МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ КОМП'ЮТЕРНИХ НАУК ТА ТЕНХНОЛОГІЙ

Кафедра Комп'ютерних інформаційних технологій

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач кафедри
Аліна САВЧЕНКО
«_____»_____2023 р.

КВАЛІФІКАЦІЙНА РОБОТА

(ДИПЛОМНА РОБОТА, ПОЯСНЮВАЛЬНА ЗАПИСКА)

ВИПУСКНИЦІ ОСВІТНЬОГО СТУПЕНЯ

"МАГІСТРА"

ЗА ОСВІТНЬО-ПРОФЕСІЙНОЮ ПРОГРАМОЮ "ІНФОРМАЦІЙНІ УПРАВЛЯЮЧІ СИСТЕМИ ТА ТЕХНОЛОГІЇ"

Тема: «Універсальний веб-сервіс керування сповіщеннями»

Виконала: Лебеденко Ганна Юріївна

Керівник: к.т.н., доцент кафедри КІТ Райчев Ігор Едуардович

Нормоконтролер Ігор РАЙЧЕВ

Київ – 2023

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Факультет *Комп'ютерних наук та технологій*.

Кафедра *Комп'ютерних інформаційних* технологій.

Галузь знань, спеціальність, освітньо-професійна програма: 12 "Інформаційні технології", 122 "Комп'ютерні науки", "Інформаційні управляючі системи та технології".

ЗАТВЕРДЖУЮ

Завідувач кафедри

Аліна САВЧЕНКО

"___"_____ 2023 р.

ЗАВДАННЯ
на виконання кваліфікаційної роботи студентки

Лебеденко Ганни Юріївни

(прізвище, ім'я, по батькові)

1.Тема роботи: «Уніерсальний веб-сервіс кераування сповіщеннями», затверджена наказом ректора від "29" вересня 2023 р. за № 1976/ст.

2. Термін виконання роботи: : з 02 жовтня 2023 р. по 31 грудня 2023 р.

3. Вихідні дані до роботи: методи та засоби проектування та реалізація програмної інтеграційної системи для автоматизації робочого процесу на основі сповіщень.

4. Зміст пояснювальної записки (перелік питань, що підлягають розробці): існуючі системи для інтеграції та автоматизації робочого процесу, важливість уніфікованої системи сповіщень в робочому процесі, складність проектування системи, яка інтегрується з багатьма API, мікросервісна архітектура, монолітна архітектура, event driven development, publish/subscribe патерн,CQRS(Command and Query Responsibility Segregation), Event Sourcing,Розподілені сховища даних Apache Kafka..

5. Перелік обов'язкового графічного матеріалу: інформативні рисунки, графічні скріншоти роботи системи, UMl.-діаграми

6. Календарний план-графік

№ з/п	Завдання	Термін виконання	Підпис керівника
1.	Пошук і дослідження наукових джерел.	06.09.2023 – 08.09.2023	
2.	Розроблення та затвердження календарного плану виконання дипломної роботи.	09.09.2023 – 10.09.2023	
3.	Проведення консультацій з науковим керівником.	11.09.2023 – 12.09.2023	
4.	Написання Розділу 1. Аналіз предметної області.	13.09.2023 –27.09.2023	
5.	Написання Розділу 2. Огляд архітектурних рішень проекту	28.09.2023– 16.10.2023	
6.	Написання Розділу 3. Проектування веб-сервісу	17.10.2023 – 31.10.2023	
7.	Оформлення пояснювальної записки дипломної роботи.	01.11.2023 – 07.11.2023	
8.	Написання, друк та підписання Рецензії у рецензента та Відгуку керівника у встановленому порядку.	08.11.2023 – 11.11.2023	
9.	Створення Презентації та доповіді.	12.11.2023 – 14.11.2023	
10.	Підготовка до захисту та попередній захист дипломної роботи на випусковій кафедрі.	15.11.2023 – 05.12.2023	

7. Дата видачі завдання: «02» жовтня 2023 р.

Керівник дипломної роботи _____ Ігор РАЙЧЕВ
(підпис керівника) (П.І.Б.)

Завдання прийняв до виконання _____ Лебеденко ГАННА
(підпис випускника) (П.І.Б.)

РЕФЕРАТ

Пояснювальна записка до дипломної роботи «Універсальний веб-сервіс керування сповіщеннями» викладена на 84 сторінках і містить 26 рисунків. Список бібліографічних посилань складається з 7 найменувань.

ІНТЕГРАЦІЯ З API, EDD, МІКРОСЕРВІСНА АРХІТЕКТУРА, МОНОЛІТНА АРХІТЕКТУРА, РОЗПОДІЛЕНІ СХОВИЩА ДАНИХ, APACHE KAFKA, CQRS, EVENT SOURCING

Об’єкт дослідження – процес створення і огляд архітектурних рішень для універсальної системи керування сповіщеннями, шляхом інтеграції з кількома API.

Предмет дослідження – універсальний веб-сервіс керування сповіщеннями.

Мета роботи. Проектування веб-сервісу керування сповіщеннями, шляхом інтеграції з декількома API на основі мікросервісної архітектури та розподіленого сховища даних.

Актуальність дипломної роботи полягає в необхідності створення більш надійної, точної та ефективної інтеграційної системи для поєднання багатьох ресурсів в одну єдину платформу.

У роботі висвітлено:

- існуючи рішення для автоматизації робочого процесу
- особливості створення системи, орієнтованої на керування сповіщеннями із залученнями сучасних архітектурних підходів та технологій
- особливості створення програмного забезпечення роботи з польотними даними з залученням сучасних технологій та мов програмування;
- функціонування веб-сервісу для керування сповіщеннями

Подальший розвиток розробленої інформаційної системи можливий у напрямку створення багатьох інтеграцій між різними платформами для забезпечення більшої інформативності системи та покращенню користувацього інтерфейсу.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ

CQRS	Command query responsibility segregation
VM	Virtual Machine
CI/CD	Continuous Integration/ Continuous Delivery
JSON	JavaScript Object Notation
XML	Extensible Markup Language
API	application programming interface
EDD	Event Driven Development
QoS	Quality of service
БД	База даних

ЗМІСТ

ВСТУП

У сучасному світі, коли інформація стає важливим ресурсом і бушує як потік надходження сповіщень з різних джерел та платформ, ефективне управління цими сповіщеннями стає актуальною та необхідною задачею. Під час нашої цифрової ери, багато користувачів мають велику кількість облікових записів на різних веб-платформах та сервісах, таких як Google Calendar, Microsoft Outlook, Jira, Trello, Telegram, Gmail.

Проект передбачає створення інтегрованої платформи, яка дозволить користувачам об’єднувати, упорядковувати та керувати сповіщеннями зі всіх їхніх облікових записів і джерел.

Постійне зростання кількості веб-програм і платформ, які користуються користувачі, робить актуальним питання ефективного керування сповіщеннями з цих джерел. Користувачі потребують зручний інструмент, який допоможе їм ефективно взаємодіяти з інформацією та не втрачати час на перевірку різних платформ. Такий проект є актуальним у сучасному світі, де інформація є ключовим ресурсом.

В сучасному бізнес-середовищі автоматизація робочих процесів є ключовим елементом для підвищення ефективності та конкурентоспроможності підприємств. Інтенсивний темп технологічного розвитку, зростання обсягів даних та динаміка бізнес-операцій ставлять перед організаціями виклики, які можна ефективно вирішити за допомогою автоматизації.

1. Збільшення обсягів даних: Спостерігається експоненційний ріст обсягів даних, які обробляються організаціями. Переважна більшість компаній працює з великою кількістю документації, електронних листів, сповіщень та інших даних. Ефективне їх управління та обробка стають необхідністю.

2. Роздрібленість джерел інформації: Компанії користуються різними програмами та платформами для ведення своєї діяльності, такими як Google Calendar, Microsoft Outlook, Jira, Trello, Telegram, Gmail та інші. В результаті

виникає необхідність об'єднати ці різнорідні джерела інформації для полегшення робочого процесу.

3. Втрата часу та ресурсів: Ручна обробка інформації та ручна взаємодії з різними системами вимагають значних зусиль та часу від співробітників. Це може вести до затримок у виконанні завдань та підвищення ймовірності помилок.

4. Брак інтеграції: Часто програмні продукти та платформи працюють в ізоляції один від одного, що створює проблеми інтеграції та обміну даними. Це ускладнює забезпечення єдиної інформаційної панелі для користувачів.

Автоматизація робочих процесів стає необхідністю для підприємств, що прагнуть залишатися конкурентоспроможними. Розробка веб-додатку для об'єднання та управління сповіщеннями з різних джерел буде актуальним та ефективним рішенням для вирішення вказаних проблем.

РОЗДІЛ 1

АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1. Огляд поняття інтеграції

У контексті розробки програмного забезпечення інтеграція стосується процесу поєднання різних компонентів або систем для спільної роботи як єдиного цілого. Мета інтеграції полягає в тому, щоб різноманітні елементи програмного забезпечення працювали злагоджено, забезпечуючи обмін даними та функціями між ними. Інтеграція має вирішальне значення для створення надійних, ефективних і взаємопов’язаних програмних рішень. Вона передбачає об’єднання незалежних систем, модулів або компонентів а також обміни даними, для функціонування єдиної уніфікованої системи. Це може включати поєднання програмного забезпечення, розробленого різними командами, програм сторонніх розробників або різних служб, передачу інформації між базами даних, API або іншими джерелами даних, щоб гарантувати, що кожен компонент має доступ до необхідних даних. Ключовим аспектом інтеграції є сумісність, що забезпечує безперебійну роботу різних компонентів програмного забезпечення. Вона передбачає визначення стандартів і протоколів для зв’язку, форматів даних та інтерфейсів для забезпечення сумісності.

Проблема безперебійної роботи на різних платформах, операційних системах і середовищах часто вирішується за допомогою інтеграції. Це може включати адаптацію коду та інтерфейсів для забезпечення узгодженої поведінки.

Інтеграція може відбуватися в режимі реального часу, коли системи спілкуються й обмінюються даними миттєво, або за допомогою пакетної обробки, коли дані збираються й обробляються періодично.

Інтеграція може відбуватись за допомогою поєднання декількох API. API відіграють вирішальну роль в інтеграції, забезпечуючи стандартизований спосіб обміну даними для різних програмних компонентів. API визначають методи та протоколи взаємодії, що дозволяє розробникам отримувати доступ до функціональних можливостей певного програмного компонента.

Проміжне програмне забезпечення, таке як брокери повідомлень або інтеграційні платформи, можна використовувати для полегшення зв’язку та обміну даними між різними системами. Вони діють як посередники, які допомагають впоратися зі складнощами інтеграції.

По суті, інтеграція програмного забезпечення полягає у створенні цілісної та сумісної екосистеми компонентів програмного забезпечення, що дозволяє їм ефективно та результативно працювати разом. Це може включати вирішення технічних проблем, визначення протоколів зв’язку та забезпечення відповідності інтегрованої системи загальним цілям і вимогам програмного рішення

Важливість безперебійного зв’язку та обміну даними між різними програмами в розробці програмного забезпечення важко переоцінити. Ключові моменти, які підкреслюють важливість безперебійного спілкування та обміну даними:

1. Оптимізовані робочі процеси:

Ефективний обмін даними забезпечує спрощений робочий процес завдяки виключенню ручного втручання та повторного введення даних. Програми можуть без проблем обмінюватися інформацією, зменшуючи надмірність і мінімізуючи ризик помилок, які можуть виникнути під час передачі даних вручну.

2. Прийняття рішень у реальному часі:

Миттєвий зв'язок і обмін даними підтримують прийняття рішень у реальному часі. Це особливо важливо в сценаріях, коли необхідна своєчасна інформація, наприклад про фінансові операції, системи моніторингу або реагування на критичні події.

3. Покращена взаємодія з користувачем:

Повна інтеграція сприяє позитивній взаємодії з користувачем. Користувачі отримують переваги від уніфікованого інтерфейсу та злагодженої взаємодії з інтегрованими програмами, що сприяє підвищенню продуктивності.

4. Точність і узгодженість даних:

Ефективний обмін даними забезпечує точність і узгодженість інформації в інтегрованих програмах. Це життєво важливо для підтримки цілісності даних і запобігання розбіжностям, які можуть виникнути, коли інформація відокремлюється або управляється незалежно.

5. Гнучкість і масштабованість:

Безперебійне спілкування сприяє гнучкості та масштабованості в екосистемі програмного забезпечення. Коли з’являються нові додатки або оновлюються існуючі, вони можуть легко інтегруватися з іншими компонентами, адаптуючись до мінливих вимог бізнесу.

1.2. Складнощі створення системи, яка інтегрується з кількома API

Створення системи, яка інтегрується з декількома API, представляє низку проблем, що охоплюють технічні тонкощі до операційних міркувань. Одна з головних проблем полягає в різноманітності API, що відрізняються за форматами, протоколами та версіями.

API розвиваються, створюючи проблеми зі змінами та керування версіями. Підтримка зворотної сумісності та адаптація до нових версій потребують постійних зусиль. Безпека є надзвичайно важливою проблемою, включаючи автентифікацію, авторизацію та шифрування для забезпечення безпечного зв’язку між системою та різноманітними API.

Складнощі відображення та перетворення даних виникають через різні формати, які використовуються API. Забезпечення безперебійного перетворення для запобігання невідповідності даних стає дуже важливим питанням. Управління залежностями є ще одним викликом, оскільки зовнішні зміни API або збої можуть потенційно вплинути на функціональність системи.

Комплексне тестування є важливим, але складним завданням, враховуючи безліч сценаріїв і крайніх випадків для багатьох API. Ефективні процедури тестування важливі для забезпечення надійності системи.

Впровадження механізмів аналітики та звітності допомагає швидко виявляти проблеми та оптимізувати роботу системи.

У міру масштабування системи забезпечення масштабованості в інтеграції API без утворення вузьких місць стає критичним питанням. Вирішення цих проблем потребує стратегічного підходу, який включає надійний архітектурний дизайн, безперервний моніторинг, ефективну комунікацію з постачальниками API та постійне технічне обслуговування для адаптації до змін у системі API.

1.3. Роль Event-Driven-Development у інтеграційній системі сповіщень

Розробка, керована подіями (EDD) — це ключовий підхід до інтеграції, який пропонує гнучкість і ефективність у обробці зв’язку між різними компонентами. У EDD потік системи диктується подіями. Роль EDD в інтеграції можна зрозуміти через її вплив на різні аспекти.

Слабкий зв’язок є відмінною рисою EDD, коли компоненти взаємодіють через події, не знаючи один про одного. Ця незалежність дозволяє вносити зміни або доповнення до системи без порушення існуючих компонентів. Події обробляються асинхронно, підвищуючи швидкість реагування та продуктивність системи. Масштабованість спрощена, оскільки нові компоненти можна плавно додавати, а робочі навантаження розподіляти між кількома примірниками.

Реагування в режимі реального часу досягається завдяки подіям, які викликають негайну реакцію на зміни. EDD добре підходить для архітектур мікросервісів, сприяючи спілкуванню між незалежними службами. Його гнучкість у шаблонах інтеграції вміщує такі сценарії, як публікація-підписка або зв’язок «point-to-point».

Події сприяють журналу аудиту та журналу, забезпечуючи ефективний засіб для відстеження дій системи, налагодження та дотримання нормативних вимог. У системах, що використовують пошук подій, стан виводиться з послідовності подій, що забезпечує точне представлення історії системи.

Інтеграція із зовнішніми системами оптимізована за допомогою подій, що дозволяє ефективно сповіщати про зміни або оновлення. EDD забезпечує динамічну та адаптовану структуру, пропонуючи переваги в модульності, незалежності та швидкості реагування. Вплив парадигми поширюється на масштабованість, обробку в реальному часі, архітектури мікросервісів і надійну обробку помилок, що робить її цінним підходом до створення інтеграцій, які можуть розвиватися відповідно до динамічних вимог сучасних систем.

1.4. Аналіз існуючих рішень

Існують різні сервіси, які пропонують схожі функціональності або спрямовані на об'єднання та управління робочим процесом, завданнями та сповіщеннями з різних джерел. Важливо зазначити, що характеристики та можливості цих сервісів можуть варіюватися. Ось деякі існуючі сервіси, які можуть бути взяті у приклад:

1.4.1. Monday.com

Monday.com — це універсальна та популярна робоча операційна система, яка служить централізованою платформою для команд і організацій для керування проектами, завданнями та робочими процесами. Він забезпечує візуальний та інтуїтивно зрозумілий інтерфейс, що дозволяє командам співпрацювати, відстежувати прогрес і ефективно організовувати роботу. Ось основні аспекти Monday.com:

Користувачі можуть створювати та налаштовувати робочі простори відповідно до своїх конкретних потреб. Ця гнучкість дозволяє адаптувати Monday.com до різних галузей, розмірів команд і вимог проекту.

Monday.com використовує візуальні дошки, часові шкали та діаграми, щоб забезпечити чіткий огляд прогресу та стану проекту. Це візуальне представлення дозволяє членам команди легко зрозуміти поточний стан завдань і проектів.

Платформа полегшує керування завданнями та проектами, надаючи інструменти для створення та призначення завдань, встановлення термінів виконання та відстеження основних етапів. Це дозволяє користувачам ефективно керувати робочими процесами.

Monday.com покращує співпрацю завдяки таким функціям, як коментарі, обмін файлами та співпраця в реальному часі. Члени команди можуть спілкуватися всередині платформи, зменшуючи потребу у зовнішніх інструментах спілкування.

Автоматизації на Monday.com дозволяють користувачам автоматизувати рутинні завдання та процеси. Це може включати призначення завдань, оновлення статусу та сповіщення, оптимізацію робочих процесів і економію часу.

Команди використовують Monday.com для планування та відстеження проектів, розподілу завдань і візуалізації графіків проектів. Візуальні інструменти платформи дозволяють легко контролювати прогрес і коригувати плани за потреби. Також зручним Monday.com є зручним для повсякденного керування завданнями, розподілу обов’язків, встановлення термінів виконання та відстеження виконання завдань. Це сприяє організації та підзвітності всередині команд.

Monday.com полегшує співпрацю, надаючи членам команди централізований простір для спілкування, обміну файлами та співпраці в режимі реального часу. Це мінімізує потребу у фрагментованому спілкуванні на кількох платформах.

Рецепти автоматизації на Monday.com допомагають оптимізувати повторювані завдання, зменшуючи ручні зусилля та підвищуючи ефективність. Це включає автоматизацію оновлення статусу, призначення завдань і сповіщень.

Таким чином, Monday.com — це комплексна робоча операційна система, розроблена для покращення співпраці, організації та ефективності в командах і

організаціях. Його адаптивність, візуальний підхід до управління проектами та можливості автоматизації роблять його популярним вибором для різноманітних випадків використання в різних галузях.

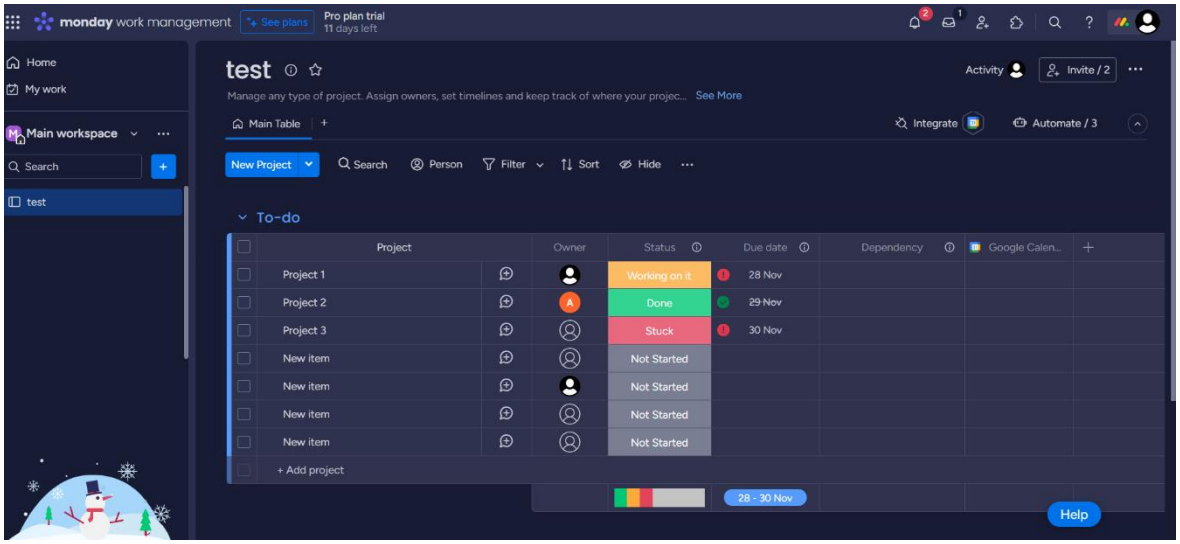

Рис.1.1. Вигляд таблиці з завданнями у застосунку Monday.com

1.4.2. Zapier

Zapier — це провідна платформа автоматизації, яка дозволяє користувачам підключати та автоматизувати робочі процеси в різних програмах без необхідності кодування. Він працює за моделлю «тригер-дія», де подія в одній програмі (тригер) запускає автоматичну дію в іншій програмі. Завдяки підтримці тисяч програм Zapier полегшує створення автоматизованих робочих процесів, відомих як Zaps, для підвищення продуктивності та оптимізації процесів.

Zapier підтримує широкий спектр програм, що дозволяє користувачам легко підключатися та автоматизувати завдання. Його зручний інтерфейс дозволяє користувачам створювати Zaps за допомогою простого процесу налаштування без кодування. Користувачі можуть розробляти складні робочі процеси, що включають в себе складну автоматизацію між багатьма програмами.

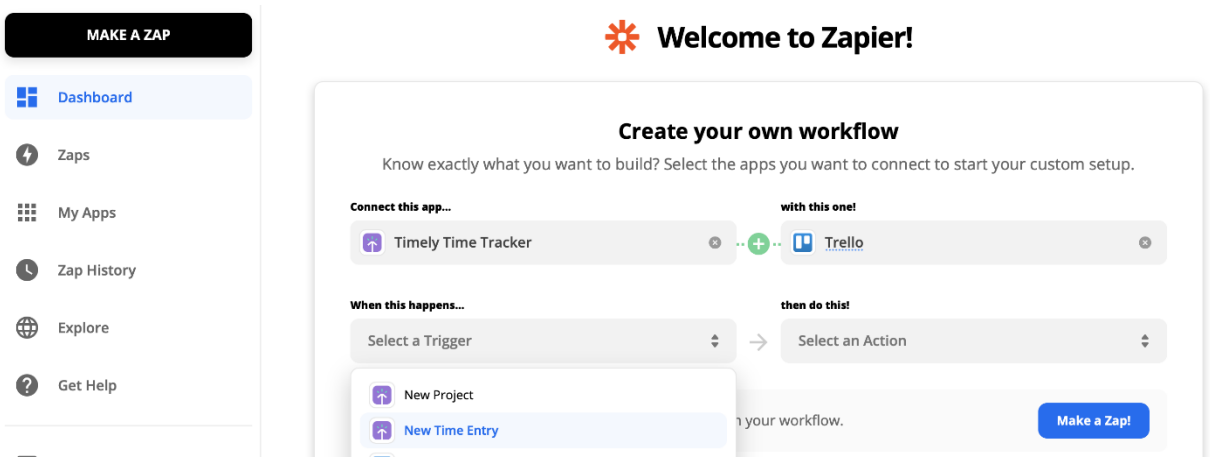

Рис.1.2. Початкова сторінка Zapier

1.4.3. IFTTT

IFTTT, що означає «If This Then That», — це популярна веб-служба, яка дозволяє користувачам створювати автоматизовані робочі процеси, які називаються аплетами, для підключення та інтеграції різних онлайн-платформ, пристроїв і служб. IFTTT дозволяє користувачам автоматизувати завдання та дії, встановлюючи умовні оператори на основі тригерів і дій.

Ключові компоненти та поняття включають в себе:

1) Аплети. Аплети в IFTTT є будівельними блоками автоматизації. Кожен аплет складається з тригера (частина «якщо це») і дії (частина «тоді те»). Коли виникає вказаний тригер, виконується відповідна дія.

2) Тригери - це події або умови, які ініціюють аплет. Вони можуть варіюватися від нових електронних листів або публікацій у соціальних мережах до зміни погодних умов або активності пристрою.

3) Дії – це завдання або операції, які виникають у відповідь на тригери. Вони можуть включати надсилання електронних листів, створення подій у календарі, публікацію в соціальних мережах або взаємодію з пристроями розумного дому.

IFTTT підтримує широкий спектр послуг, включаючи популярні програми, платформи соціальних мереж, розумні домашні пристрої тощо. Користувачі можуть підключати та автоматизувати взаємодію між цими службами.

Цей сервіс має величезну бібліотеку аплетів, створених користувачами, і підтримує інтеграцію з численними популярними програмами та службами. Користувачі можуть досліджувати та відкривати аплети, які відповідають їхнім потребам, або створювати власні аплети, адаптовані до їхніх конкретних вимог.

Він також спрощує автоматизацію, надаючи зручну платформу для підключення та інтеграції різних онлайн-сервісів і пристроїв. Це дає змогу користувачам створювати персоналізовані робочі процеси для оптимізації завдань і підвищення ефективності як в особистому, так і в професійному контексті.

ВИСНОВОК ДО РОЗДІЛУ 1

Безперебійне підключення та співпраця між різними програмними системами стали обов’язковими для компаній, які прагнуть досягти ефективності та інновацій. Важливість інтеграції API дуже вагома, оскільки вона служить основою для забезпечення сумісності та потоку даних між програмами.

Однак шлях створення системи, яка легко інтегрується з кількома API, не позбавлений труднощів. Розробники стикаються зі складнощами, починаючи від різних форматів даних і закінчуючи методами автентифікації, що робить процес складним і вимогливим.

Розробка, орієнтована на сповіщення, стає ключовим вирішенням цих проблем. Дозволяючи системам реагувати в режимі реального часу на певні події та тригери, архітектура, керована подіями, доводить свою важливу роль у створенні динамічних інтеграцій. Такий підхід не тільки підвищує гнучкість додатків, але й гарантує, що вони можуть легко адаптуватися до мінливих вимог.

Крім того, у сфері інтеграційних рішень такі платформи, як Zapier, Monday.com і IFTTT, уже зробили значний вплив, але ще немає такої платформи, яка б могла об’єднати їх функціонал.

РОЗДІЛ 2

ОГЛЯД АРХІТЕКТУРНИХ РІШЕНЬ ПРОЕКТУ

2.1. Монолітна та мікросервісна архітектура

Еволюція архітектур програмного забезпечення спрямована на досягнення кращого розділення обов'язків. Термін "розділення обов'язків" вказує на можливість розкладання та організації систем на логічно зв'язані та слабко зв'язані модулі, які приховують свою реалізацію один від одного та надають послуги через чітко визначені інтерфейси. На сьогодні дві парадигми інженерії програмного забезпечення домінують у розробці сучасних корпоративних додатків: монолітна та архітектура на основі мікросервісів. Перша - це традиційний підхід, в якому додаток створюється з одної кодової бази, що включає кілька служб. Ці служби не є незалежно виконуваними. Вони взаємодіють з кінцевими користувачами та зовнішніми системами через різні інтерфейси, включаючи HTTP(S)/HTML, веб-служби та REST API.

Архітектура мікросервісів декомпозує бізнес-домен на малі, постійно зв'язані контексти, реалізовані автономними, самостійними, слабко зв'язаними та незалежно встановлюваними службами. Одна з перших компаній, яка почала переходили від своєї монолітної архітектури до мікросервісів була компанія Netflix в 2009 році, коли навіть ще не існувало терміну "мікросервіс". Термін був придуманий групою архітекторів програмного забезпечення в 2011 році і офіційно оголошений рік потому. Проте він не став набувати популярності до 2014 року, доки Netflix не поділився своїм досвідом успішного переходу, відкриваючи шлях іншим компаніям.

З того часу мікросервіси отримали значну увагу як в академічному, так і в промисловому середовищі, а поширення технологій контейнерів, таких як Kubernetes та Docker, допомогло цьому новому стилю архітектури отримати ще більшу популярність, особливо в хмарних середовищах . Зрештою, мікросервіси успішно впроваджуються глобальними компаніями, такими як Amazon, eBay, Zalando, Spotify, Uber, Airbnb, LinkedIn, Twitter, Groupon та Coca-Cola.

2.1.1. Монолітна архітектура

Корпоративні додатки часто створюються відповідно до класичної трьохрівневої моделі і, отже, складаються з: коду інтерфейсу користувача (зазвичай сторінки HTML та JavaScript, що працюють у браузері на машині користувача); логіки бізнесу на стороні сервера, яка обробляє HTTP-запити, виконує бізнес-логіку, отримує та оновлює дані з бази даних; і бази даних. Серверний додаток - це моноліт - один логічний виконавчий файл. З точки зору операційної системи монолітний додаток виконується як один процес в середовищі сервера додатків. При розгортанні нової версії додатка вона замінює попередню версію додатка за один крок (наприклад, для розгортання додатка під сервером додатків JEE, потрібно скопіювати один файл EAR/WAR, що містить виконавчий файл додатка, в визначену теку).

Найбільш значущою перевагою монолітної архітектури є її простота - порівняно з розподіленими додатками різних видів, монолітні додатки набагато легше тестувати, розгортати, відлагоджувати та контролювати. Всі дані зберігаються в одній базі даних, і немає потреби в її синхронізації; вся внутрішня комунікація відбувається за допомогою механізмів внутрішнього процесу. Таким чином, взаємодія відбувається швидко і зазвичай не стикається з проблемами, що є типовими для міжпроцесової комунікації (IPC).

Підхід моноліту - це цілком нормальний (і зазвичай він є першим) вибір для побудови додатка - вся логіка обробки запитів виконується в одному процесі. Основні особливості мови, якій віддає перевагу команда розробників, можна

використовувати для структурування програми в класи, функції та просторів імен. Проте, зі зростанням розміру та складності додатка починають виникати проблеми - модифікація вихідного коду додатка стає складнішою, оскільки все більше та більше складного коду починає вести себе непередбачуваним чином. Зміни в одному модулі можуть призвести до непередбачуваної поведінки в інших модулях та каскаду помилок. Сам розмір моноліту призводить до більшого часу запуску, що в свою чергу уповільнює розробку і стає перешкодою для безперервного розгортання. З часом для розробницької команди стає все складніше зберігати зміни, що стосуються певного модуля, щоб вони впливали тільки на цей самий модуль і, в кінцевому підсумку, зберігати модульну структуру додатка. Крім того, зі зростанням розміру додатка збільшується кількість розробників, що часто призводить до нерівномірного використання робочих ресурсів та, в результаті, втрат продуктивності .

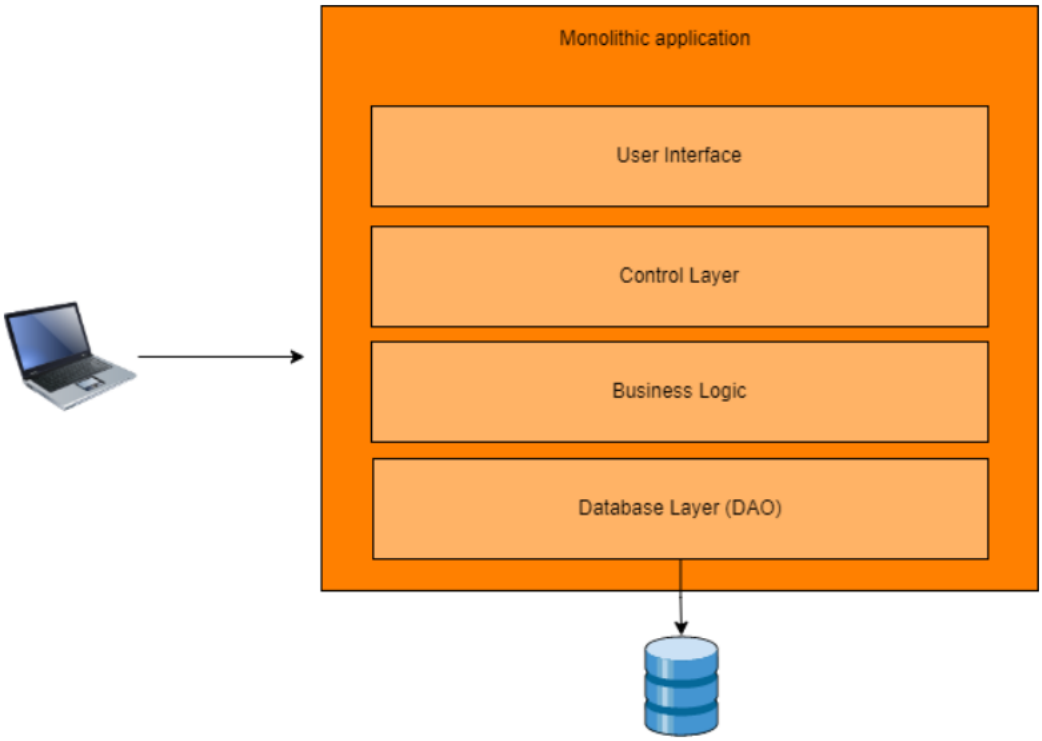

Рис. 2.1. Вигляд монолітної архітектури

2.1.2. Мікросервісна архітектура

Однією з перших спроб описати стиль архітектури мікросервісів була робота Льюїса та Фаулера. У їхньому відомому блозі вони визначили цей новий архітектурний термін як "підхід до розробки одного застосунку у вигляді набору невеликих сервісів, кожен з яких працює у власному процесі та взаємодіє за допомогою легких механізмів, часто через HTTP API ресурсів". Кожен мікросервіс містить свої функції обробки користувача, бізнес-логіку та функції бекенду. Мікросервіс також може включати в себе власний сервіс бази даних (але можливе також спільне використання одного бекенду серед кількох мікросервісів).

Основні принципи цієї архітектури:

• Одна відповідальність на один сервіс - згідно з принципами SOLID, один модуль повинен мати лише одну відповідальність, і в жоден момент часу два

модулі не повинні ділити одну відповідальність або один модуль не може мати більше однієї відповідальності.

• Мікросервіси автономні - вони є самостійними та можуть бути повністю відокремлені для розгортання певного бізнесу. Завдяки своїй автономності вони включають всі залежності, такі як бібліотеки, середовища виконання, веб-сервери та контейнери або віртуальні машини.

• Сервіси публікують кінцеві точки обслуговування як API та абстрагують всі свої деталі виконання. Внутрішня структура - логіка виконання, архітектура та технології (включаючи мову програмування, базу даних і т.д.) - повністю приховані за API.

Важливо відзначити, що парадигма комунікації в мікросервісах значно відрізняється від підходів, заснованих на архітектурі орієнтованій на службу (SOA), таких як Enterprise Service Bus (ESB), які включають складні та "важкі" можливості маршрутизації, фільтрації та трансформації повідомлень. Немає стандарту для механізмів комунікації чи транспортних засобів для мікросервісів. Мікросервіси взаємодіють один з одним, використовуючи добре стандартизовані легкі інтернет-протоколи, такі як HTTP та REST, або протоколи обміну повідомленнями, такі як JMS або AMQP.

Найпривабливіше в мікросервісній архітектурі - розкладання складних застосунків на менші компоненти, які набагато легше розробляти, управляти та підтримувати, ніж великий монолітний застосунок. Поки публічний API не змінюється, внутрішні модифікації одного сервісу простіші, легші та менш витратні, ніж в разі подібної зміни в традиційній моделі. Мікросервіси є автономними та спілкуються через відкриті протоколи, тому їх можна розробляти майже незалежно навіть за використання різних технологій.

Застосунки на основі мікросервісів добре масштабуються горизонтально, не тільки з технічної точки зору, але й з точки зору організації структури команд розробників, які можуть бути меншими та більш гнучкими. Зусилля зі збагачення таких опцій в адаптивне управління бізнес-процесами вже знайшли свій шлях в бізнес-практиці. Крім того, розбиття великих застосунків на окремі мікросервіси

надає наступний рівень незалежності командам, які працюють у agile - середовищі, що підтримує масштабування agile методів. Кожна команда може працювати над різними мікросервісами та розробляти user-story, які впливають лише на їхні мікросервіси. Доки команда не змінює контракти між сервісами, рішення може бути прийняте в межах команди сервісу, а не серед кількох команд, які працюють над великим монолітним застосунком. Таким чином, впровадження архітектури мікросервісів передбачає зменшення потреби в міжкомандній координації, що є серйозним викликом у розробці програмного забезпечення великого масштабу.

Ще однією перевагою мікросервісних застосунків є те, що їхня слабо зчеплена архітектура робить їх більш стійкими до відмов - відмова одного компонента не обов'язково призводить до недоступності всієї системи, оскільки функціонуючі сервіси все ще можуть обслуговувати запити користувачів. Також можливо ідентифікувати критичні бізнес-функціональності та розгортати відповідні мікросервіси в більш резервованому середовищі.

Окрім численних переваг, архітектура мікросервісів має свої недоліки та недоліки, головним чином пов'язані з її розподіленою природою. Розгортання, масштабування та моніторинг багатосервісної системи - це більш складна задача, ніж у випадку монолітного застосунку. З цієї причини різні процедури автоматизації в конвеєрі CI/CD, моніторинг та автоматичне масштабування використовуються в розробці таких застосунків. Щоб повністю скористатися перевагами короткого етапу «від розробки до введення в експлуатацію» також потрібно автоматизувати тестування життєвого циклу, що є більш складною задачею в розподілених середовищах. Інша проблема полягає в розробці засобів керування даними – принципи архітектури мікросервісу стверджують, що максимальна ізоляція сервісу є кращою. Таким чином, в розподіленому застосунку вводяться кілька незалежних систем баз даних, що збільшує складність та зменшує керованість.

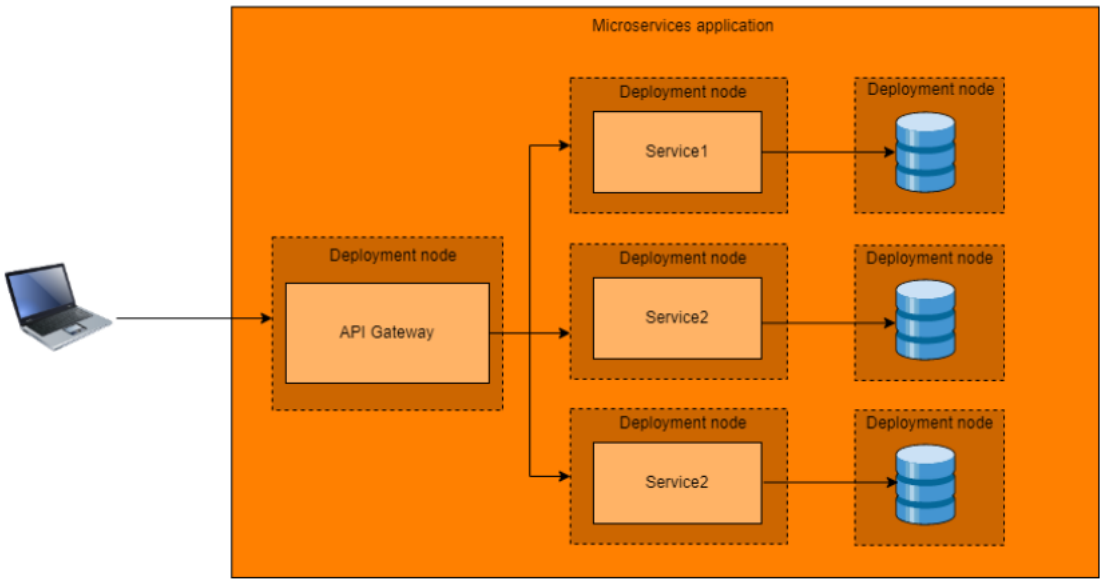

Рис.2.2. Вигляд мікросервісної архітектури.

2.2. Вертикальне та горизонтальне масштабування

Масштабованість - це властивість системи обробляти зростаюче навантаження шляхом додавання ресурсів до системи. Спосіб, яким додаються додаткові ресурси, визначає, який з двох підходів до масштабування використовується - вертикальне масштабування або горизонтальне масштабування. Перше, також відоме як "масштабування вгору", передбачає додавання додаткових ресурсів (ЦП, пам'яті та сховища) до існуючої машини. Це більш прямолінійний підхід, але він обмежений найпотужнішим апаратним забезпеченням, яке є на ринку. Щодо служби Azure App, найпотужніший екземпляр віртуальної машини має всього вісім ядер і 32 ГБ оперативної пам'яті. Крім того, за межами конкретної конфігурації апаратних ресурсів вартість значно збільшується. Важливо відзначити, що у хмарних платформах вертикальне масштабування дозволяє додавати або вилучати віртуальні ресурси до працюючої віртуальної машини (VM), тому воно не призводить до перерви в роботі.

Натомість горизонтальне масштабування, також відоме як "масштабування вбік", передбачає додавання додаткових машин та розподіл робочого навантаження. Це більш складний підхід, оскільки він має вплив на архітектуру додатку, але може надати масштаби, які далеко перевершують ті, що можливі з вертикальним масштабуванням. Горизонтальне масштабування є більш поширеним у додатках на основі мікросервісів, хоча моноліт також може бути масштабований за допомогою балансувальника навантаження, запустивши багато екземплярів. Тим не менш, масштабування монолітного застосунку може бути менш ефективним, оскільки він зазвичай пропонує багато служб, деякі з яких популярніші за інші. Для збільшення доступності монолітного застосунку потрібно реплікувати весь застосунок, що призводить до перенасичення в непопулярних службах, які використовують ресурси сервера, навіть коли вони неактивні, і в результаті призводить до неоптимального використання ресурсів. З іншого боку, для збільшення доступності мікросервісного застосунку отримають додаткові екземпляри лише високопопулярні мікросервіси, які використовують велику кількість ресурсів сервера.

2.3. Способи комунікації між мікросервісами

У розподілених системах, таких як програмне забезпечення на основі мікросервісів, яке складається з десятків і сотень окремих та автономних служб, необхідний спосіб співпраці між цими службами. Ця співпраця та взаємодія можливі завдяки міжпроцесовій комунікації. Міжпроцесова комунікація - це механізм, який надається операційними системами для створення середовища та дозволяє різним процесам (службам) взаємодіяти між собою. Існують різні технології для реалізації міжпроцесової комунікації. Можна використовувати синхронний механізм комунікації, такий як REST або gRPC, або асинхронний стиль комунікації, такий як обмін повідомленнями. Також існують різні формати для передачі даних через ці механізми комунікації, такі як JSON, XML і протокольні буфери.

Розглянемо варіанти комунікації. Перший аспект - взаємодія один до одного та один до багатьох:

- Один до одного - кожен запит клієнта обробляється саме одним сервісом.

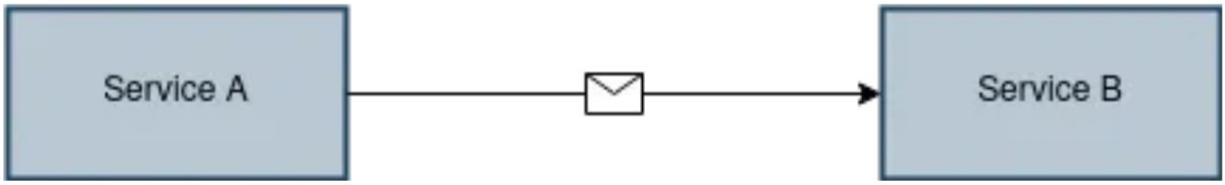

Рис. 2.3. Вигляд взаємодії «один до одного»

- Один до багатьох - кожен запит обробляється кількома сервісами.

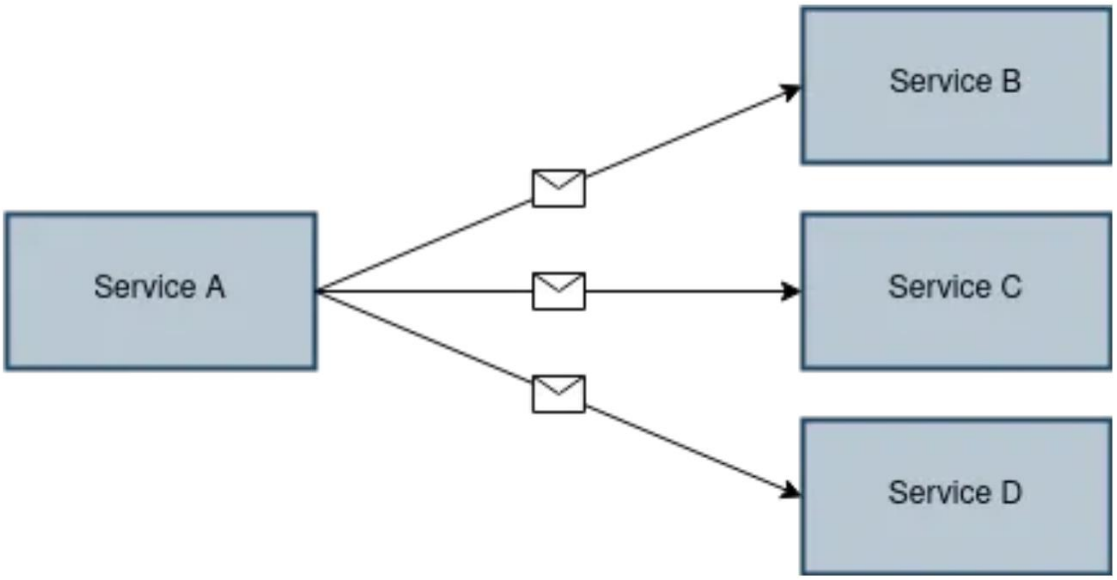

Рис. 2.4. Вигляд взаємодії «один до багатьох»

Другий аспект - взаємодія синхронна та асинхронна:

- Синхронна - клієнт очікує негайної відповіді від сервісу і може блокуватися, поки чекає на неї.

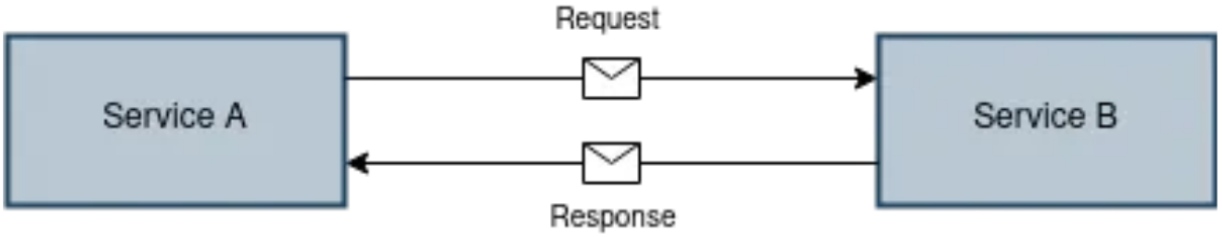

Рис. 2.5. Вигляд синхронної взаємодії

- Асинхронна - клієнт не блокується, і відповідь, якщо вона є, не обов'язково відправляється одразу.

Рис. 2.6. Вигляд асинхронної взаємодії

Типи взаємодії «один до одного»:

- Запит/відповідь: Клієнт сервісу робить запит до сервісу і чекає на відповідь. Клієнт очікує, що відповідь надійде одразу. Він може навіть блокувати свою роботу, доки відповідь не надійде. Це стиль взаємодії, який, як правило, призводить до тісного зв'язку між сервісами.
- Асинхронний запит/відповідь: Клієнт сервісу відправляє запит до сервісу, який відповідає асинхронно. Клієнт не блокується під час очікування, оскільки сервіс може не надсилати відповідь довгий час.

Типи взаємодії «один до багатьох»:

- Publish/subscribe — клієнт публікує сповіщення, яке використовується нульовою або більшою кількістю підписаних служб.
- Publish/async responses — клієнт публікує повідомлення із запитом, а потім чекає певний час на відповіді від підписаних служб.

2.3.1. Синхронне спілкування

У цьому типі взаємодії клієнт надсилає запит із даними до сервісу. Сервіс обробляє вхідний запит на основі визначеної бізнес-логіки, а потім надсилає відповідь клієнту. Деякі клієнти можуть блокувати виконання, чекаючи на відповідь, тоді як інші можуть використовувати неблокуючу та реактивну архітектуру. Як вже зазначалося, існують два механізми синхронного спілкування:

1) REST

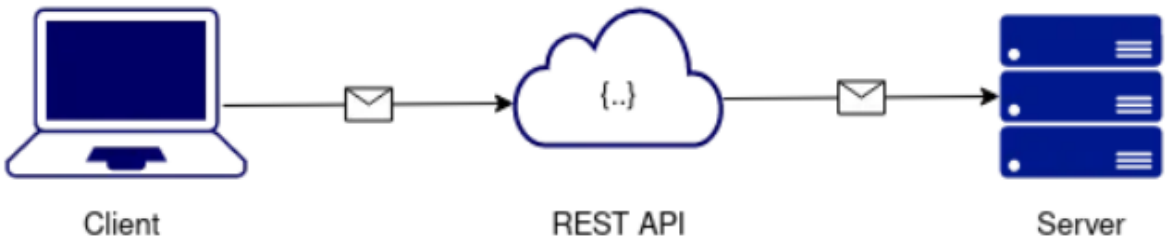

Рис. 2.7. REST як механізм спілкування

На сьогоднішній день зазвичай використовується RESTful стиль для реалізації синхронних систем обміну повідомленнями. REST є міжпроцесовим зв'язком, який використовує протокол HTTP. Ресурс, який представляє бізнес-об’єкт, відіграє ключову роль в цьому механізмі. Можна виконувати різні операції за допомогою HTTP запитів:

- POST-запит для створення нового ресурсу.
- GET-запит для отримання представлення ресурсу.
- PUT/PATCH-запит для оновлення існуючого ресурсу.
- DELETE-запит для видалення існуючого ресурсу.

Однією з труднощів, з якими можна зіткнутися при використанні REST, є обмеження видів запитів HTTP. Іноді важко виконати кілька дій оновлення чи створення для одного й того ж ресурсу у стилі RESTful. Тому в таких ситуаціях можна використовувати gRPC замість REST.

2) gRPC було вперше представлено компанією Google у 2016 році. Це сучасна міжплатформна структура віддаленого виклику процедур з відкритим кодом, яка допомагає визначати операції, які можна викликати віддалено, разом із їхніми параметрами та типами повернення в певному форматі повідомлення, яким є буфери протоколу. Буфери протоколу використовуються для визначення структури повідомлень.

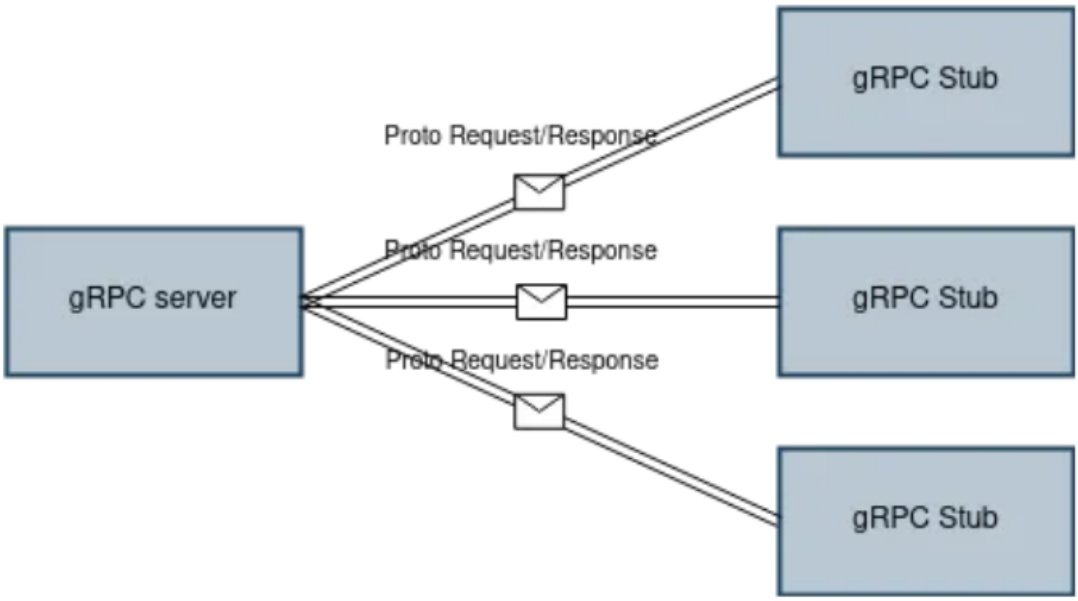

Рис. 2.8. gRPC як механізм спілкування

2.3.2. Асинхронне спілкування

У такому вигляді спілкування клієнт надсилає запит, що містить дані, до служби. Служба обробляє вхідний запит на основі визначеної бізнес-логіки, але існує затримка в часі, перш ніж клієнт отримає відповідь. Системи, які використовують асинхронний обмін повідомленнями як стиль спілкування, зазвичай використовують брокер повідомлень. В асинхронному обміні повідомленнями є дві важливі концепції:

- Повідомлення —складається із заголовка та тіла, що містить дані.
- Канал — обмін повідомленнями через канали.

В асинхронному обміні повідомленнями можна надіслати запит, не очікуючи негайної відповіді. Існує кілька шаблонів, які можна використовувати у системі залежно від потреб цієї системи.

Асинхронний запит/відповідь — у цьому шаблоні обміну повідомленнями спочатку клієнт (виробник) надсилає запит через чергу запитів. Тоді сервер (споживач) обробить вхідне повідомлення асинхронно, а потім надішле відповідь назад через чергу відповідей. Врешті, клієнт отримує відповідь із черги.

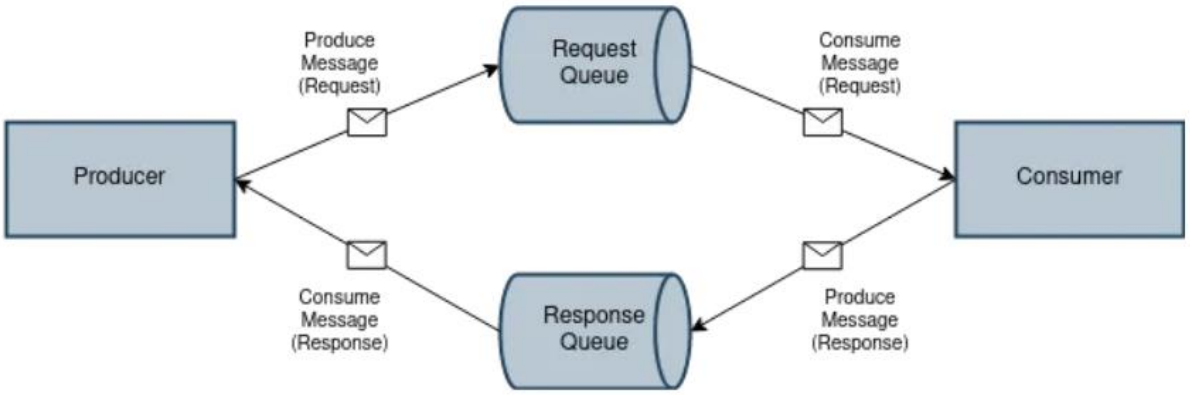

Рис. 2.9. Представлення шаблону асинхронного запиту/відповіді

Publish/Subscribe — у цьому шаблоні обміну повідомленнями виробник публікує повідомлення через брокера повідомлень. Тоді, з іншого боку, може бути кілька споживачів, які можуть споживати опубліковане повідомлення від брокера повідомлень.

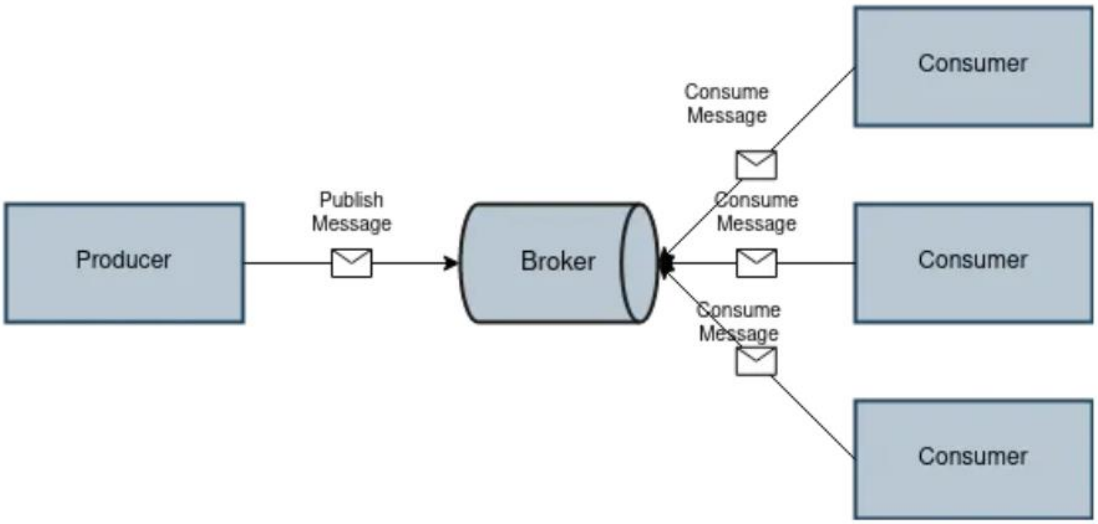

Рис. 2.10. Представлення шаблону Publish/Subscribe

Сповіщення. У цьому зразку взаємодії клієнт ініціює комунікацію, висилаючи повідомлення через чергу чи канал. З іншого боку, сервер споживає та обробляє вхідне повідомлення; проте, на відміну від інших шаблонів, немає явної відповіді, що відправляється назад клієнту. Акцент робиться на односторонній комунікації, де клієнт повідомляє сервер про подію чи дію, не очікуючи безпосередньої відповіді. Цей шаблон часто використовується у випадках, коли клієнт зацікавлений у розсилці інформації чи виклику певних дій на сервері, не потребуючи негайної віддачі.

З метою забезпечення автономії мікросервісів та стабільності системи рекомендується мінімізувати ланцюжки запитів/відповідей. Також, важливо використовувати асинхронні методи інтеграції, використовуючи механізми обміну повідомленнями або подій. Навіть при використанні протоколу HTTP, краще використовувати асинхронні запити, незалежно від ініціювання циклу запиту/відповіді HTTP.

Загалом, синхронна інтеграція не рекомендується для внутрішньої комунікації у мікросервісних системах. Вона порушує автономію, а при збоях одного сервісу загальна продуктивність може значно знизитися. Для інтеграції мікросервісів рекомендується використовувати брокери повідомлень, такі як RabbitMQ чи інші системи черг.

2.3.3. API в архітектурі мікросервісів

API або інтерфейси є центральними аспектами у розробці програмного забезпечення. Додаток складається з модулів. Кожен модуль має інтерфейс, який визначає набір операцій, які клієнти цього модуля можуть викликати. Добре розроблений інтерфейс відкриває користовну функціональність, приховуючи реалізацію. Це дозволяє змінювати реалізацію інтерфейсів модулів без впливу на клієнтів.

У монолітному застосунку інтерфейс зазвичай зазвичай визначається за допомогою конструкції мови програмування, наприклад такої як Java. Інтерфейс Java визначає набір методів, які клієнт може викликати. Реалізаційний клас

прихований від клієнта. Більше того, оскільки Java є статично типізованою мовою, якщо інтерфейс стає несумісним з клієнтом, програма не буде компілюватися. API та інтерфейси так само важливі в архітектурі мікросервісів.

API сервісу - це контракт між сервісом та його клієнтами. API сервісу складається з операцій, які клієнти можуть викликати, та подій, які публікує сервіс. Ці операції мають назву, параметри та тип, який буде повернуто. Проблема полягає в тому, що API служби не визначається за допомогою простої конструкції мови програмування. За визначенням, служба та її клієнти не компілюються разом. Якщо нову версію служби розгорнуто з несумісним API, помилки компіляції не буде. Натомість будуть збої під час виконання

2.3.4. Зміна API та способи версіонування

API невпинно змінюються з часом, оскільки додаються нові функції, змінюються існуючі функції та (можливо) видаляються старі функції. У монолітній програмі відносно просто змінити API та оновити всі виклики. Якщо ви використовуєте статично типізовану мову, компілятор допомагає, надаючи список помилок компіляції. Єдиною проблемою може бути масштаб змін. Щоб змінити широко використовуваний API, може знадобитися багато часу.

У програмі на основі мікросервісів змінити API служби набагато складніше. Клієнтами сервісу є інші сервіси, які часто розробляються іншими командами. Клієнтами можуть бути навіть інші програми за межами організації. Зазвичай неможливо змусити всіх клієнтів оновлюватись разом із службою. Крім того, оскільки сучасні програми зазвичай ніколи не припиняють роботу на технічне обслуговування, розробники, як правило, виконують постійне оновлення служби, тож і стара, і нова версії служби працюватимуть одночасно. Важливо мати стратегію вирішення цих проблем.

Один з способів вирішення таких проблем є специфікація семантичного нумерування версій. Це корисне керівництво для версіонування API та набір правил, які визначають, як використовуються та інкрементуються номери версій. Семантичне нумерування спочатку призначалося для версіонування програмних

пакетів, але його можна використовувати і для версіонування API в розподіленій системі. Специфікація семантичного нумерування (Semvers) вимагає, щоб номер версії складався з трьох частин: MAJOR, MINOR, PATCH. Кожну частину номеру версії слід інкрементувати за такими умовами:

MAJOR: Коли внесено несумісні зміни до API

MINOR: При внесенні сумісних удосконалень до API

PATCH: Коли виправлено помилку зворотної сумісності

Іншим способом є внесення мінорних, сумічних удосконалень. Ідеальною є спроба робити лише сумісні удосконалення. Сумісні зміни - це додаткові зміни до API:

- Додавання необов'язкових атрибутів до запиту
- Додавання атрибутів до відповіді
- Додавання нових операцій

Зробивши лише такі зміни, старі клієнти будуть працювати з новішими службами, за умови дотримання принципу стійкості, який стверджує: "Будьте консервативними в тому, що ви робите, та будьте ліберальними в тому, що приймаєте від інших". Служби повинні надавати значення за замовчуванням для відсутніх атрибутів запиту. Так само клієнти повинні ігнорувати будь-які зайві атрибути відповіді.

2.4. Архітектура мікросервісів на основі подій

Системи, що працюють на основі керованих подій, спроектовані навколо конкретних фрагментів роботи, відомих як події. Події сигналізують про виникнення значущих змін стану, які можуть бути прочитанмим одним чи декількома компонентами системи. Вони стають особливо корисними, коли вони мають впливове значення в межах системи чи поза нею.

Можна представити архітектуру, керовану подіями, з точки зору трьох основних компонентів:

Виробник (продюсер): Цей компонент виявляє події та публікує їх за допомогою брокера подій.

Посередник подій: Цей компонент зберігає події в одній або декількох чергах або журналах, доки споживач не використає ці події.

Один чи кілька споживачів: Ці компоненти підключаються до посередника подій і читають події з різних черг або журналів для подальшої обробки.

Архітектура, заснована на керованих подіях, є альтернативою традиційним стилям архітектури програмного забезпечення типу "запит/відповідь". Традиційні шаблони архітектури зазвичай акцентуються на командах, ініційованих користувачем, як основному механізму обробки в системі. Відмінність між подіями та командами досить невелика. Команда - це прохання виконати дію, яка ще не відбулася. Подія ж вказує на те, що вже відбулося. Важливою особливістю архітектури, заснованої на керованих подіях, є підвищена увага до подій порівняно з командами, що дозволяє створювати системи, в яких компоненти сильно відокремлені один від одного і можуть легше піддаватися змінам з часом.

2.4.1. Виробники та споживачі в архітектурі, керованій подіями

Виробники представляють собою компоненти, які виявляють події та надсилають повідомлення про ці події іншим системам чи компонентам. На відміну від цього, споживачі – це компоненти, які відслідковують конкретні типи подій і виконують визначені дії при отриманні повідомлень про події, які вони відстежують. Виробники та споживачі можуть приймати форму будь-якого типу коду чи обладнання, включаючи сервери баз даних, служби, пристрої та користувачів. Важливо зауважити, що в одній системі один компонент може виступати як виробник і споживач подій, взаємодіючи з іншими частинами системи за допомогою виявлення та реагування на події.

2.4.2. Брокери подій

Посередники подій в архітектурі, орієнтованій на події, відіграють ключову роль у розповсюдженні подій у системі та їх доставці підписаним споживачам. Ці посередники, також відомі як брокери повідомлень, є системами, які отримують події від виробників і передають їх споживачам. Вони забезпечують асинхронний обмін повідомленнями, що дозволяє виробникам і споживачам взаємодіяти, висилати повідомлення один одному без очікування відповіді. Крім того, вони створюють засоби взаємодії між різними системами, дозволяючи обміну даними та повідомленнями.

Брокери подій підтримують як модель push, так і pull. У моделі push, споживачі підписуються на конкретні події, а брокер подій відправляє ці події споживачам, які підписалися на них. У моделі pull, споживачі запитують події від брокера подій. Вибір моделі споживання та конкретного брокера подій залежить від конкретного випадку використання. Apache Kafka, RabbitMQ або AWS EventBridge є прикладами брокерів подій, які широко використовуються для реалізації архітектур, що працюють на основі подій.

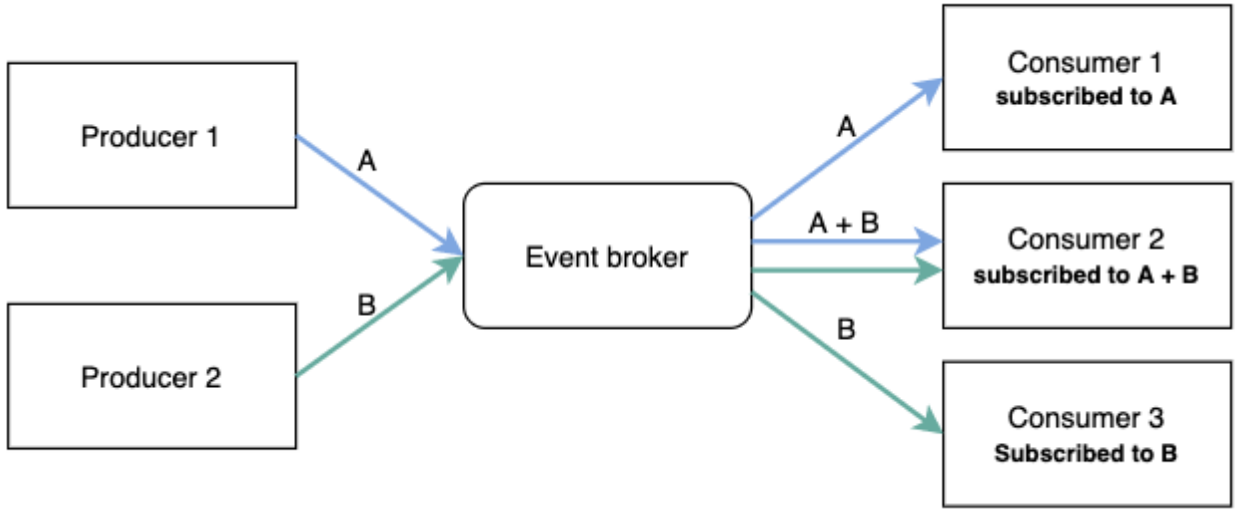

Рис. 2.11. Робота брокеру подій

2.4.3. Переваги архітектури, орієнтованої на події

Зменшена затримка**.** У порівнянні з традиційними методами, такими як синхронний запит/відповідь чи виклики віддалених процедур (RPC), архітектура, керована подіями, дозволяє компонентам не чекати відповіді на свої повідомлення. Асинхронний обмін повідомленнями допомагає зменшити відчутну затримку, пов'язану з передачею даних через мережеві з'єднання.

Слабкі зв’язки. Архітектура, орієнтована на події, спрощує створення систем зі слабкими зв'язками. Системи з низьким зчепленням легше підтримувати, оскільки зміни в коді або функціональності одного компонента ,ймовірно, менше вимагатимуть змін в інших компонентах.

Масштабованість. Архітектура, керована подіями, спрощує масштабування системи, дозволяючи легко додавати додаткових виробників і споживачів, без необхідності переписування коду чи управління складнощами прямого міжпроцесного зв'язку.

Незалежне масштабування та відмовостійкість. Системи, керовані подіями, масштабуються та виходять з ладу незалежно. Наприклад, перевантаженому споживачеві можна виділити більше ресурсів, не впливаючи на інших споживачів чи виробників. Також, через слабк зв’язування, збій одного компонента майже не вплине на інші, підвищуючи відмовостійкість.

Аудит і моніторинг. Деякі реалізації архітектур, орієнтованих на події, особливо підходять для моніторингу та аудиту, оскільки події зазвичай централізовано зберігаються, а не розподіляються по різних базах даних або файлах журналу.

Подійна аналітика. Дані, що містяться в подіях, є оптимальними для сукупного аналізу чи аналітики, допомагаючи відповідати на бізнес-питання та відстежувати ключові показники ефективності.

Обробка даних майже в реальному часі. Архітектура, орієнтована на події, дозволяє системам отримувати та обробляти дані майже в реальному часі, що

особливо корисно для бізнес-аналітики та прийняття кращих бізнес-рішень на основі актуального стану системи.

2.4.4. Недоліки архітектури, орієнтованої на події

Архітектура, орієнтована на події, із численними перевагами, все ж не обійдеться без певних труднощів:

Визначення обсягу інформації у подіях. В ході реалізації архітектури, орієнтованої на події, важливо точно визначити, який обсяг інформації включати в кожну подію. Збільшення інформації може спростити логіку обробки для споживачів, але при цьому може призвести до збільшення обсягу події і вплинути на продуктивність системи. З іншого боку, обмеження обсягу інформації може зменшити витрати на мережу, але споживачам доведеться виконувати додаткові запити для отримання необхідних даних.

Складнощі асинхронної комунікації. Асинхронний обмін повідомленнями викликає складнощі через накладні витрати та розсіяність процесів. У зв'язку з асинхронним типом архітектур, орієнтованих на події, виникає складність визначення зв'язку між двома подіями в часі, що ускладнює пошук та відлагодження систем, побудованих на такій архітектурі.

Гарантії доставки та ідемпотентність. Деякі брокери подій можуть гарантувати доставку подій принаймні один раз, що може призвести до дублювання подій. Забезпечення ідемпотентності обробки подій стає критичним завданням, оскільки події можуть дублюватися. Це вимагає від програм розробити механізми обробки однакових подій лише один раз, навіть якщо вони надходять в систему повторно.

Всі ці виклики потребують ретельного врахування та налагодження для досягнення ефективної та стійкої архітектури, орієнтованої на події.

2.5. Вибір розподіленого сховища подій

Індивідуальні точкові та синхронні засоби комунікації призводять до створення жорстких та статичних додатків, ускладнюючи розробку динамічних великомасштабних програм. Для полегшення завдань розробників додатків, з'єднання між різними сутностями в таких масштабних налаштуваннях ефективніше забезпечувати за допомогою спеціалізованої проміжної інфраструктури, що ґрунтується на правильній схемі зв'язку. Схема взаємодії "publish/subscribe" забезпечує слабко зв'язаний спосіб взаємодії, необхідний в таких великих налаштуваннях.

Apache Kafka та RabbitMQ - це дві відомі системи pub/sub з відкритим вихідним кодом та успішно використовуються протягом майже десяти років у корпоративних компаніях.

Незважаючи на схожі риси, обидві системи мають відмінні історії та цілі розробки, а також унікальні особливості. Наприклад, вони використовують різні архітектурні моделі: у RabbitMQ виробники публікують (пакети) повідомлень із ключем маршрутизації в мережу обмінів, де відбувається вирішення щодо маршрутизації, що завершується в черзі, де споживач може отримати повідомлення через push (переважно) або тяговий механізм. У Kafka виробники публікують (пакети) повідомлень у дисковому журналі, що залежить від теми. Будь-яка кількість споживачів може витягти збережені повідомлення через механізм індексації.

Парадигма publish/subscribe є розподіленою взаємодією, яка ідеально підходить для реалізації масштабованих та слабко зв'язаних систем.

2.5.1. Publish/Subscribe система

Однією з ключових функцій системи публікації/підписки є відокремлення видавців(publishers) та підписників (subscribers). Вважається, що це є, можливо, найбільш фундаментальною характеристикою такої системи. Відокремлення може бути розглянуте в контексті трьох основних аспектів:

Відокремлення сутностей: Видавці та споживачі не повинні мати інформації один про одного. Інфраструктура публікації/підписки виступає посередником у взаємодії.

Часове відокремення: Залучені сторони не обов'язково повинні активно брати участь в процесі взаємодії в один і той же момент часу.

Відокремлення синхронізації: Взаємодія між видавцем або споживачем та інфраструктурою публікації/підписки не вимагає синхронного блокування виробника або виконавчих потоків споживача. Це дозволяє максимально використовувати ресурси процесора як на стороні виробника, так і на стороні споживача, без синхронних блокувань.

Іншою ключовою характеристикою в системах pub/sub є логіка маршрутизації, також відома як модель підписки. Ця логіка вирішує, чи має пакет, що надходить від виробника, потрапити до конкретного споживача. Різні методи визначення подій призвели до виникнення кількох схем підписки, які увібрали в себе гнучкість та продуктивність. Існують два основних типи логіки маршрутизації:

Підписка на основі теми: Цей тип характеризується тим, що видавець статично маркує повідомлення набором тем. Ці теми потім використовуються для ефективної фільтрації, яка визначає, яке повідомлення відправити конкретному споживачеві. Багато систем дозволяють темам містити символи підстановки, а їх імена можуть мати ієрархічну структуру для поліпшення можливостей фільтрації за рахунок більшої навантаженості на процесор.

Підписка на основі вмісту: У цьому випадку виробнику не потрібно явно позначати тему повідомлення для маршрутизації. Умови фільтрації використовують всі поля даних і метаданих повідомлення. Споживачі підписуються на конкретні події, вказуючи фільтри, які визначають обмеження за допомогою властивостей пари ім'я-значення та базових операторів порівняння, що ідентифікують події. Можливість логічного комбінування цих фільтрів (логічне "і", "або" і т.д.) дозволяє створювати складні шаблони підписки.

2.5.2. QoS pub/sub системи

Точність. Концепція точності може бути окреслена трьома фундаментальними принципами: відсутність втрат, відсутність дублювання, відсутність безладу. Виходячи з цих фундаментальних принципів, наступні два критерії є доречними в системах публікації/підписки:

Гарантії постачання:

- Щонайбільше один раз (зазвичай це називається «максимальним зусиллям», що гарантує відсутність дублікатів): у цьому режимі за типових робочих умов пакети доставляються; однак під час системних збоїв може статися втрата пакетів. Спроба перевершити цей рівень надійності незмінно призведе до додаткових системних ресурсів.
- Принаймні один раз (забезпечення відсутності втрат): у цьому режимі система гарантує відсутність втрачених пакетів. Тоді як відновлення після збоїв може призвести до передачі невпорядкованих дублікатів пакетів.
- Тільки один раз (забезпечення відсутності втрат і дублікатів): це вимагає дорогого наскрізного двофазного процесу фіксації.

Гарантії впорядкування:

- Відсутність порядку: відсутність гарантій впорядкування є ідеальним сценарієм для продуктивності.
- Розділене впорядкування: у цьому режимі можна призначити окремий розділ для кожного потоку повідомлень, який вимагає послідовного споживання. Незважаючи на те, що вона більш ресурсомістка, ніж перша група, вона може підтримувати високопродуктивні реалізації.
- Глобальне впорядкування: встановлення гарантії глобального впорядкування спричиняє значні накладні витрати на синхронізацію, потребує значних додаткових ресурсів і потенційно спричиняє серйозне зниження продуктивності.

Надійність. позначає здатність розподіленої системи надавати свої послуги, навіть якщо один або кілька її програмних або апаратних компонентів виходять з ладу.

Це, безперечно, є однією з головних очікуваних переваг розподіленого рішення, заснованого на припущенні, що сервіс, який постраждав внаслідок збою, завжди можна замінити іншим, і це не перешкоджає виконанню запитуваного завдання. Очевидним наслідком є те, що надійність залежить від резервування програмних компонентів, мережевих з'єднань і даних.

Доступність. Доступність — це здатність системи максимізувати час безвідмовної роботи. Важливо зауважити, що це неявно передбачає, що система є надійною: можна виявити збої та розпочати налагодження.

Транзакційність. У системах обміну повідомленнями транзакції використовуються для групування повідомлень в атомарні одиниці: або надіслано (отримано) повну послідовність повідомлень, або жодне з них. Наприклад, виробник, який публікує кілька семантично пов'язаних повідомлень, може не хотіти, щоб споживачі бачили частковий (непослідовний) ланцюг повідомлень, якщо не вдалось прочитати один з них.

Подібним чином критично важлива програма може захотіти отримати одне або кілька повідомлень, обробити їх і лише потім зафіксувати транзакцію. Якщо у споживача виникає помилка перед фіксацією цієї транзакції, усі повідомлення залишаються доступними для повторної обробки після налагодження помилки.

Масштабованість. Поняття масштабованості означає здатність системи постійно розвиватися, щоб підтримувати зростаючу кількість завдань. У випадку pub/sub систем масштабованість може брати участь в різних аспектах, наприклад: споживачі/виробники, теми та повідомлення.

2.5.3. Існуючі реалізації. Apache Kafka

Велика кількість фреймворків та бібліотек може бути класифіковані як ті, що мають функціоналість розсилки pub/sub. Один з підходів до класифікації полягає в розташуванні їх на спектрі складності, що починається з легких систем

з меншою кількістю функцій і закінчується складними системами, які пропонують багатий набір можливостей. На легкій стороні спектра ми маємо ZeroMQ, Finagle, Apache Kafka та інші. Важчі приклади включають реалізації Java Message Service (JMS), такі як ActiveMQ, JBOSS Messaging, Glassfish і інші. AMQP 0.9, популярний та стандартизований протокол pub/sub, має кілька реалізацій, таких як RabbitMQ, Qpid, Hornet та інші. Ще складнішими та функціонально багатшими є розподілені фреймворки RPC, які включають в себе pub/sub, наприклад, MuleESB, Apache ServiceMix, JBossESB та інші.

Apache Kafka спочатку був створений LinkedIn як централізована платформа обробки подій, яка замінила різноманітні системи інтеграції по типу «point-to-point». Команда Kafka спочатку досліджувала кілька альтернатив, зокрема ActiveMQ - популярну систему обміну повідомленнями на основі JMS. Однак під час виробничих випробувань виникли дві значущі проблеми:

1) якщо черга перевищила обсяг, який можна було зберігати в пам'яті, продуктивність значно погіршувалася

2) наявність кількох споживачів вимагала дублювання даних для кожного споживача у власній черзі.

Проблема полягала в тому, що системи обміну повідомленнями спрямовані на роботу в умовах низької затримки, а не на високовитратне масштабування, яке було потрібне в LinkedIn. В результаті вони вирішили побудувати власну інфраструктуру для забезпечення ефективної надійності, роботи з великими чергами споживачів, підтримки кількох споживачів з низькими накладними витратами і явної підтримки розподіленого споживання, при цьому зберігаючи чистий абстракційний підхід до обміну повідомленнями в реальному часі, який властивий системам обміну повідомленнями.

Результатом цього процесу стала масштабована система обміну повідомленнями «постачальник-споживач» ("producer-consumer"), побудована на основі розподіленого логу фіксацій (commit-log). Велика пропускна здатність є однією з переваг дизайну систем агрегації логів порівняно з більшістю систем

обміну повідомленнями. Дані записуються в набір лог файлів без негайного виведення на диск, що дозволяє дуже ефективне використання I/O операцій.

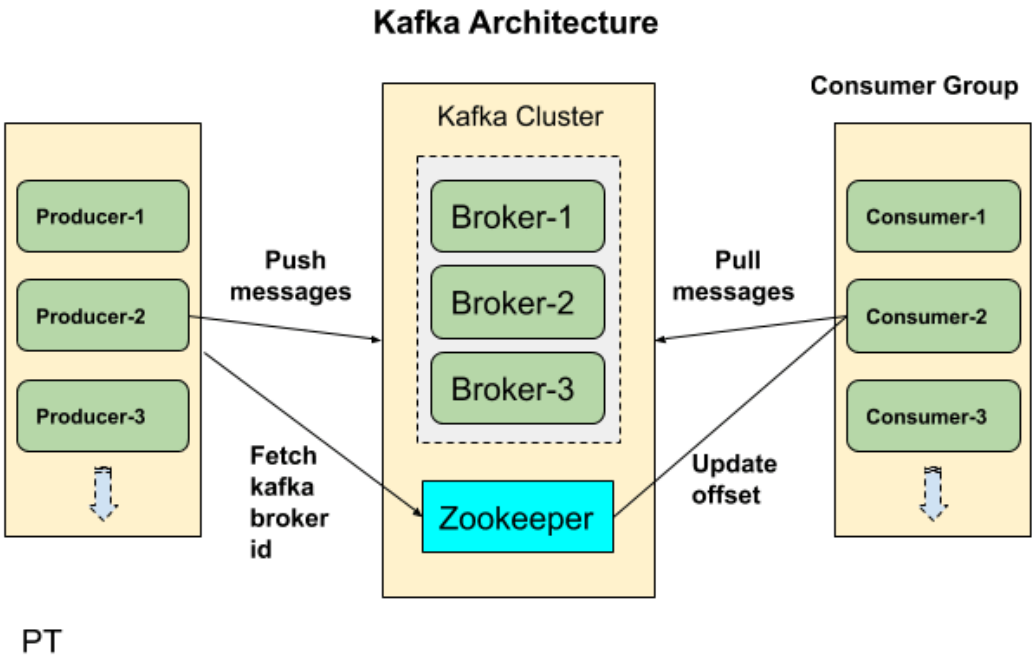

Рис. 2.12. Архітектура Kafka

На малюнку показана високорівнева архітектура Kafka. Постачальники (Producers) відправляють повідомлення на Kafka Topic, який містить потік усіх повідомлень для цієї теми. Кожна тема розподіляється між кластером брокерів Kafka, при цьому кожен брокер має нуль або більше розділів для кожної теми. Кожен розділ - це впорядкований журнал повідомлень, який зберігається на диску. Усі теми доступні для читання будь-якою кількістю споживачів.

Kafka має дуже простий макет зберігання. Кожен розділ теми відповідає логічному логу. Фізично лог реалізований як набір сегментних файлів приблизно однакового розміру (наприклад, 1 ГБ). Кожного разу, коли постачальник публікує повідомлення в розділ, брокер просто додає його до останнього файлу сегмента.

У порівнянні з традиційними publish/subscriber системами, поняття споживача в Kafka узагальнено і представляє собою групу співпрацюючих процесів, які працюють у вигляді кластера. У Apache Kafka тему можна

розділити на кілька розділів. Кожен розділ - це впорядкована послідовність повідомлень, і ці розділи дозволяють паралельну обробку та масштабованість. Групи споживачів формуються набором споживачів, які співпрацюють разом для споживання повідомлень з тем Kafka. Кожен розділ винятково призначається одному споживачу в межах групи споживачів. Це забезпечує, що кожне повідомлення в межах розділу споживається лише одним споживачем у групі.

Виробники повідомлень розподіляють навантаження на посередників і підрозділи або випадковим чином або за допомогою ключа, наданого програмою, для хешування повідомлень розділів посередника. Це розділення на основі ключів має два використання. По-перше, коли дані подаються в систему та розподіляються по різних розділах, в межах кожного розділу зберігається відповідний порядок даних. Тобто, якщо ви додаєте повідомлення до певного розділу, вони будуть оброблятися в тій послідовності, в якій вони надходять.

В контексті обробки потоків даних, особливо у системах, які базуються на архітектурі Kafka або подібних технологіях, важливо розуміти, що порядок повідомлень між різними розділами (partitions) не гарантований. Це означає, що повідомлення, що надходять у різні розділи, можуть бути оброблені в різному порядку. Щоб ефективно вирішити цю проблему, використовується стратегія розділення за ключами. Детальніше про це:

1. Розділення за Ключами (Key-Based Partitioning): Цей метод передбачає визначення ключа для кожного повідомлення, відповідно до якого воно буде направлене в певний розділ. Таким чином, всі повідомлення з однаковим ключем потрапляють в один і той же розділ, що забезпечує збереження порядку в межах цієї групи. Це ефективно при використанні у системах, де важливо зберігати послідовність обробки для повідомлень, пов'язаних з певним об'єктом чи сутністю.

2. Дрібне Розділення Потоків (Granular Stream Division): Використання ключа, такого як ідентифікатор користувача, дозволяє дрібно розділити потік даних на підпотоки. Споживачі, які обробляють ці дані, можуть фокусуватися на

повідомленнях, пов'язаних із конкретним користувачем, забезпечуючи більш цілеспрямовану обробку.

Важливі аспекти цього підходу:

1. Маршрутизація за Ключем (Key-Based Routing): Споживачі можуть вибирати конкретний ключ, наприклад, ідентифікатор користувача, і обробляти лише ті повідомлення, що відповідають цьому ключу. Це розподіляє навантаження між споживачами, які спеціалізуються на різних користувачах або сегментах.

2.Спрощений Аналіз Сесій (Simplified Session Analysis): Ключ дозволяє споживачам аналізувати сесії користувачів без необхідності зберігання всієї історії взаємодій в одному місці. Це полегшує обробку та аналіз взаємодій конкретного користувача, оскільки всі повідомлення, пов'язані з цим користувачем, будуть оброблятися одним і тим же споживачем.

Без застосування такої стратегії, обробники повідомлень могли б зіткнутися з необхідністю доступу до загального сховища даних для синхронізації станів, що могло б значно ускладнити процес обробки та підвищити вартість обміну даними і зберігання інформації.Якщо не було б цієї гарантії (що всі повідомлення для одного користувача надходять до того самого споживача), то розподілені обробники повідомлень були б змушені взаємодіяти зі спільним сховищем для збереження агрегованих станів, що могло б призвести до дорогого обміну даними та операцій зберігання для кожного повідомлення.

Узагальнюючи, Kafka відрізняється від класичних принципів систем обміну повідомленнями кількома моментами:

- Дані розподіляються так, що виробництво, передача та споживання даних обробляються кластерами машин, які можна збільшувати поступово зі зростанням навантаження.
- Kafka гарантує, що повідомлення з одного розділу доставляються споживачеві в порядку. Але немає гарантії порядку повідомлень, що надходять з різних розділів.

- Повідомлення не видаляються з журналу, але їх можна відтворити споживачами (наприклад, під час обробки помилок програми споживача).
- Крім того, стан читача зберігається лише споживачами, що вказує на те, що видалення повідомлень може базуватися лише на вручну налаштованій політиці зберігання. Ця політика зберігання може бути виражена або у вигляді кількості повідомлень (message count), які система зберігає, або у вигляді віку повідомлень (message age), тобто періоду часу, протягом якого повідомлення зберігається в системі.

2.6. Стратегії управління даними: Впровадження Event Sourcing і CQRS у розподілених системах

Загальний шаблон для будь-якої орієнтованої на дані програми - це багаторівнева архітектура . Основна ідея полягає в тому, щоб використовувати розділення відповідальності для того, щоб утримувати читання, зберігання даних та бізнес-логіку відокремлено одне від одного. Рівню збереження (persistence layer) не повинно бути відомо про механізми, що використовуються для зберігання та отримання даних, він відповідає лише за саму операцію зберігання. Рівень даних (data layer) повинен мати справу з різними відносинами між сутностями і часто виступає як міст між моделлю домену програми та її нормалізованим виглядом у сховищі даних. Зміни даних, як правило, виражаються як C, U і D у CRUD (створення, оновлення та видалення).

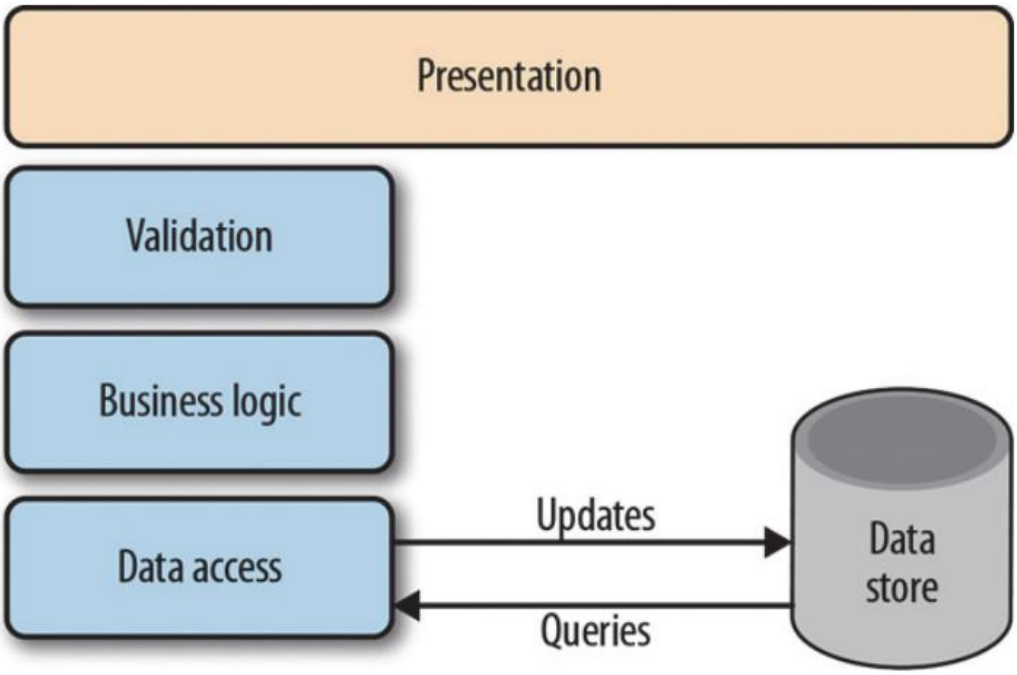

Рис.2.13. Традиційна CRUD архітектура

Загальна проблема CRUD-програм полягає в тому, що вони отримують всі моделі даних з основного сховища даних, від якого вони залежать. Це накладає дві різні вимоги до структури даних: швидкі записи і швидке читання. Ці параметри важко збалансувати, використовуючи лише одне рішення, і в більшості випадків цю проблему пом'якшують, додаючи кеші. Однак із кешами виникає додаткова складність, яка вимагає величезних знань для правильного управління. Ще однією проблемою із системами подібними до CRUD є порушення принципу єдиної відповідальності, оскільки операція оновлення може не лише виконувати оновлення, але і читати нові змінені дані. Таким чином, код стає менш підтримуваним зі зростанням обсягу застосування.

Існує багато способів вирішення описаних вище проблем. Більшість додатків використовують кеші як швидкий та денормалізований спосіб доступу до часто запитуваних даних. Однак їх впровадження додає новий рівень складності, оскільки кеші повинні бути синхронізовані для всіх екземплярів додатку. Розмір та об'єкти, які вони містять, - це спірний момент, оскільки різні додатки можуть використовувати кеші з різних причин. Однак найскладнішою

задачею, пов'язаною з кешами, є їх актуалізація. Кеші не є постійним сховищем даних, що означає, що їх потрібно відновлювати при кожному запуску додатку з якогось джерела. Це створює розрив між цим сховищем даних - source of truth - та кешами.

Транзакції часто погано розглядаються для CRUD-додатків. Хоча вони ефективні для збереження цілісності даних в базах даних SQL, вони викликають багато накладних витрат та додають бізнес-логіку до сховища даних, подальше порушення принципу єдиної відповідальності. Транзакції, які тримаються відкритими протягом тривалого часу, змушують сховище даних відстежувати змінені рядки часто змінюваних таблиць, які можуть бути очищені. Крім того, відкат транзакцій є дуже дороговартісним. Для деяких баз даних відкат транзакцій займає більше часу, ніж їх здійснення.

Інший спосіб підвищення продуктивності додатку - це вертикальне масштабування. У більшості випадків сервери вже на повній потужності. Щоб побачити реальний вплив масштабування, зазвичай потрібно придбати цілком новий сервер для заміни старого. З точки зору бази даних/додатку це все ще вертикальне масштабування. Мати один потужний сервер, як правило, вартує дорожче, ніж кілька менш потужних.

2.6.1. CQRS (Відокремлення обов'язків команд та запитів)

Основна ідея CQRS (Command Query Responsibility Segregation) полягає в відокремленні моделей для оновлення та читання інформації. Колаборація та застарілість даних є основною силою CQRS.

Колаборація означає набір правил того, як багато учасників буде використовувати/змінювати ті самі загальні дані. Часто існують правила, які вказують, який актор може виконати модифікації, які модифікації можуть бути прийняті в одному випадку, які можуть бути заборонені в іншому. Акторами можуть бути люди, такі як звичайні користувачі, або автоматизовані системи.

Застарілість даних демонструється тим, що при спільному використанні, коли дані показуються одному актору, вони можуть бути змінені іншим, і для

першого актора дані залишаться незмінними. Практично будь-яка система, яка використовує кеші, надає застарілі дані - часто з метою поліпшення продуктивності. Це означає, що ми не можемо врахувати рішень акторів цієї системи, оскільки вони можуть бути прийняті на основі застарілих даних. Стандартні рівневі архітектури не вирішують жодну з цих проблем. Хоча весь обсяг даних в одній базі може бути кроком для вирішення питання колаборації, проблема застарілості даних зазвичай більш неприємна в цих архітектурах через використання кешів як покращення продуктивності вже після завершеного проектування моделі даних. Ці проблеми вирішуються в CQRS через відокремлення моделей читання та запису, де запити призначені для читання, а команди - для оновлення даних.

Команди повинні говорити, що потрібно зробити, а не яким чином (Наприклад - "увімкнути світло", а не "встановити статус світла на УВІМКНЕНО"). В більшості випадків їх розміщують в чергу для подальшої асинхронної обробки.

Запити не мають права нічого змінювати в сховищі даних. Вони повинні повертати DTO (об'єкт передачі даних), який виступає як контейнер для даних. Його можна розглядати як структуру.

Порівнюючи потік даних додатків, орієнтованих на CRUD, із тими, що використовують CQRS, реалізація та процес розробки стають значно простішими завдяки відокремленню відповідальностей. Обидві моделі можуть бути розроблені за допомогою одного сховища даних або бути повністю відокремленими з різною структурою. Розділення сховищ читання та запису також дозволяє кожному з них масштабуватися відповідно до навантаження. Наприклад, сховища читання, як правило, стикаються з набагато більшим навантаженням, ніж сховища запису. Денормалізація даних дозволяє створювати оптимізоване читання, яке може суттєво покращити продуктивність, зменшуючи кількість трансформацій даних, необхідних для формування результату.

Переваги цього підходу:

1) Оптимізовані схеми даних. Можливість розробки моделей даних для оптимізації читання, при цьому модель запису повинна гарантувати швидкі операції запису.

2) Незалежне масштабування. Розділення моделей дозволяє незалежно масштабувати різні служби в залежності від їхнього завантаження.

3) Прості запити. Оптимізовані для читання перегляди роблять операцію запиту швидшою та простішою.

4) Безпека. Впровадження прав доступу краще контролюється для акторів, які виконують операції запису.

5) Розділення обов'язків. Підтримка та гнучкість стають кращими внаслідок відокремлення моделей даних. Модель запису повинна бути відповідальною за складну бізнес-логіку, тоді як модель читання залишається відносно простою за структурою та обов’язками.

Недоліки:

1) Складність. Хоча основа CQRS досить проста, фактична реалізація може бути складною.

2) Передача повідомлень У більшості випадків обмін повідомленнями йде рука об руку з CQRS, хоча це не є обов’язковою умовою. Це забезпечує асинхронність у дизайні програми.

3) Узгодженість у кінцевому рахунку. Розділення моделей читання та запису може призвести до застарілих даних читання. Однак втрата або невідповідність даних неможливі за даним дизайном.

2.6.2. Event sourcing

Event sourcing доповнює CQRS, підвищуючи доступність та ефективність програми. Основна ідея полягає в тому, щоб мати упорядкований набір змін, а не агрегований стан. Це означає, що кожна подія не замінює загальних даних, а являє собою дельту. Зазвичай це представлено як append-only журнал, де актор може надсилати команди, а коли потрібно створити стан сутності - можна просто відтворити кожну з цих команд.

Робота з подіями передбачає введення асинхронності в програму. Таким чином, ця модель є в кінцевому підсумку послідовною: кожна подія буде оброблена в майбутньому. Використання журналу як засобу транспортування та зберігання подій запобігає будь-яким паралельними проблемам, оскільки кожна команда для однієї сутності обробляється одна за одною.

Event sourcing та модель журналу широко використовуються в базах даних як інструмент для синхронізації різних вузлів. Кожен запис у базі даних зберігається в її журналі (логу), і коли відбувається реплікація, новому вузлу потрібно просто відтворити всі команди з журналу

При використанні підходу event sourcing і CQRS, різні служби обробки даних можуть споживати події та оновлювати свою модель даних без небажаних наслідків у вигляді проблем з дозволами чи цілісністю даних. Це дає можливість зберігати дані в тій моделі, яка найкраще підходить для конкретної задачі, дублювати та попередньо обчислювати будь-які необхідні дані. Зазвичай програми, які використовують event sourcing, також мають денормалізовані дані, тобто дані, які зберігаються у більш зручному для читання вигляді, ніж для запису.

Маючи один журнал всіх команд, легко здійснювати синхронізацію різних сховищ даних, які можуть бути налаштовані як на інтенсивне читання, так і на інтенсивний запис. Це дуже відрізняється від стандартів нормалізації в типових CRUD-програмах. Наявність різних оптимізованих для читання представлень даних дозволяє значно підвищити продуктивність, і проблеми з цілісністю даних вирішуються кожною службою обробки даних окремо під час оновлення її моделі. Існує чимало переваг використання event sourcing:

- Асинхронність.
- Джерело правильних даних (Source of truth)
- Паралелізм.
- Трасіровка
- Масштабованість.
- Повторне використання.

Event sourcing - це методологія розробки програмного забезпечення, яка дозволяє простим і надійним способом реєструвати зміни в стані системи за допомогою протоколу без втрат. За наявності впорядкованого журналу або журналу подій як основи таких систем, відновлення стану системи стає легким і ефективним процесом.

Відсутність жорсткого зв'язку є результатом створення таких додатків, які зберігають стан. Усунення неполадок і підвищення масштабованості - це найбільш вагомі переваги. Наявність різних представлень даних для оптимізації читання значно покращує продуктивність, і проблеми з цілісністю даних вирішуються кожною службою обробки даних окремо при оновленні її моделі.

2.7. Api Gateway патерн в мікросервісній архітектурі

Api Gateway патерн мікросервісів діє як єдина точка входу для всіх запитів клієнтів. Патерн API Gateway забезпечує консолідований інтерфейс, спрощує загальну архітектуру, підвищує безпеку та сприяє масштабованості та гнучкості. Шлюз API відповідає за обробку автентифікації, маршрутизації, балансування навантаження, кешування тощо. Основна ідея шаблону шлюзу API полягає в тому, щоб відокремити клієнтів від усіх мікросервісів і забезпечити спрощений спільний інтерфейс для зв’язку.

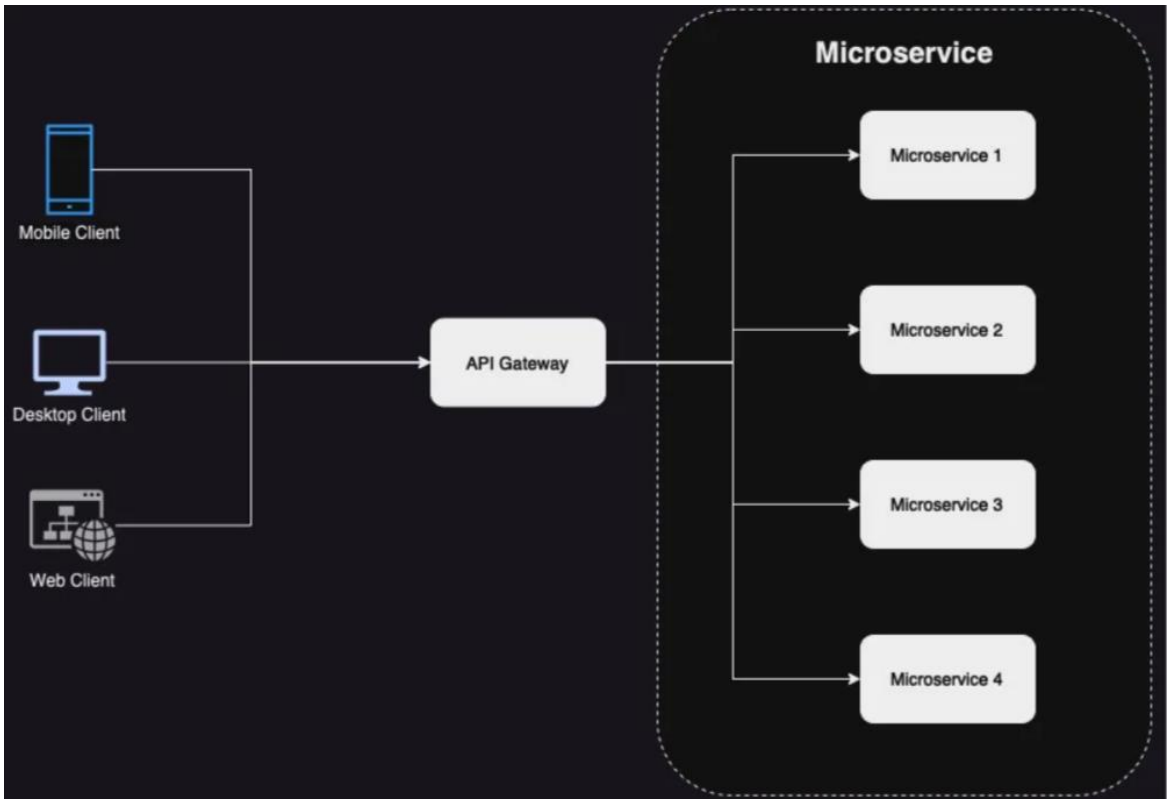

Рис 2.14. Мікросервісна архітектура з використанням шлюзу

У мікросервісних архітектурах клієнт зазвичай споживає більше ніж один мікросервіс. Якщо немає шлюзу, доведеться розгортати мікросервіси окремо, і клієнт повинен мати справу з комунікацією з кількома мікросервісами. Це може стати проблемою в довгостроковій перспективі. Пряма комунікація клієнта з мікросервісом може призвести до проблем, таких як залежність, проблеми з безпекою і численні виклики служб для виконання всього лише одного завдання.

Переваги використання шаблону API-шлюзу мікросервісів:

1) Простий інтерфейс клієнта. Шаблон API-шлюзу спрощує взаємодію клієнта з мікросервісами. Клієнти можуть робити запити в єдиній уніфікованій точці входу.

2) Балансування навантаження та масштабування. Шлюзи API можуть динамічно масштабувати базовий рівень обслуговування на основі вхідного трафіку, що забезпечує оптимальне використання ресурсів і високу доступність.

3) Безпека та автентифікація. Шлюз API може обробляти автентифікацію та авторизацію для всіх вхідних запитів, узгоджено

дотримуючись політик безпеки для всіх мікросервісів. Шлюзи можуть перевіряти токени доступу, аутентифікувати клієнтів, встановлювати обмеження швидкості та забезпечувати, що тільки аутентифіковані запити обслуговуються базовим рівнем сервісу. Ще однією перевагою шаблону шлюзу є можливість реалізації безпеки з єдиної точки.

4) Версіонування API та агрегація. Шаблон API-шлюзу дозволяє версіонування API, що забезпечує плавні переходи та зберігаючи зворотну сумісність. Використовуючи шлюз, ви можете відправляти агреговані відповіді клієнтам. Наприклад, скажімо, вам потрібно отримати дані з трьох мікросервісів для одного запиту клієнта. Шлюз буде обробляти комунікацію та агрегувати всі дані перед відправленням відповіді клієнту.

5) Кешування. Впровадження механізмів кешування в межах API-шлюзу може значно покращити продуктивність та зменшити навантаження на мікросервіси.

6) Трансформація. Використовуючи API-шлюз, можена виконувати трансформації запитів та відповідей. API-шлюзи підтримують різні протоколи зв'язку та керують протоколами відповідно до конкретних потреб клієнтів та мікросервісів.

7) Журналювання та моніторинг. З API-шлюзом стає простіше отримувати та аналізувати журнали та показники, що пов’язані з вхідними запитами. Це допомагає покращити моніторинг, відлагодження та аналізо продуктивності всієї екосистеми мікросервісів.

Недоліки використання шаблону API Gateway

1) Одна точка несправності. Основним недоліком шаблону шлюзу є одноточковий збій. Якщо шлюз виходить з ладу через якусь проблему, серверна частина перестає відповідати. Як заходи запобігання, можна розглядати використання кількох завершених шлюзів для вимог клієнта (BFF) або подумати про масштабоване рішення шлюзу

2) Додаткові витрати на розробку та обслуговування. Ще однією проблемою є додаткові витрати на розробку та обслуговування. Якщо шлюз не

масштабується належним чином, він також може стати «вузьким місцем» для системи.

Використання найкращих практик та використання потужності шаблону API-шлюзу дозволяють розблокувати повний потенціал мікросервісних додатків.

ВИСНОВОК ДО РОЗДІЛУ 2

У даному розділі здійснено детальний аналіз архітектурних рішень проекту. Вивчення монолітної та мікросервісної архітектури дозволило зрозуміти їх основні переваги та недоліки, а також сценарії ефективного використання. Виокремлення вертикального та горизонтального масштабування виявило різні підходи до збільшення продуктивності та масштабованості систем. Вивчення різних способів комунікації між мікросервісами, включаючи синхронне та асинхронне спілкування, засвідчило важливість вибору відповідного методу для оптимізації взаємодії сервісів.

Особлива увага була приділена архітектурі мікросервісів на основі подій, включаючи розгляд виробників та споживачів в подійно-орієнтованих системах, а також ролі брокерів подій. Аналіз переваг та недоліків такої архітектури забезпечив глибоке розуміння її практичного застосування та впливу на загальну ефективність системи.

В кінці розділу було розглянуто вибір розподіленого сховища подій, аналіз системи Publish/Subscribe, а також існуючі реалізації, зокрема Apache Kafka. Це допомогло визначити ключові компоненти для створення ефективної та надійної архітектури на основі подій.

РОЗДІЛ 3

ПРОЕКТУВАННЯ ВЕБ-СЕРВІСУ

3.1. Визначення основних елементів бізнес-логіки створюваного веб-сервісу

Цей веб-сервіс надає користувачам можливість ефективно організовувати свої завдання, керувати календарем та інтегрувати його з різними популярними API, такими як Jira, Google Calendar та іншими службамию. У даному описі представлені основні елементи бізнес-логіки створюваного веб-сервісу, спрямованого на автоматизацію робочого процесу на основі сповіщень:

Аутентифікація користувача:

- Користувачі можуть реєструватися або входити за допомогою OAuth з Github, Google або Microsoft.
- Після успішної аутентифікації застосунок генерує токен доступу, який використовується для подальших запитів до API.

API-шлюз:

- Служить центральною точкою входу для всіх вхідних запитів API.
- Обробляє аутентифікацію та авторизацію за допомогою токенів OAuth.
- Маршрутизує запити до відповідних мікросервісів на основі ендпоінта.

Користувацька панель:

- Після входу користувачам відображається панель із дошкою та календарем.

Дошка:

- Користувачі можуть підключатися та інтегруватися з різними API (Jira, GitHub, GitLab), щоб отримувати завдання та проекти.
- Користувачі можуть створювати власні тригери та події для кожного підключеного API.
- Користувачі можуть перетягувати завдання та проекти між різними дошками.
- Оновлення надходять в реальному часі, відображаючи зміни в підключених API.

Календар:

- Користувачі можуть інтегруватися з API, такими як Google Calendar, Microsoft Teams і т.д., для отримання подій календаря.
- Користувачі можуть створювати, редагувати та видаляти події календаря безпосередньо з застосунку.

Існуючі інтеграції:

- Користувачі можуть створювати інтеграції, на основі вже існуючих. Таких як Jira, Google Calendar, Gmail, Microsoft Teams

Власні інтеграція:

- Користувачі можуть створювати власні інтеграції API, за допомогою визначення тригерів (triggers) та дій (actions). Triggers – подія, при якій треба виконати дію. Action – дія, яка має виконатись.
- Інтерфейс зручний для користувача і дозволяє вказувати ендпоінти API, дані аутентифікації та відображення даних.
- Для отримання оновлень в реальному часі можна налаштувати polling для визначення частоти оновлення інформації.

Система повідомлень:

- Користувачі отримують сповіщення про важливі події, оновлення або зміни в інтегрованих службах.

- Повідомлення можна налаштовувати відповідно до вибору користувача.

Налаштування користувача:

- Користувачі можуть керувати налаштуваннями свого облікового запису, включаючи інформацію про профіль, налаштування повідомлень та інтеграцій API.

Безпека:

- Всі комунікації між мікросервісами та API-шлюзом захищені за допомогою протоколу HTTPS.
- Контроль доступу на основі ролей гарантує, що користувачі можуть отримувати доступ та модифікувати дані тільки з необхідними дозволами.

Масштабованість:

- Архітектура мікросервісів дозволяє масштабувати кожен мікросервіс незалежно від потреби.

Логування та моніторинг:

- Реалізовано комплексне логування для відстеження дій користувачів, API-запитів та помилок.

3.2. API-шлюз

API Gateway діє як центральна точка входу для зовнішніх клієнтів, полегшуючи зв’язок із IntegratorService. Його мета — абстрагувати внутрішню структуру, підвищити безпеку та надати додаткові функції, такі як автентифікація, журналювання та моніторинг.

3.2.1. Основні компоненти та обов'язки:

Маршрутизація запиту. Шлюз API відповідає за маршрутизацію вхідних запитів до відповідних мікросервісів. У цьому випадку всі запити спрямовуються до IntegratorService.

Безпека та автентифікація. Обробляє автентифікацію та авторизацію вхідних запитів перед їх пересиланням до IntegratorService. Це гарантує, що лише автентифіковані користувачі можуть взаємодіяти з системою.

Логування та моніторинг. Реалізує централізовану реєстрацію та моніторинг. Кожен вхідний запит разом із відповідними метаданими реєструється для аналізу та налагодження. Цю інформацію можна використовувати для відстеження використання системи, виявлення проблем і створення показників продуктивності.

Балансування навантаження. Надає можливості балансування навантаження, розподіляючи вхідні запити між кількома примірниками IntegratorService. Це забезпечує оптимальне використання ресурсів і масштабованість у міру зростання системи.

Абстракція внутрішньої структури. Абстрагує внутрішню структуру мікросервісів від зовнішніх клієнтів. Клієнти взаємодіють зі шлюзом API без необхідності визначення конкретної URL-адреси чи кінцевої точки сервісів.

Централізована конфігурація. Керує централізованою конфігурацією правил маршрутизації та політик безпеки. Це дозволяє динамічно оновлювати поведінку системи, не вимагаючи змін індивідуальних конфігурацій клієнта.

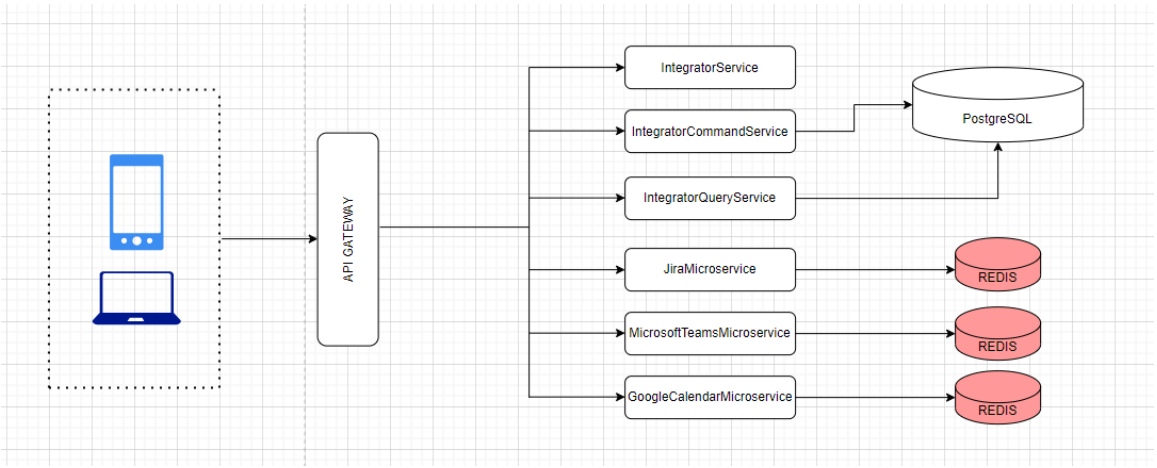

Рис.3.1. Загальний вигляд мікросервісної архітектури з імплементацією API-Gateway

3.2.2. Реалізація

Для реалізації використовується Spring Cloud Gateway як базова структура для реалізації API Gateway. Він надає такі функції, як маршрутизація, фільтрація та балансування навантаження з коробки. Spring Cloud Gateway підтримує балансування навантаження з коробки. Якщо у є кілька екземплярів IntegratorService, Spring Cloud Gateway автоматично розподілятиме запити. Якщо вводяться додаткові мікросервіси або маршрути, шлюз може легко врахувати ці зміни.

Приклад налаштування API-шлюзу. Тут визначаються маршрути в конфігурації API Gateway. У цьому випадку вказується маршрут, який пересилає запити до ItegratorService, IntegratorCommandService, IntegratorQueryService, GoogleCalendarMicroservice, JiraMicroservice, CustomIntegrationMicroservice

```
@Configuration
@EnableWebSecurity
public class GatewayConfig {
   @Bean
   public RouteLocator customRoutes(RouteLocatorBuilder builder) {
      return builder.routes()
            .route("query-service ", r -> r.path("/query/**")
                  .uri("http://locahlhost:8081/query-service"))
            .route("command-service", r -> r.path("/command /**")
                  .uri("http://localhost:8082/command -service"))
            .route("integrator-service", r -> r.path("/integrator /**")
                  .uri("http://localhost:8083/integrator -service"))
           .route("g-calendar-service", r -> r.path("/g-calendar /**")
                  .uri("http://localhost:8084/google-calendar-service”))
            .route("jira-service", r -> r.path("/jira/**")
                  .uri("http://localhost:8084/jira-service"))
            .route("ms-teams-service", r -> r.path("/ms-teams/**")
```

```
            .uri("http://localhost:8085/microsoft-teams-service "))
            .route(“custom-integration-service", r -> r.path("/cus-teams/**")
            .uri("http://localhost:8086/custom-integration-service "))
            .build();
    }
}
```

Фільтр журналювання потрібен, щоб фіксувати релевантну інформацію про кожен запит, включаючи позначки часу, деталі клієнта та дані запитів/відповідей.

```
@Configuration
public class LoggingFilter {
    private Logger logger = Logger.getLogger(LoggingFilter.class.getName());
    @Bean
    public GlobalFilter customLoggingFilter() {
        return (exchange, chain) -> {
            ServerHttpRequest request = exchange.getRequest();
            ServerHttpResponse response = exchange.getResponse();
            // Log relevant information about each request
            logRequestDetails(request);
            // Continue processing the request
            return chain.filter(exchange)
                    .doOnTerminate(() -> {
                        logResponseDetails(response);
                    });
        };
    }
    private void logRequestDetails(ServerHttpRequest request) {
        logger.info("Request received at: " + System.currentTimeMillis());
        logger.info("Request Method: " + request.getMethodValue());
        logger.info("Request URI: " + request.getURI());
```

```
    }
    private void logResponseDetails(ServerHttpResponse response) {
        logger.info("Response sent at: " + System.currentTimeMillis());
        logger.info("Response Status Code: " + response.getStatusCode());
    }
}
```

3.2.3. Авторизація та аутентифікація

Як описувалось раніше, API- шлюз буде оброблювати запити за допомогою токенів, наданих OAuth серверами та валідувати їх.

Аутентифікація користувачів відбувається за допомогою декількох OAuth2 серверів. Таких як GitHub, Google, Microsoft. Розглянемо прикла дреалізації налаштування аутентифікації за допомогою Github (для інших випадків логіка така сама):

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests()
                .antMatchers("/", "/login**", "/error**")
                .permitAll()
                .anyRequest().authenticated()
                .and()
            .oauth2Login()
                .loginPage("/")
                .and()
            .logout()
                .logoutSuccessUrl("/")
```

```
            .permitAll();
    }
}
```

Для користуванням OAuth за допомогою GitHub ми маємо зареєструвати свій додаток у відповідному сервісі та вказати домашню сторінку додатку та url сторінки для перенаправлення.

Використовуючи механізм авторизації OAuth, служби можуть авторизуватися від імені користувача:

1. Додаток запитує у GitHub код користувача
2. GitHub запитує схвалення від користувача
3. Користувач дозволяє
4. GitHub надає програмі маркер обмеженої авторизації , який містить лише наявні дозволи

Програма передає маркер авторизації кожного разу до GitHub, валідуючи його актуальність та правильність.

В OAuth є чотири основні актори:

- Власник ресурсу (RO) : суб’єкт, який контролює дані, надані через API, як правило, кінцевий користувач.
- Клієнт : мобільний додаток, веб-сайт тощо, який хоче отримати доступ до даних від імені власника ресурсу
- Сервер авторизації (AS) : Служба маркерів безпеки (STS) або, простіше, сервер OAuth, який видає маркери
- Сервер ресурсів (RS) : служба, яка надає доступ до даних, тобто API

У наведеному вище коді ми хочемо, щоб кожен запит був автентифікований. Тепер, якщо ми спробуємо отримати доступ localhost:8080 - у браузері буде перенаправлено на сторінку входу GitHub. Коли надсилається запит до localhost:8080, Spring security намагатиметься знайти автентифікований об’єкт, але зрештою це не вдасться. Тому він перенаправляє на http://localhost:8080/oauth2/authorization/github. Внутрішньо цей запит

обробляє OAuth2AuthorizationRequestRedirectFilter, який використовує інструменти doFilterInternal, які збігаються з /oauth2/authorization/githubURI, і перенаправляє запит на https://github.com/login/oauth/authorize?response_type=code&client_id=<clientId>&scope=read:user&state=<state>&redirect_uri=http://localhost:8080/login/oauth2/code/github

Після успішної автентифікації на GitHub користувач буде перенаправлений із login/oauth2/code/github з кодом автентифікації в параметрах запиту. Це буде оброблено OAuth2LoginAuthenticationFilter, який виконає POST запит до GitHub API, щоб отримати маркер автентифікації.

3.3. Огляд імплементованих мікросервісів

3.3.1. IntegratorService

IntegratorService діє як основний елемент для обробки бізнес-логіки та координації взаємодії між різними мікросервісами. Ця служба є нерозривною частиною екосистеми мікросервісів, граючи роль центрального вузла для багатьох критично важливих операцій.

Основні завдання та функції IntegratorService можна описати наступним чином:

1. Виконання бізнес-лоігки: IntegratorService здійснює основну бізнес-логіку, оброблює події необхідним чином, віддає клієнту дані, які були запрошені.

2. Агрегація та Трансформація Даних: Служба агрегує та перетворює дані, отримані з різноманітних джерел, включаючи зовнішні сервіси як Jira, Google та Microsoft Teams. Це дозволяє створювати єдиний вигляд інформації з різних джерел для легшого доступу та обробки.

3. Взаємодія з Kafka: IntegratorService використовує Kafka для підписки на події, що дозволяє службі швидко реагувати на зміни як у внутрішній системі,

так і в інтегрованих зовнішніх сервісах. Водночас, вона також публікує події у Kafka, щоб інші мікросервіси могли їх обробляти.

4. Виклики REST API: Для ситуацій, що вимагають безпосередньої взаємодії, наприклад для надання негайної відповіді клієнту, IntegratorService використовує REST API виклики до інших мікросервісів. Це дозволяє ефективно інтегрувати та перетворювати дані з різних джерел, забезпечуючи консистентність інформації в усій системі.

3.3.2. IntegratorQueryService:

IntegratorQueryService відіграє важливу роль в архітектурі системи, виступаючи як центральна точка доступу для зовнішніх систем і клієнтів, коли вони потребують інформації про поточний стан системи. Ця служба фокусується на операціях читання, відповідаючи за витягування та надання актуальної інформації.

В рамках архітектури, заснованої на принципах Command Query Responsibility Segregation (CQRS), IntegratorQueryService відповідає за 'Read' частину, тобто за запити на читання даних. Він ефективно керує цими запитами, використовуючи оновлену модель читання, яка синхронізується через події, отримані від системи Kafka. Це означає, що служба постійно оновлює свою інформацію, слідкуючи за новими подіями, які відбуваються в системі, і відображаючи ці зміни у відповідях на запити.

Завдяки інтеграції з Kafka, IntegratorQueryService функціонує як споживач подій, обробляючи їх за допомогою спеціалізованих обробників. Це дозволяє службі підтримувати модель читання в актуальному стані, що є ключовим для забезпечення точної та своєчасної інформації.

Що стосується зберігання даних, IntegratorQueryService використовує базу даних PostgreSQL, яка оптимізована для ефективних операцій читання. Ця база даних допомагає в забезпеченні швидкого доступу до великих обсягів інформації.

IntegratorQueryService пропонує стандартизований RESTful API, який дозволяє легко і зручно запитувати інформацію з системи. Цей підхід до дизайну API спрощує інтеграцію з іншими сервісами та клієнтськими програмами.

Для забезпечення стабільності та ефективності, IntegratorQueryService включає механізми керування паралельністю і управління транзакціями, які дозволяють обробляти множинні запити одночасно, зберігаючи цілісність даних.

Завдяки цим характеристикам, IntegratorQueryService значно сприяє ефективній та послідовній взаємодії з системою, забезпечуючи чітке розділення відповідальностей між операціями читання та запису в рамках загальної архітектури системи.

3.3.3. IntegratorCommandService:

IntegratorCommandService відіграє критичну роль у взаємодії з зовнішніми системами та клієнтами. Ця служба спеціалізується на обробці команд, які ініціюють зміни у стані системи. Вона виконує ключові операції запису, такі як створення нових записів, оновлення існуючих та видалення застарілих даних.

У рамках архітектури, IntegratorCommandService впроваджує патерн Command Query Responsibility Segregation (CQRS), схоже на те, як це робить IntegratorQueryService. Проте на відміну від останнього, який зосереджений на читанні даних, IntegratorCommandService орієнтований на запис, виступаючи як авторитетне джерело для відслідковування змін у системі.

Значною характеристикою цієї служби є її інтеграція з системою обміну повідомленнями Kafka. Замість прямого оновлення моделі читання, IntegratorCommandService генерує події, які публікуються в Kafka. Ці події відображають зміни, зроблені користувачами, і є незмінними після їх публікації, що забезпечує надійне збереження історії змін.

Використовуючи Kafka, IntegratorCommandService дозволяє асинхронну взаємодію між різними компонентами системи. Це дає можливість

IntegratorQueryService та іншим мікросервісам обробляти ці події відповідно до власного графіка та потреб.

Додатково, ця служба включає в себе ретельну перевірку вхідних даних, забезпечуючи їх відповідність встановленим бізнес-правилам та забезпечуючи їх достовірність. Вона також імплементує оптимістичний контроль паралельності, що дозволяє ефективно управляти одночасними змінами.

Забезпечуючи кінцеві точки для зовнішніх систем і клієнтів, IntegratorCommandService надає їм можливість ініціювати операції запису. Ці кінцеві точки слідують RESTful принципам, пропонуючи стандартизований спосіб надсилання запитів.

В результаті, IntegratorCommandService грає ключову роль у забезпеченні цілісності та ефективності обробки операцій запису, одночасно пропонуючи повну історію змін стану системи.

3.3.4. JiraMicroservice, MicrosoftTeamsMicroservice, GoogleCalendarMicroservice, CustomIntegrationMicroservice

Це спеціалізовані мікросервіси, призначені для взаємодією з відповідними платформами. Їх основна мета — забезпечити безперебійну інтеграцію та синхронізацію даних між платформою інтеграції та відповідними сервісами.

Ключові обов'язки:

1) Споживання події. Підписується на відповідні події брокера повідомлень Kafka, пов’язані з інтеграціями платформ. Відстежують зміни та оновлення, які викликають події на інтеграційній платформі.

2) Зберігання даних у Redis. Зберігають конфігураційні дані про події в базі даних Redis.Зберігають такі важливі деталі, як userId, authToken, URL-адреса, тіло запиту, заголовки та pollingFrequency.

3) Взаємодія з платформами. Використовують інформацію про частоту оновлення, що зберігається в Redis, щоб визначити, як часто потрібно ходити на кінцеві точки платформ для оновлень інформації тобто періодично робить запити на кінцеві точки відповідно до вказаних інтервалів.

4) Використання маркера автентифікації. Використовують маркери автентифікації для встановлення безпечних з'єднань із кінцевими точкам. Гарантують, що всі взаємодії автентифіковані та авторизовані на рівні прав користувача

5) Обробка даних. Обробляють дані, отримані з платформ під час періодичних запитів. Застосовує будь-які необхідні перетворення або фільтрування для отримання відповідної інформації.

6) Публікація події до Кафки: Публікує відповідні події, що відображають зміни або оновлення від відповідної платформи до брокера повідомлень Kafka. Та навпаки. В деяких випадках, наприклад як отримати всі завдання з проекту Jira, використовується REST запити, оскільки такий запит вимагає негайної відповіді від сервера.

Завдяки підписці на події та опитуванню кінцевих точок ці мікросервіси гарантують, що інтеграційна платформа залишається синхронізованою з інформацією.

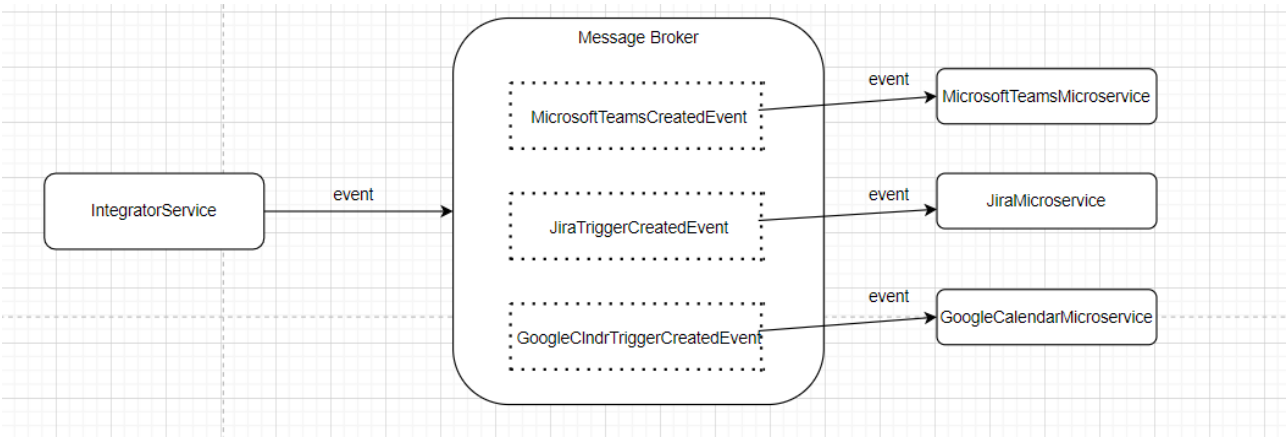

Рис. 3.2. Представлення взаємодії між мікросервісами за допомогою публікацій та зчитування подій

3.4. Імплементація CQRS

3.4.1. Реалізація Read-частини

У цьому контексті частина READ зазвичай включає обробку запитів і обслуговування даних із читаної бази даних. Ми створимо обробник запитів і REST API для отримання даних. Наприклад, маємо TaskCreatedEvent.:

```
@Data
public class TaskCreatedEvent {
    private UUID uuid;
    private Task task;
}
```

Подія TaskCreatedEvent у наданому прикладі є представленням події, яка виникає, коли в системі створюється нове завдання. Ця подія містить інформацію про створене завдання, його ідентифікатор (taskId) і опис.

Для того, щоб оновлювати ReadModel, і мати актуальні дані – треба створити свій Listener:

```
@Log4j2
@Component
public class KafkaTaskEventListener {
    @Autowired
    private TaskService taskService;
    @KafkaListener(topics = "${message.topic.createTask}")
    public void listenCreateTask(ConsumerRecord<String, String> stringStringConsumerRecord) throws Exception {
TaskEvent taskEvent = new Gson().fromJson(stringStringConsumerRecord.value(), TaskEvent.class);
        taskService.createTask(taskEvent.getTask());
        log.info("Insert Task {} in Read database", taskEvent.getTask());
    }
}
```

Вказаний код визначає клас слухача подій Kafka (KafkaTaskEventListener). Цей слухач призначений для споживання повідомлень із теми Kafka та

виконання певних дій, коли повідомлення отримано. Кожен раз, коли в EventStorage будуть нові події – ReadModel буде виконувати відповідні запити до БД, оновлюючи стан бази.

3.4.2. Реалізація Write частини.

У контексті CQRS частина Write передбачає обробку команд, які є операціями, що змінюють стан системи. Частина Write відповідає за обробку команд, їх перевірку та генерацію відповідних подій, які представляють зміни стану.

Приклад створення події:

```
@Component
@Log4j2
public class KafkaTaskEventSourcing {
    @Autowired
    private KafkaTemplate<String, String> kafkaTemplate;
    @Value(value = "${message.topic.createTask}")
    private String topicName;
    public TaskCreatedEvent publicCreatePhoneEvent(Task task) throws JsonProcessingException {
        val id = UUID.randomUUID();
        ObjectWriter objectWriter = new ObjectMapper().writer().withDefaultPrettyPrinter();
        val json = objectWriter.writeValueAsString(taskEvent);
        log.info("Send json '{}' to topic {}", json, topicName);
        kafkaTemplate.send(topicName, json);
        return TaskCreatedEvent.builder().uuid(id).task(task).build();
    }
}
```

У даному випадку джерелом подій (event storage) є Kafka (Тобто Event Bus). Події не видаляються при їх прочитанні, а лише позначаються як прочитані.

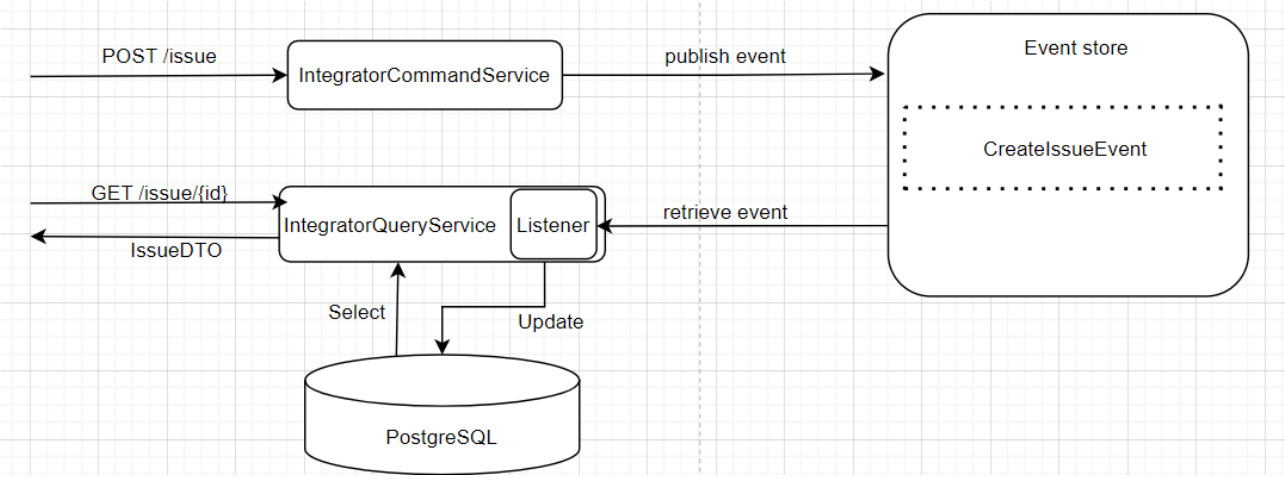

Рис.3.3. Вигляд імплементованої структури CQRS

3.5. Реалізація мікросервісів

Загалом реалізація інших мікросервісів дуже схожа між собою, оскільки вимагаються майже одні й ті самі дії. Розрізняє їх лише контекст та спосіб обробки інформації.

Розглянемо JiraMicroservice. Цей мікросервіс має свій REST API, для негайного отримання інформації. Наприклад:

```
@RestController
@RequestMapping("/api/jira")
public class JiraController {
    private final JiraService jiraService;
    public JiraController(JiraService jiraService) {
        this.jiraService = jiraService;
    }
    @GetMapping("/issues")
    public String getIssues(
            @PathVariable String issueKey,
            @RequestHeader("Authorization") String jwtToken) {
        // Pass the JWT token to the JiraService
```

```
        return jiraService.getIssueDetails(issueKey, jwtToken);
    }
}
```

Даний контроллер вертає всі пов’язані з користувачем завдання. JiraMicroservice не має функціоналу для аутентифікації юзерів. Натомість JWT токен, з яким треба звернутись до Jira надходить із зовнішньої системи. В даному випадку це буде IntegratorCommandService.

Цей мікросервіс також спілкується з Kafka, для періодичного звертання до Jira. Коли юзер створює свою інтеграції – IntegratorCommandService надсилає до Kafka івент «JiraTriggerCreated”, який каже про те, що новий івент був створений. Jira зберігає цей трігер на своїй стороні у базі даних Redis. Цей івент містить основну інформації про Trigger: id, userId, url, httMethod, jsonBody, authToken, pollingFrequency.

За допомогою поля pollingFrequency визначається, як часто треба виконувати цей трігер. Кожен раз, коли трігер був використаний – JiraMicroservice записує нову подію «TriggerEvaluatedEvent” у Kafka, яка каже про те, що звернення до Jira завершилось успіхом.

Приклад реалізації JiraService:

```
@Service
public class IntegratorQueryService {
    private final KafkaTemplate<String, Object> kafkaTemplate;
    private final JiraRestClient  jiraRestClient;
    private final ThreadPoolTaskScheduler taskScheduler;
    private static final String JIRA_TRIGGER_CREATED_TOPIC = "jira-trigger-created";
  private static final String TRIGGER_EVALUATED_TOPIC = "trigger-evaluated";
        // Map to store the scheduled tasks for each trigger
        private final Map<String, ScheduledFuture<?>> scheduledTasks = new HashMap<>();
        public IntegratorQueryService(
```

```
            KafkaTemplate<String, Object> kafkaTemplate,
            ThreadPoolTaskScheduler taskScheduler,
            JiraRestClient jiraRestClient
        ) {
            this.kafkaTemplate = kafkaTemplate;
            this.taskScheduler = taskScheduler;
            this. jiraRestClient = jiraRestClient;
        }
        public void createJiraIntegration(JiraTriggerCreatedEvent event) {
            // Save the trigger information in the Redis database of Jira
            saveTriggerInRedis(event);
            // Send JiraTriggerCreatedEvent to Kafka
            kafkaTemplate.send(JIRA_TRIGGER_CREATED_TOPIC, event);
    // Schedule polling based on pollingFrequency
            schedulePolling(event);
        }
        private void saveTriggerInRedis(JiraTriggerCreatedEvent event) {
          redisTemplate.opsForValue().set(event.getId(), event);
        }
        private void schedulePolling(JiraTriggerCreatedEvent event) {
            String triggerId = event.getId();
            int pollingFrequency = event.getPollingFrequency();
            // Cancel existing scheduled task if it exists
            cancelScheduledTask(triggerId);
            // Schedule a new task based on pollingFrequency
            ScheduledFuture<?>                    scheduledTask                    =
taskScheduler.scheduleAtFixedRate(() -> performPolling(event),
                    Duration.ofMinutes(pollingFrequency)   );
            // Save the scheduled task in the map
            scheduledTasks.put(triggerId, scheduledTask);
```

```
        }
        private void performPolling(JiraTriggerCreatedEvent event) {
            jiraRestTemplate.sendRequest(event);
            triggerEvaluatedSuccessfully(event.getId());
        }
public void triggerEvaluatedSuccessfully(String triggerId) {
            TriggerEvaluatedEvent triggerEvaluatedEvent = new
TriggerEvaluatedEvent(triggerId);
            kafkaTemplate.send(TRIGGER_EVALUATED_TOPIC, triggerId,
triggerEvaluatedEvent);
        }
        private void cancelScheduledTask(String triggerId) {
            ScheduledFuture<?> scheduledTask = scheduledTasks.get(triggerId);
            if (scheduledTask != null && !scheduledTask.isCancelled()) {
                scheduledTask.cancel(true);
            } }}
```

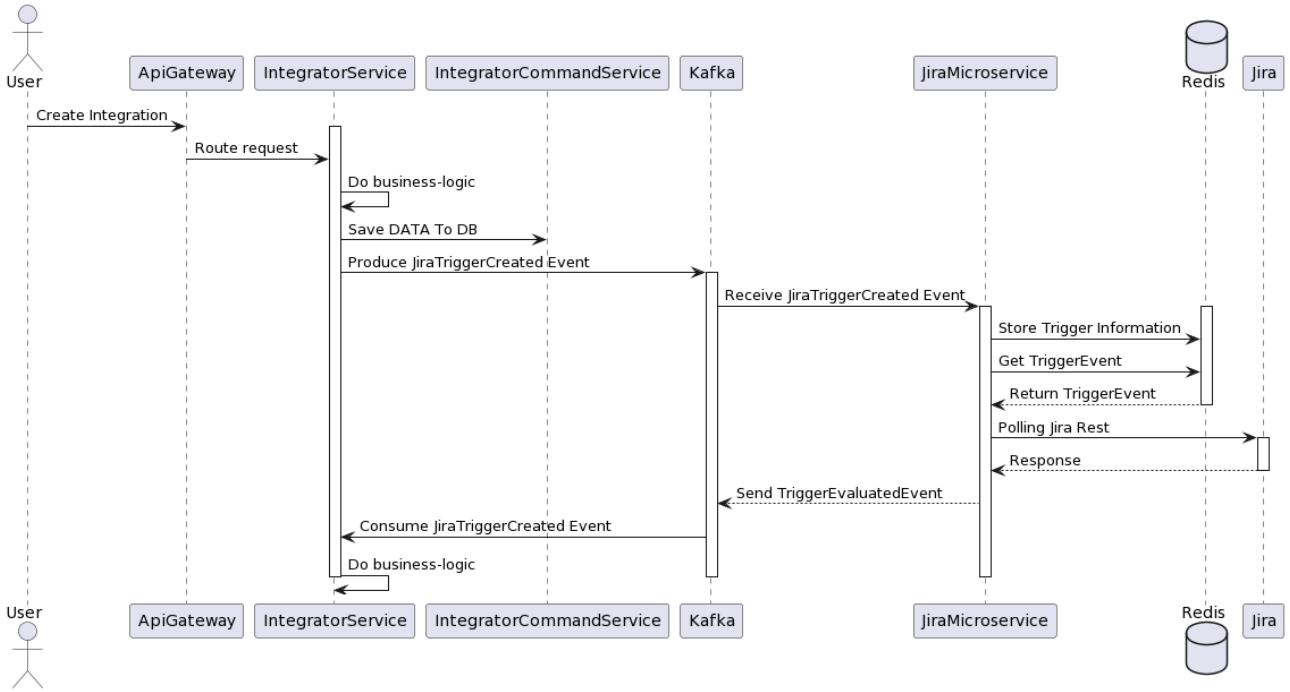

Рис.3.4. Sequence діаграма надсилання користувачем POST запиту на створення інтеграції.

На даній діаграмі зображено:

ApiGateway – шлюз, який авторизує користувача та перенаправляє запити. IntegratorService отримує запит, робить небхідну бізнес-логіку, надсилає запит до IntegratorCommandService для збереження даних, а також створює подію, яка каже про те, що був створений новий Trigger. При отриманні подій – також робить необхідну бізнес-логіку.В даному випадку – виконує всі Action, пов’язані з виконаним трігером.

Jira Microservice - мікросервіс, який відповідає за обробку подій, пов'язаних з Jira. Після отримання команди від IntegratorCommand Service, він взаємодіє з Jira API для отримання додаткових даних. Пересилає подію до Kafka для підписки і обробки подальших операцій.

Jira API - інтерфейс взаємодії з Jira, використовуваний для отримання додаткових даних, пов'язаних із Jira.

Kafka - система для обміну повідомленнями, яка використовується для асинхронної комунікації між мікросервісами. Події публікуються та споживаються мікросервісами через Kafka.

Redis – in-memory база даних типу ключ-значення. Тут зберігаються необхідні дані для періодичного надсилання даних на потрібний ресурс.

Діаграма показує, як подія Jira обробляється в системі, збереження потрібної інформації про подію в базі даних, відправку події до Kafka та обробку події мікросервісом Jira.

3.6. Інтерфейс користувача

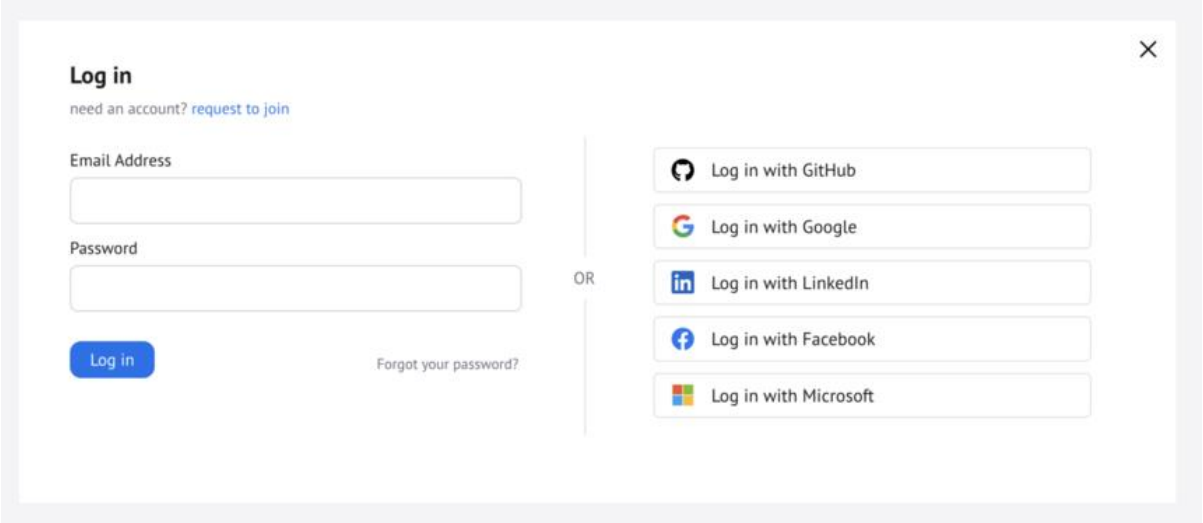

Рис.3.5. Сторінка авторизації.

Авторизацію можна виконати як просту за допомогою пошти, так і за допомогою SSO.

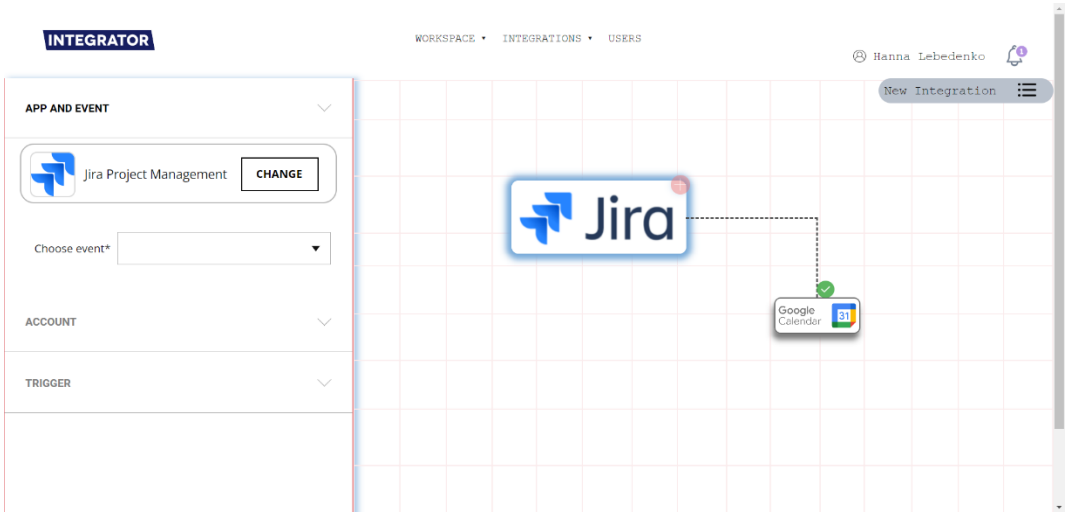

Рис.3.6. Створення тригеру

Приклад налаштування тригеру. Тут вказується Application, який буде використаний в ролі тригера, також авторизацію за допомогою OAuth

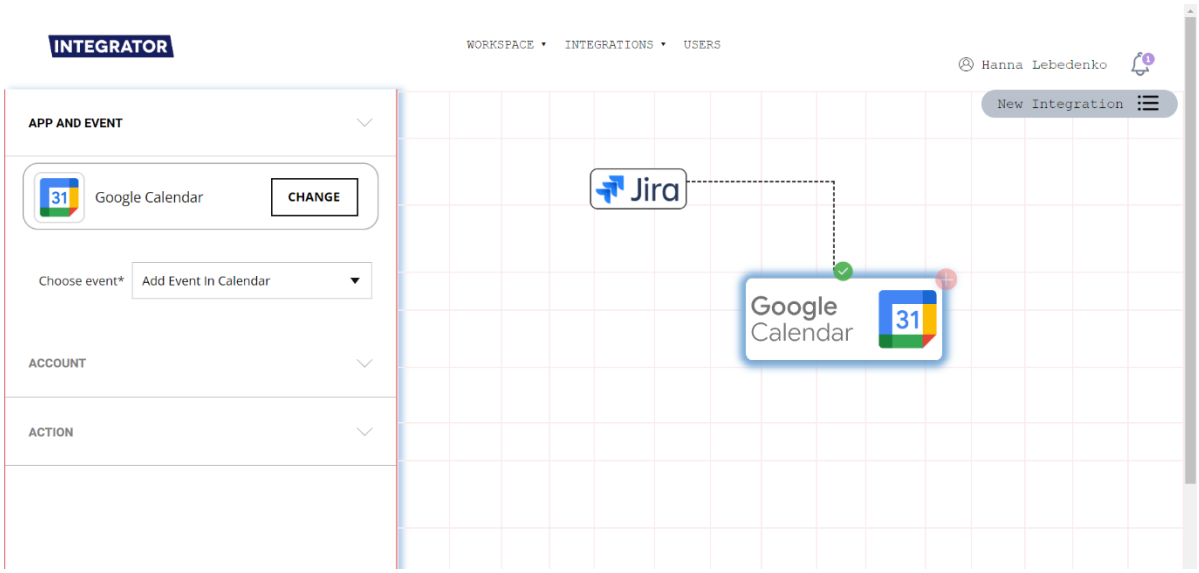

Рис.3.6. Створення Ation

Приклад налаштування Action. Тут вказується Application, який буде використаний у ролі Action та подія, яка буде виконана. В даному випадку – створення нової події у Google Calendar

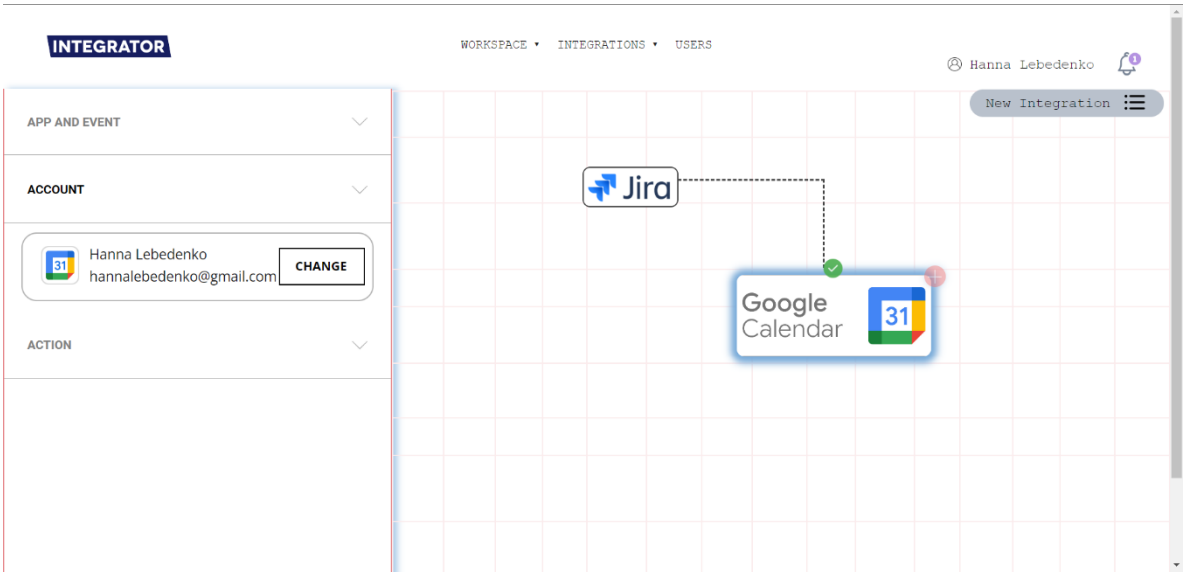

Рис.3.7. Налаштування авторизації для Action за допомогою Oauth

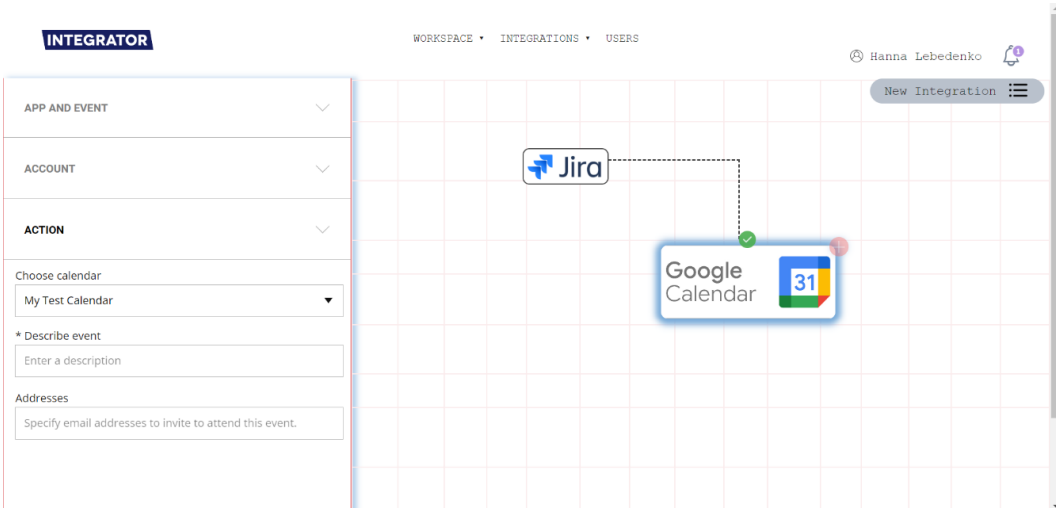

Рис.3.8. Налаштування дій, які будуть виконані на основі сповіщення

Налаштування конкретних дій, які будуть виконані для цього Action (В який календар буде створена нова подія,опис події та отримувачі сповіщення про нову подію).

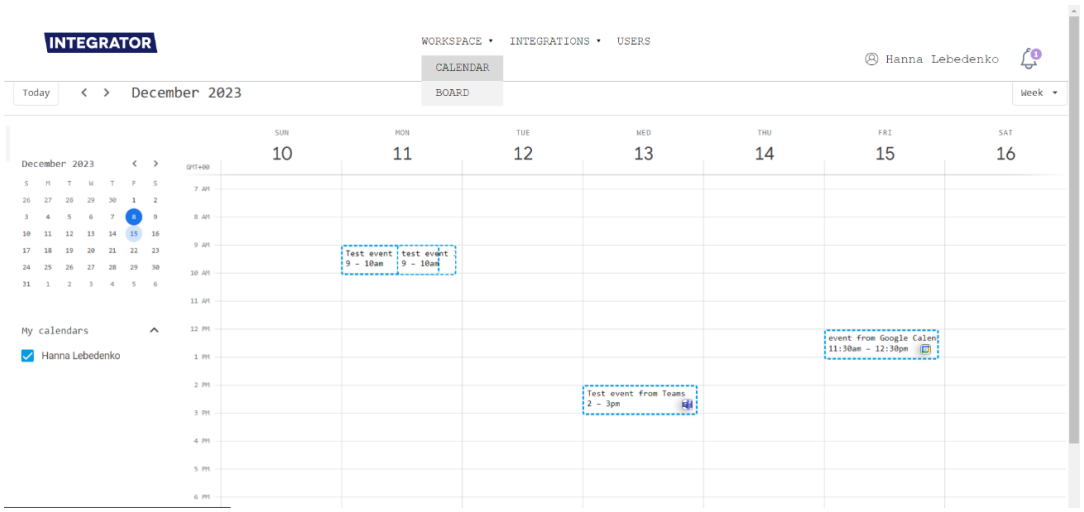

Рис.3.9. Універсальний календар

Вигляд календаря, який об’єднує події з декількох систем. Тут присутні як внутрішні подіїї, що притаманні тільки даній системі, а також події синхронізовані з різними платформами.

ВИСНОВОК ДО РОЗДІЛУ 3

У третьому було зосереджено на проектуванні веб-сервісу, розглядаючи ключові аспекти від визначення основних елементів бізнес-логіки до реалізації конкретних мікросервісів.

Було почато з визначення основних елементів бізнес-логіки, що є фундаментом для створюваного веб-сервісу. Подальше обговорення стосувалося розробки API-шлюзу, включаючи його основні компоненти, обов'язки та реалізацію, а також важливі аспекти авторизації та аутентифікації.

Окрему увагу було приділено огляду та аналізу імплементованих мікросервісів, які включають IntegratorService, IntegratorQueryService, IntegratorCommandService, а також специфічні мікросервіси для інтеграції з різними зовнішніми сервісами, такими як Jira, Microsoft Teams, Google Calendar. Важливим аспектом проектування була імплементація CQRS (Command Query Responsibility Segregation), яка дозволила оптимізувати процеси читання та запису в системі, розділивши їх на дві окремі частини.

Також була розглянута реалізація мікросервісів, яка є ключовим елементом у створенні ефективного та гнучкого веб-сервісу.

ВИСНОВКИ

Дана робота охоплює важливість інтеграції API, розгляд монолітної та мікросервісної архітектур, а також проектування веб-сервісів, підкреслюючи критичне значення адаптивності та масштабованості в сучасних програмних рішеннях.

Перший розділ підкреслює роль інтеграції API як фундаменту для ефективної взаємодії між програмами, звертаючи увагу на складнощі, пов'язані з інтеграцією різних API. Визначається, що архітектура, керована подіями, і платформи, такі як Zapier і IFTTT, мають важливе значення у розвитку інтеграційних рішень.

Другий розділ зосереджений на аналізі архітектурних рішень, зокрема на перевагах та недоліках монолітних та мікросервісних архітектур. Важливість різних способів масштабування та комунікації між мікросервісами виокремлюється, а також обговорюється роль подійно-орієнтованих систем, зокрема Apache Kafka, у створенні ефективних архітектур.

Третій розділ охоплює аспекти проектування веб-сервісів, включаючи розробку API-шлюзу, реалізацію CQRS та мікросервіси для інтеграції з зовнішніми сервісами.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Lequerica J. R. Web Services. Anaya Publishers, 2003. 336 с.
2. Microservices Patterns. CHRIS RICHARDSON : Manning Publications Co., 2019. 522 с.
3. Fowler M. Patterns of Enterprise Application Architecture / M. Fowler. — Boston: Addison-Wesley Longman Publishing Co., Inc., 2002. — 576 с.
4. Udi D. Clarified CQRS [Електронний ресурс]: http://udidahan.com/2009/12/09/clarified-cqrs (дата звернення: 18.11.2023).
5. Richards M. Software Architecture Patterns / M. Richards. — Sebastopol: O'Reilly Media, 2015. — 47 с. 5. Korkmaz N. Practitioners' view on command query responsibility segregation / N. Korkmaz, M. Nilsson // School of Econom
6. Sandoval J. RESTful Java web services: Master core REST concepts and create RESTful web services in Java. Birmingham, U.K : Packt Pub., 2009.
7. MicroservicePremium. martinfowler.com. [Електронний ресурс]: https://Martinfowler.com/bliki/MicroservicePremium.html (дата звернення: 18.11.2023).