Перегляд подій в агрегаті через веб-інтерфейс EventStore

Event Stream 'BookAd-b6c56872-a325-4cc5-bfe5-8b7493576116'

Pause | Edit ACL | Add Event | Delete | Query | Back

self | first | previous | metadata

Event #	Name	Type	Created Date	
15	15@BookAd-b6c56872-a325-4cc5-bfe5-8b7493576116	BookAdViewedEvent	2023-05-27 10:59:19	JSON
14	14@BookAd-b6c56872-a325-4cc5-bfe5-8b7493576116	BookAdViewedEvent	2023-05-27 10:59:17	JSON
13	13@BookAd-b6c56872-a325-4cc5-bfe5-8b7493576116	BookAdViewedEvent	2023-05-27 10:58:57	JSON
12	12@BookAd-b6c56872-a325-4cc5-bfe5-8b7493576116	BookAdUpdatedEvent	2023-05-27 10:54:26	JSON
11	11@BookAd-b6c56872-a325-4cc5-bfe5-8b7493576116	BookAdViewedEvent	2023-05-27 10:54:19	JSON
10	10@BookAd-b6c56872-a325-4cc5-bfe5-8b7493576116	BookAdAvailabilityChangedEvent	2023-05-27 10:54:13	JSON
9	9@BookAd-b6c56872-a325-4cc5-bfe5-8b7493576116	BookAdAvailabilityChangedEvent	2023-05-27 10:53:47	JSON
8	8@BookAd-b6c56872-a325-4cc5-bfe5-8b7493576116	BookAdAvailabilityChangedEvent	2023-05-27 10:53:44	JSON
7	7@BookAd-b6c56872-a325-4cc5-bfe5-8b7493576116	BookAdAvailabilityChangedEvent	2023-05-27 10:53:42	JSON
6	6@BookAd-b6c56872-a325-4cc5-bfe5-8b7493576116	BookAdAvailabilityChangedEvent	2023-05-27 10:53:40	JSON
5	5@BookAd-b6c56872-a325-4cc5-bfe5-8b7493576116	BookAdAvailabilityChangedEvent	2023-05-27 10:53:38	JSON
4	4@BookAd-b6c56872-a325-4cc5-bfe5-8b7493576116	BookAdAvailabilityChangedEvent	2023-05-27 10:53:37	JSON
3	3@BookAd-b6c56872-a325-4cc5-bfe5-8b7493576116	BookAdAvailabilityChangedEvent	2023-05-27 10:53:36	JSON

Перегляд створених підписок

Stream/Group(s)	Rate (messages/s)	Messages			Connections	Status # of msgs / estimated time to catchup in seconds	
		Known	*Current*	*In Flight*			
$ce-BookAd -	0	228	0	0	0	228 / 0	n/a
UpdatePublishersBookAds	0	76	0	0	0	76 / 0	Edit Delete Detail Replay Parked Messages View Parked Messages
UpdateNotificationsBookAds	0	76	0	0	0	76 / 0	Edit Delete Detail Replay Parked Messages View Parked Messages
UpdateStatisticsBookAds	0	76	0	0	0	76 / 0	Edit Delete Detail Replay Parked Messages View Parked Messages
$ce-User -	0	4	0	0	0	4 / 0	n/a
UpdatePublishersUsers	0	4	0	0	0	4 / 0	Edit Delete Detail Replay Parked Messages View Parked Messages