МІНІСТЕРСТОВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Факультет кібербезпеки, комп‘ютерної та програмної інженерії

Кафедра комп’ютерних інформаційних технологій

ДОПУСТИТИ ДО ЗАХИСТУ

Завідувач випускової кафедри

____________ Аліна САВЧЕНКО

«____» __________ 2021 р

ДИПЛОМНА РОБОТА

(ПОЯСНЮВАЛЬНА ЗАПИСКА)

ВИПУСКНИКА ОСВІТНЬОГО СТУПЕНЯ

“МАГІСТРА”

ЗА ОСВІТНЬО-ПРОФЕСІЙНОЮ ПРОГРАМОЮ “ІНФОРМАЦІЙНІ УПРАВЛЯЮЧІ СИСТЕМИ ТА ТЕХНОЛОГІЇ”

Тема: “Мікросервісна архітектура у високонавантажених додатках у реальному часі”

Виконавець: Волошин Олександр Олександрович

Керівник: д.т.н., доцент Савченко Аліна Станіславівна

Нормоконтролер: ______________________ Ігор РАЙЧЕВ

Київ – 2021

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Факультет кібербезпеки, комп'ютерної та програмної інженерії

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

«____»_____________2021р.

ЗАВДАННЯ

на виконання дипломної роботи студента

Волошина Олександра Олександровича

(прізвище, ім'я, по батькові)

1. Тема роботи: «Мікросервісна архітектура у високонавантажених додатках у реальному часі» затверджена наказом ректора від 12.10.2021 за № 2228/ст.

2. Термін виконання роботи: з 12.10.2021 по 31.12.2021.

3. Вихідні дані до роботи: теоретичні відомості та основи проектування мікросервісних систем обробки данних у реальному часі, бізнес-вимоги для реалізації програмної логіки для відповідних навантаженнях.

4. Зміст пояснювальної записки: вступ, огляд теоретичної бази побудови репозитарію альтернативних архітектур, функціональна модель програмного комплексу управління репозитарієм патернів компонентів програмних систем, опис та технологія розроблення та використання програмного комплексу управління репозитарієм патернів компонентів програмних систем.

5. Перелік обов'язкового ілюстративного матеріалу: слайди, презентація.

6. Календарний план-графік

№ п/п	Завдання	Термін виконання	Підпис керівника
1.	Отримання завдання на дипломну роботу та побудова плану-графіку виконання робіт.	12.10.2021 – 15.10.2021	
2.	Огляд та аналіз відомих програмних архітектур і патернів.	16.10.2021 – 19.10.2021	
3.	Огляд та створення концепції репозитарію патернів альтернативних архітектур ПС.	20.10.2021 – 24.10.2021	
4.	Проектування програмної системи управління репозитарієм патернів.	25.10.2021 – 31.10.2021	
5.	Написання Розділу 1 дипломної роботи.	01.11.2021 – 07.11.2021	
6.	Розробка та реалізація програмного комплексу управління репозитарієм патернів. Написання Розділу 2 дипломної роботи.	08.11.2021 – 17.11.2021	
7.	Написання Розділу 3 дипломної роботи. Завершення створення пояснювальної записки дипломної роботи.	18.11.2021 – 01.12.2021	
8.	Оформлення та друк пояснювальної записки.	02.12.2021 – 11.12.2021	
9.	Створення презентації, доповіді та підготовка до захисту дипломної роботи	12.12.2021 – 20.12.2021	

7. Дата видачі завдання: 12.10.2021р.

Керівник дипломної роботи ________________ Аліна САВЧЕНКО
(підпис керівника)

Завдання прийняв до виконання ____________ Олександр ВОЛОШИН
(підпис випускниці)

РЕФЕРАТ

Пояснювальна записка до дипломної роботи “Мікросервісна архітектура у високонавантажених додатках у реальному часі” складається із вступу, трьох розділів, загальних висновків, списку використаних джерел і містить 109 сторінок тексту, 28 рисунків та 1 таблиці. Список використаних джерел містить **10** найменувань.

Метою дипломної роботи є розгляд теоретичних та практичних засад створення програмної системи ставок на спорт, яка використовується з метою контролю та моніторингу ставок у реальному часі, на основі наборів високонавантажених патернів і архітектур, що реалізують відповідні модулі програмних систем.

Предметом дослідження є мікросервісна архітектура у високонавантажених проектах, яка містить програмні комплекси, що формують гнучкі для масштабування сервіси для поточної обробки даних у реальному часі.

Об’єктом дослідження є процес моніторингу ставок на спорт для багатьох операторів з різними налаштуваннями.

Ключові слова: МІКРОСЕРВІСНА АРХІТЕКТУРА, БРОКЕР ПОВІДОМЛЕНЬ, КЛАСТЕРИ, КОНТЕЙНЕРИЗАЦІЯ, ІНФОРМАЦІЙНА СИСТЕМА.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ 8
ВСТУП 9
РОЗДІЛ 1. АНАЛІТИЧНИЙ ОГЛЯД І ПОСТАНОВКА ЗАДАЧІ 11
1.1. Особливості реалізації системи моніторингу ставок на спортивні події у реальному часі 11
1.2. Поняття розподіленої системи. Масштабування, види масштабування ... 12
1.3. Куб масштабування 12
1.3.1. Масштабування по осі X 14
1.3.2. Масштабування по осі Y 15
1.3.3. Масштабування по осі Z 15
1.4. Монолітні додатки та їх проблеми, перехід до мікросервісів 17
1.5. Теорема CAP 18
1.5.1. Узгодженність + доступність 19
1.5.2. Узгодженність + роздільність 19
1.5.3. Доступність + роздільність 20
1.6. Перехід від моноліту до мікросервісів 20
1.7. Мікросервісна архітектура, мікросервіси, види їх взаємодії, переваги та недоліки 22
1.7.1. Поняття мікросервіс 22
1.7.2. Характеристики мікросервісів 23
1.7.3. Поділ на компоненти (сервіси) 24
1.7.4. Угрупования з бізнес завдань 24
1.7.5. Розумні сервіси та прості комунікації 25
1.7.6. Децентралізоване зберігання 26
1.7.7. Автоматизація розгортання та моніторингу 26
1.7.8. Обробка помилок 26
1.7.9. Різні типи мікросервісної архітектури 27
1.7.10. Переваги та недоліки мікросервісів 27

1.8. Постановка задачі 29
Висновки до розділу 1 31
РОЗДІЛ 2. ОБҐРУНТУВАННЯ ВИБОРУ ІНСТРУМЕНТІВ РОЗРОБКИ ТА МОВИ ПРОГРАМУВАННЯ 32
2.1. Вибір мови програмування для розробки мікросервісної системи 32
2.1.1. Python 32
2.1.2. Java language 35
2.1.3. Javascript 39
2.2. Вибір брокера повідомлень для системи 44
2.2.1. Apache Kafka 50
2.2.2. RabbitMQ 50
2.2.3. Apache Pulsar 51
2.2.4. Порівняння Apache Pulsar, переваги та недоліки 53
2.3. Вибір клієнтського фреймворку для розробки 59
2.3.1. Angular 59
2.3.2. React.js. 60
2.3.3. Vue.js 61
2.3.4. Порівння та вибір клієнтського фреймворку 62
2.4. Вибір реалізації бази даних для системи 63
2.4.1. MongoDB 63
2.4.2. MySQL 63
2.4.3. Ключові різниці 64
Висновки до розділу 2 74
РОЗДІЛ 3. РОЗРОБКА МІКРОСЕРВІСНОГО ДОДАТКУ ОБРОБКИ ДАННИХ У РЕАЛЬНОМУ ЧАСІ 76
3.1. Загальна концепція роботи додатку 76
3.2. Віртуалізація за допомогою Docker 82
3.3. Розробка додатку 83
3.4. СУБД MongoDB 86

3.5. Оновлення подій у реальному часі за допомогою SSE та MongoDB ChangeStreams. 90
Висновки до розділу 3 93
ВИСНОВКИ 94
СПИСОК БІБЛІОГРАФІЧНИХ ПОСИЛАНЬ ВИКОРИСТАНИХ ДЖЕРЕЛ....94
ДОДАТКИ 99

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ

СУБД – система управління базами даних.

БД – база даних.

ПС – програмна система.

API – програмний інтерфейс додатку.

ORM – об’єктивно-реляційне відображення.

UI – інтерфейс користувача.

DOM – об’єктна модель документа.

ACID – атомарність, узгодженість, ізольованість, довговічність.

HTTP – різновид текстового інтерфейсу користувача й комп’ютера.

UX – досвід користування.

PK – первинний ключ.

FK – зовнішній ключ.

ВСТУП

Створення високонавантаженої системи обробки даних у реальному часі вимагає від проектувальників брати до уваги об’єми данних, які щосекундно будуть проходити через неї, вирішувати питання гнучного масштабування та підтримки додатковими програмними частинами.

Під час розробки високонавантаженої системи часто застосовується мікросервісна архітектура, яка базується на використанні мікросервісів, що забезпечують легкість масштабування відповідних складових частин великої системи.

Архітектура у цій технології проектується на основі вимог до програмної системи (ПС) шляхом обрання відповідних технологій для доставки повідомлень від одного сервісу до іншого за допомогою брокерів повідомлень, API-шлюзів.
Популярність використання мікросервісів почалась з компанії Netflix, які активно просувають цю технологію в маси, а такі гіганти як Amazon, у свою чергу, з 2015 року регулярно випускають у провідних видавництвах книги про мікросервісну архітектуру і навіть проводять кілька регулярних конференцій, повністю присвячених мікросервісам.

Протилежністю мікросервісів є монолітний підхід до розробки. Концепція “моноліту” полягає в тому, що різні компоненти додатка об’єднуються в одну програму на одній платформі. Усі частини програмного забезпечення уніфіковані, і його функції управляються одному місці.

Головна проблема такого підходу була виявлена безліччю нових компаній, таких як Facebook, Uber та Spotify, які прийшли з інноваційними ідеями, агресивною стратегією та рішенням швидко рухати інший спосіб розробки, який би призводив до високого зростання їх додатків.

Подібні компанії помітили, що монолітний підхід не справляється із завданнями розробки швидкодіючого програмного забезпечення обробки даних у реальному часі, а часто зіткнення з проблемою поділу відповідальності між модулями – лише призводило до бажання створити більш релевантну архітектуру.

Потребувався перехід від програми, де все працює в єдиному цілому – до додатку, який можна розділити на кілька блоків та визначити між ними протоколи (правила) взаємодії.

Головною причиною появи мікросервісів стала гостра необхідність розробки продукту кількома командами та дроблення продукту на кілька незалежних елементів. Більше того, ефективність такого підходу мала призвести до того, щоб кожна команда могла розробляти свою частину незалежно від інших абсолютно різними підходами або навіть різними мовами програмування.

Мікросервісна архітектура полягає у розробці програмних додатків шляхом створення окремих незалежних один від одного модулів. Кожен з них відповідає за певне завдання, може бути змінено, доповнено та розширено.

Програма складається з великої кількості сервісів, які взаємодіють між собою за допомогою обміну повідомленнями.

Мікросервісна архітектура поділяє систему на модулі (сервіси) відповідно до запитів бізнесу. Подібні послуги складаються з повного набору технологій, необхідних для конкретного бізнес-запиту: інтерфейс користувача, сховище, зовнішні зв'язки. Окремі модулі можуть бути написані різними мовами, використовувати різні бібліотеки.

Отже, метою дипломного проекту є розробка мікросервісної архітектури у високонавантажених додатках в реальному часі та створення інтерфейсу прикладного програмування. Для досягнення поставленої мети необхідно проаналізувати та вирішити наступні задачі:

- провести аналітичний огляд предметної області;
- визначити критерії оцінки їх ефективності;
- розробити архітектуру для спілкування мікросервісів;
- розробити інфтерфейс прикладного програмування для застосування розробленої технології.

РОЗДІЛ 1. АНАЛІТИЧНИЙ ОГЛЯД І ПОСТАНОВКА ЗАДАЧІ

1.1. Особливості реалізації системи моніторингу ставок на спортивні події у реальному часі

Система моріторингу та налаштування ставок на спортивні події в цілому складається з аналізу та налаштувань відповідних подій (матч Реал Мадрид – Барселона), налаштування маркетів, подій виконання відповідних подій у системі (у Барселоні буде більше 3-х кутових і т.п.), події беруть налаштування за допомогою шаблонів налаштувань.

Особливості реалізації системи залежить від швидкості надходження оновлень подій, для реактивного реагування та економії грошей. Через великий об’єм данних та їх постійне оновлення, існування великої кількості операторів з більшими повноваженнями, налаштування конкретних подій з можливістю аналізувати ризики по виплатам, кількість поставлених ставок та максимальні виплати стосовно відповідних подій.Через велику кількість оновлень та записів у БД, система повинна підтримувати поставлені SLA, які поставив бізнес. Через наявності альтернативних аналогів найбільш важлива є ціна підтримки та доповнення сервісів а також робота системи з багатьма користувачами та оновленнями багатьох подій у системі.

Мережеві затримки та комунікація поміж сервісами, забезпечення роботи з надійністю у 99.9% грають велику роль у реалізації системи. Десять хвилин відмови роботу сервісу під час піків навантаження можуть принести велики збитки для бізнесу. Для того щоб реалізувати реального конкурента альтернативним сервісам конкурентів потрібно правильно розуміти ризики.

1.2. Поняття розподіленої системи. Масштабування, види масштабування, куб масштабування

Розподілена система – це система, яка працює відразу на безлічі машин, що утворюють цілісний кластер. Кластер – це набір комп’ютерів/серверів, об’єднаних мережею, які взаємодіють між собою. Найважливіші плюси такого підходу розробки додатків – велика доступність та відмовостійкість.

Існує 2 вида масштабування: вертикальне та горизонтальне. Вертикальна масштабованість – це нарощування ресурсів хост машини, тобто збільшення кількості ядер, оперативної пам'яті тощо на одному сервері.

Цей підхід дуже простий та зрозумілий, але ресурси не можна нарощувати нескінченно, при додаванні ресурсів (оперативної пам’яті і т.п.) зазвичай доведеться вимикати сервери.

Горизонтальна масштабованість – це додавання нових серверів з більш-менш будь-якими характеристиками до обчислювального кластера. При такому виді масштабованості відсутні проблеми які існують при вертикальній, але горизонтальна масштабованість підтримується не скрізь, а також не всі системи працюють у кластерах, а ті, що в них працюють, зазвичай досить складні в експлуатації.

Відмовостійкою називається система, де відсутня єдина точка відмови, її конфігурацію можна коригувати, підлаштовуючись під відмови. Єдина точка відмови характерна для нерозподілених систем, у яких якщо один сервер відмовить то вся система у цілому не буде працювати.

1.3. Куб масштабування

Існують три ортогональні способи збільшення продуктивності програми:

- Sharding;
- Mirroring;

- Microservices.

Sharding (“шардинг”, “розбиття”) – поділ однотипних, але різних даних на різних серверах. Маючи ключ шардування, можна визначити, що дані, наприклад, які починаються на літери А і Б зберігаються на одному сервері, В і Г – на іншому сервері і т. п. Таким чином, використовуючи правильно налаштований вирівнювач навантаження, можна розділити систему і досягти більш високої продуктивності.

Mirroring («дзеркалювання») — горизонтальне дублювання чи клонування всіх даних. Процес дублювання додатку на абсолютно однакові хости. Таким чином, досягається повне копіювання данних. Це зачасту потрібно для того, щоб відповідати відповідним вимогам по кількості запитів, часу їх очікування і т.п. (SLA).

Microservices (мікросервіси) – розділення функціональності по бізнес-задачам. Кожен сервіс виконуватиме певні завдання.

Куб масштабування (Scale Cube) — це модель для визначення мікросервісів та масштабованих продуктів.

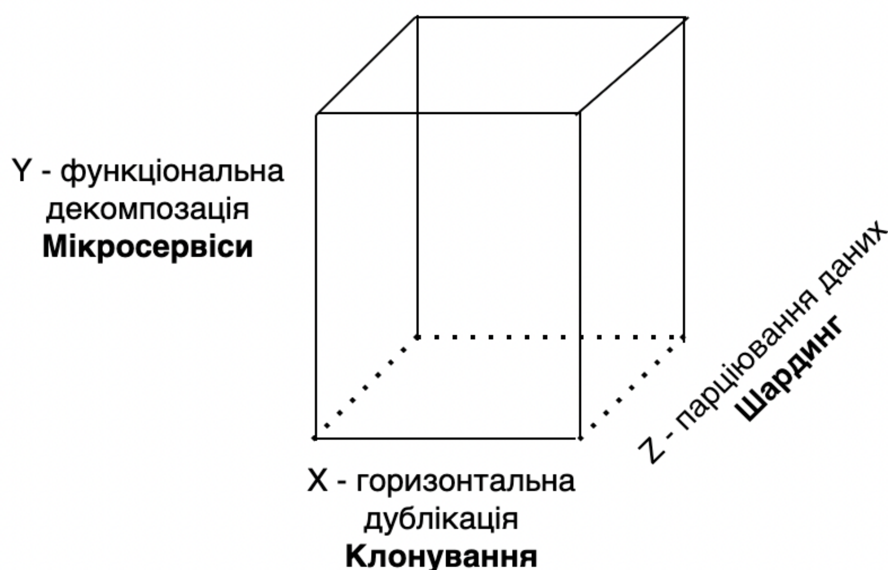


Рис. 1.1. Структура куба масштабування

Куб масштабування складається з 3 осей:

- вісь X: горизонтальне дублювання та клонування сервісів та даних (mirrorring);
- Вісь Y: функціональна декомпозиція та сегментація – мікросервіси(microservices);
- Вісь Z: розділення сервісів і даних на потреби клієнта – поди/шарди (sharding).

У цій трьохвимірній моделі масштабування програми шляхом запуску її клонів які стоять за балансувальником навантаження відоме як масштабування по осі X. Іншими два типи масштабування – це масштабування по осі Y та масштабування по осі Z. Архітектура мікросервісів є прикладом масштабування по осі Y.

1.3.1. Масштабування по осі X

Масштабування по осі X полягає в запуску кількох копій програми за балансирувальником навантаження. Якщо існує N копій, то кожна копія обробляє 1/N навантаження. Це найпростіший і часто використовуваний підхід до масштабування програм.

Одним з недоліків цього підходу є те, що, оскільки кожна копія потенційно отримує доступ до всіх даних, для ефективної роботи та настраювання кешу потрібно більше пам’яті. Інша проблема цього підходу полягає в тому, що він не вирішує проблеми покращення розробки додатку та декомпозиції самої складності проекту.

1.3.2. Масштабування осі Y

На відміну від осі X та Z–осі, які маштабують додаток, запускаючи кілька ідентичних копій програми, масштабування осі Y розбиває програму на кілька різних сервісів. Кожен сервіс відповідає за одну або кілька тісно пов'язаних функцій. Існують різні способи розкладання програми на сервіси. Одним з таких підходів є використання декомпозиції на основі функціонала додатку, створюючи відповідні сервіси, які реалізують відповідні речі, наприклад покупка товару. Інший варіант — розкласти програму за сущністю і створити служби, відповідальні за всі операції, пов'язані з цим об'єктом, наприклад сервіс керування клієнтами. При масштабуванні додаток може користуватися двома варіантами одночасно.

1.3.3. Масштабування по осі Z

При використанні масштабування по осі Z кожен сервер запускає ідентичну копію коду. У цьому плані це схоже на масштабування по осі X. Велика відмінність полягає в тому, що кожен сервер відповідає лише за частину даних. За маршрутизацію кожного запиту на відповідний сервер відповідає певний компонент системи. Одним із часто використовуваних критеріїв маршрутизації є атрибут запиту, наприклад первинний ключ об'єкта, до якого здійснюється доступ. Іншим поширеним критерієм маршрутизації є тип клієнта. Наприклад, програма може надати клієнтам з платною підпискою вищий рівень обслуговування, ніж безкоштовним клієнтам, перенаправляючи їхні запити на інший набір серверів з більшою потужністю.

Розділи по осі Z зазвичай використовуються при масштабуванні баз даних. Дані розподіляються між набором серверів на основі атрибута кожного запису. У цьому прикладі первинний ключ таблиці RESTAURANT використовується для

розподілу рядків між двома різними серверами баз даних. Клонування по осі X може застосовуватися до кожної частини данних шляхом розгортання одного або кількох серверів як реплік/підлеглих. Масштабування по осі Z можна застосувати не тільки до БД а і до програм. Маршрутизатор надсилає кожен запит у відповідний розділ, де він індексується та зберігається. Агрегатор запитів надсилає кожен запит до всіх розділів і об'єднує результати кожного з них.

Масштабування по осі Z має ряд переваг:

- Кожен сервер обробляє лише підмножину даних;
- Данний спосіб полегшує використання кешу та зменшує використання пам'яті та трафік який передається по мережі;
- Покращення масштабованості транзакцій, оскільки запити зазвичай розподіляються між кількома серверами;
- Покращена ізоляція помилок, оскільки збій у роботі додатку робить недоступною лише частину даних.

Масштабування по осі Z має наступні недоліки:

- Підвищена складність програми;
- Зачасту застосовується лише у БД;
- Потрібно реалізувати схему розподілу данних, яка є душе складною, особливо якщо знадобиться у майбутньому повторно розділити дані.

Масштабування по осі Z не вирішує проблеми керування розробкою та складністю програми. Щоб вирішити ці проблеми, потрібно застосувати масштабування по осі Y, що і використовує поняття "мікросервіс".

1.4. Монолітні додатки та їх проблеми, CAP теорема, перехід до мікросервісів

Класичні монолітні додатки складаються з наступної архітектури (рис. 1.2):

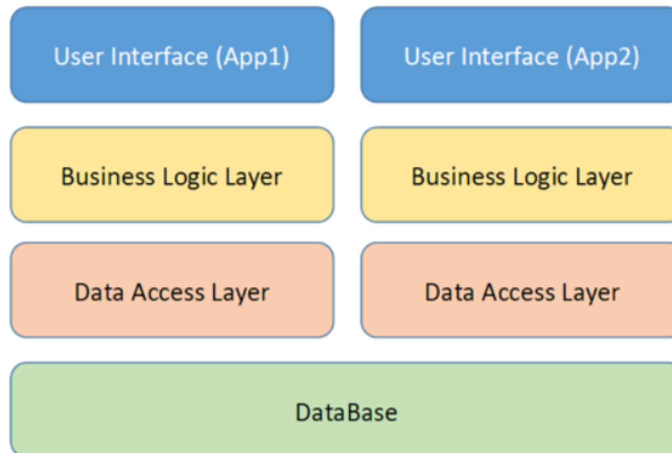


Рис. 1.2. Структура монолітного додатку

Для маленьких і простих програм така архітектура чудово показує себе, але при покращенні програми, додаючи до неї нові сервіси та логіку, або коли інша програма, працює з тими ж даними (наприклад, мобільний клієнт), тоді архітектура програми трохи зміниться.

У міру зростання та розвитку програми, часто можна зіткнутися з проблемами монолітних архітектур:

- складність системи невпинно зростає;
- підтримувати її все складніше та складніше;
- розібратися в ній важко – особливо якщо система переходила з покоління в покоління, логіка забувалась, відбувалася зміна кадрів, а коментарі та тести відсутні;

- багато помилок та мало тестів – моноліт неможливо розібрати і протестувати окремі частини, зазвичай є тільки UI тести, підтримка яких зазвичай займає багато часу;
- велика вартість внесення змін;
- повільний розвиток технологій на проекті з часом.

1.5. Теорема CAP

При масштабуванні системи потрібно забезпечити наступні потреби:

- Доступність системи з різних регіонів;
- Узгодженність данних при умові, що система розподілена;
- Як прискорити систему у N разів.

Рис. 1.3. Трикутник CAP-теореми

Наступні проблеми формулюють CAP-теорему, сформульованої Брюером в 2000 р.

Теорема полягає в тому, що теоретично, не можна забезпечити системі одночасно і узгодженість (consistency), і доступність (availability), і роздільність (partitioning). Тому доводиться жертвувати однією з трьох властивостей на користь двом інших. Аналогічно, як при виборі з «швидко, дешево та якісно» доводиться обирати лише два варіанти. Можнжа розглянути варіанти масштабування, які відповідають теоремі CAP.

1.5.1. Узгодженість + доступність

За такого розкладу дані на всіх серверах узгоджені та доступні. Доступність означає, що гарантується відгук за передбачуваний час. Цей час не обов'язково повинен бути маленьким (можлива хвилина чи більше), але відповідь не більше за відповідний час можна гарантувати. На жаль, при цьому підході жертвується можливістю поділу на секції – відсутня можливість розгорнути 300 таких хостів і розподілити всіх користувачів між цими хостами. Система не буде працювати, так як буде порушена злагодженість транзакцій.

1.5.2. Узгодженість + роздільність

Варіант, коли дані на всіх хостах узгоджені та розподілені на незалежні секції. При цьому ми готові пожертвувати часом, який потрібний на узгодження всіх транзакцій — відповідь буде дуже довга. Це означає, що, якщо два користувача послідовно будуть запитувати одні й самі дані, невідомо, як довго будуть погоджуватися між собою дані для другого користувача.

Така поведінка характерна для монолітів, яким довелося масштабуватися, незважаючи на їх давню розробку.

1.5.3. Доступність + роздільність

Останній варіант – коли система доступна з передбачуваним часом відгуку та розподілена. При цьому нам доведеться відмовитися від цілісності результату — дані більше не є консистентними в кожний момент часу, і серед них з'являються застарілі (від мікросекунд до днів). Але насправді программи завжди оперують старими даними. Маючи трьохшарову монолітну архітектуру з веб-програмою, коли веб-сервер відповідає пакетом з даними, які ви відобразили для користувача, вони вже застаріли. Адже відсутня впевненість у тому, чи не прийшов у цей момент хтось інший і не змінив ці дані. Тож факт, що дані мають кінцеву узгодженість (eventually consistent), — нормально. "Кінцеву узгодженість" означає, що при відсутності на систему перестане зовнішніх впливів, через деякий час вона прийде в узгоджений стан.

Яскравий приклад – класичні DNS-системи, які синхронізуються із затримкою до днів.

Ознайомившись із CAP теоремою, зрозумілі шляхи, по яким можна розвивати систему так, щоб вона була швидкою, доступною та розподіленою, забезпечивши тільки дві властивості з трьох.

1.6. Перехід від моноліту до мікросервісів

При переході від моноліта до мікросервісів треба сегментувати модулі обмежені за бізнес-логікою. При аналізі моноліта, гарним кандидатом на виділення в окремий сервіс є частина моноліту, яка часто потребує змін. Завдяки цьому отримується негайна вигода від виділення сервіса – відсутня потреба у частому

тестуванні моноліта. Гарною практикою є виділяти в окремий сервіс тих програмних частин, що завдають найбільшої кількості проблем або погано працює.

При поділу моноліта на сервіси, варто мати на увагу, як у вас структуровані команди. Емпіричний закон Конвею (Conway's law) стверджує, що структура програми повинна повторювати структуру організації. Якщо організація побудовано на технологічних ієрархіях, побудувати мікросервісну архітектуру буде дуже важко. Важливо виділити feature-команди, які матимуть усі необхідні навички, щоб написати потрібну логіку від початку до кінця.

Рідко можна зустріти чиста мікросервісна архітектура. Найчастіше розробляється щось середнє між монолітом та мікросервісами. Чудовим прикладом буде великий legacy-код, що історично склався, і його розділяють на мікросервіси, відокремлюючи від нього частини. При розробці проекта з нуля, при виборі архітектури треба брати до уваги наступні моменти.

Моноліт краще показує себе у таких випадках:

- Існує нова доменна область і/або немає ніяких знань про цей домен;
- При розробці прототипу або швидкого рішення на певний час;
- Якщо команда не кваліфікована;
- Якщо потрібно просто написати код і забути про нього;
- Грають роль фінансові питання — мікросервіси коштуватимуть дорого.

Мікросервіси обирають, якщо:

- Точно знадобиться лінійне масштабування;
- Ви розумієте бізнес–домен, зможете виділити обмежений контекст і забезпечити узгодженість на бізнес-рівні;

- Команда висококваліфікована, є досвід і пара загублених проектів із мікросервісами в минулому (однак одно з першого разу зробити мікросервіси не виходить);
- Має бути довгострокова співпраця із замовником;
- Достатньо коштів для інвестування в інфраструктуру.

1.7. Мікросервісна архітектура, мікросервіси, види їх взаємодії, переваги та недоліки

Для переходу з моноліту до мікросервісів або для розробки додатку, використовуючи мікросервісну архітектуру, перш за все треба розуміти розуміти бізнес-завдання. Мікросервісами складніше керувати. Не можно розділити код на певні частини і назвати таку архітектуру мікросервісами.

1.7.1. Поняття мікросервіс

Мікросервіси – це архітектурний шаблон, у якому сервіси:

- маленькі (small);
- сфокусовані (focused);
- слабопов'язані (loosely coupled);
- високоузгоджені (highly cohesive).

Один мікросервіс у мікросервісній архітектурі не може розроблятися більш ніж однією командою. Зазвичай одна команда розробляє десь 5 – 6 сервісів. При цьому кожен сервіс вирішує одне конкретне бізнес-завдання, і його здатна зрозуміти одна людина. Якщо ж сервіс не відповідає цим критеріям, його треба розділити на менші. Якщо сервіс невеликий, все набагато простіше. Цей підхід, можна застосовувати окремо, навіть не дотримуючись мікросервісної архітектури загалом.

Фокусування сервісу означає, що сервіс вирішує лише одне бізнес-завдання, і вирішує його добре. Такий сервіс має свою певну ціль у відриві від інших сервісів. Сервіс може праювати як одним так і у з'язці з іншими сервісами.

Сервіс називається слабкозв'язним, коли зміна одного сервісу не потребує змін в іншому. Всі вони пов'язані за допомогою інтерфейсів, через DI та IoC.

Високоузгодженим сервіс називається тоді, коли клас або компонент містить усі необхідні методи вирішення поставленого завдання. Часто виникає питання, чим висока узгодженість (high cohesion) відрізняється від (принципу єдиної відповідальності) SRP? Припустимо, у нас є клас, який відповідає за керування кухнею. У випадку SRP такий клас працює тільки з кухнею і більше ні з чим, але при цьому він може містити не всі методи управління кухнею. У випадку ж високої узгодженості, всі методи управління кухнею містяться тільки в цьому класі. Цю важливу відмінність потрібно розуміти.

1.7.2. Характеристики мікросервісів

Основні характеристики мікросервісів:

- Поділ на компоненти (сервіси);
- Групування по бізнес-завданням;
- Сервіси мають бізнес–сенс;
- Розумні сервіси та прості комунікації між ними;
- Децентралізоване керування;
- Децентралізоване керування даними;
- Автоматизоване розгортання та моніторинг;
- Обробка помилок.

1.7.3. Поділ на компоненти (сервіси)

Компоненти бувають двох видів: бібліотеки та сервіси, які взаємодіють через мережу. Мартін Фаулер визначає компоненти як частини системи, які незалежно можна змінити та розгорнути. Якщо є можливість замінити певний модуль на нову версію - це компонент. А якщо компонент тісно пов'язаний з іншим та їх незалежно замінити не можна (потрібно враховувати контракти, версії) – вони разом утворюють один компонент. Якщо частину додатку не можна розгорнути незалежно, і потрібна логіка зі стороннього сервісу – це також не компонент.

1.7.4. Угруповання з бізнес-завдань (сервіси мають бізнес-сенс)

Стандартне компонування моноліту виглядає наступним чином:

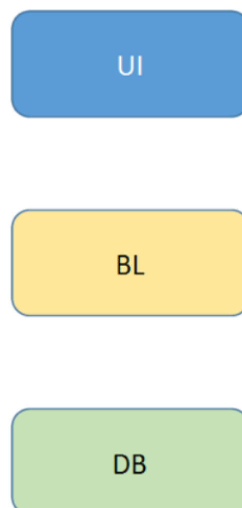


Рис. 1.4. Архітектура монолітного додатку

Для підвищення ефективності розробки часто потрібно ділити за цими шарами і команди. Існує команда, яка займається UI, команда, яка займається ядром додатку,

і є команда, яка розбирається з БД. При переході до мікросервісної архітектури, сервіси та команди діляться за бізнес–задачами:

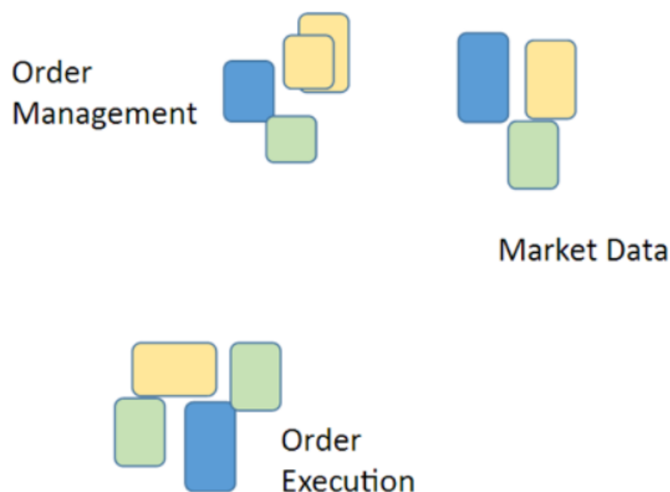


Рис. 1.5. Мікросервісна організація структури додатку

Наприклад, може бути група, яка займається управлінням замовленнями, — вона може обробляти транзакції, робити по ним звіти тощо. Така група займатиметься і відповідними БД, і відповідною логікою, і, можливо, навіть UI. Хоча найчастіше UI залишається монолітним, але присутня можливість розбивання UI за допомогою мікрофронтенд.

1.7.5. Розумні сервіси та прості комунікації

Існують різні варіанти взаємодії сервісів. Інколи користуються розумноб шиною, яка знає і про роутинг, і про бізнес-правила (BizTalk), і до сервісів прилітають вже готові об'єкти. У цьому випадку маємо “розумний” middleware та “дурні” endpoint-и. Це є антишаблоном. Як показав час (на прикладі того ж інтернету), у нас дуже просте і невигадливе середовище передачі даних — їй абсолютно байдуже, що ви передаєте, середовище передачі нічого не знає про бізнес. Бізнес-логіка знаходиться у сервісах. Це важливо розуміти. Якщо ж всю

логіку розміщати у середі передачі буде “розумний” моноліт і “тупі” сервіси-обгортки баз даних.

1.7.6. Децентралізоване зберігання

З погляду сервісно-орієнтованих архітектур і, зокрема, мікросервісів, децентралізоване зберігання є дуже важливим моментом. Децентралізоване зберігання означає, кожен сервіс має власну і лише свою БД. Єдиний випадок, коли різні служби можуть використовувати одне сховище, якщо ці служби являють собою точні копії один одного. Бази даних одна з одною не взаємодіють:

Єдиний варіант взаємодії – мережна взаємодія між сервісами. Middleware може бути різним, виявлення сервісів та взаємодія між ними може відбуватися просто безпосередньо через виклик RPC, через ESB або MessageBrokers.

1.7.7. Автоматизація розгортання та моніторингу

Автоматизація розгортання та моніторингу одна з основних особливостей мікросервісної архітектури. Маючи справу з мікросервісами, обов’язково знадобиться автоматичне розгортання, безперервна інтеграція та постачання. Також потрібний безперервний моніторинг, інакше буде відсутня можливість слідкувати за численними сервісами. Корисно використовувати централізоване логування. На ринку є хороші готові рішення на кшталт ELK або Amazon CloudWatch.

1.7.8. Обробка помилок

Починаючи будувати мікросервісну архітектуру, потрібно розраховувати варіанти, що сервіси інколи можуть бути недоступними. Іншими словами, сервіс

повинен бути готовий, що відповідь на запит ніколи не прийде, якщо він чекає на якісь дані.

Компанія Netflix розробила Chaos Monkey — інструмент, який ламає сервіси, хаотично вимикає їх, руйнуючи з’єднання. Це потрібно, щоб оцінити надійність системи.

1.7.9. Різні типи мікросервісної архітектури

Припустимо, у нас є UI, API Gateway і десяток сервісів, які стоять за ним. Зазвичай послуги якось взаємопов'язані. Існують три способи зв'язати сервіси:

- Service Discovery (RPC Style) – сервіси знають один про одного і спілкуються безпосередньо;
- Message Bus (Event–driven) — якщо ви використовуєте шаблон «видавець–передплатник». Ні «підписник» не знає тих, хто на нього підписаний, ні «видавець» не знає, звідки надходять данні. Вони зацікавлені лише у данних певного типу – вони підписуються на повідомлення. Це і називається message–driven–або event–driven–архітектура;

Hybrid – змішаний варіант, коли для одних випадків застосовується RPC, а для інших – message bus.

1.7.10. Переваги та недоліки мікросервісів

Мікросервісний підхід можна застосувати майже у будь-якій предметній сфері. Можна розгортати сервіси навіть у межах одного домену програми. Будуть створені ізольовані модулі, які взаємодіють за допомогою обміну крізь пам’ять. Система буде працювати швидко, через відстуність мережевих затримок та інших мережевих витрат.

Основні переваги:

- Чіткий поділ на модулі. Дозволить посилити модульну структуру, кожний модуль мікросервісної архітектури робить одну справу і робить її добре;
- Висока доступність. Деякі сервіси можуть не працювати, при цьому у цілому система буде доступною але буде працювати без відповідних сервісів;
- Різноманітність технологій і можливість використовувати правильний інструмент для відповідного завдання. При потребі у побудові сховища даних, можна підібрати той інструмент, який дійсно вміє зберігати великі обсяги даних і швидко опрацьовувати їх. Мікросервіси дозволяють випробувати технологію на відповідному сервісі, не впливаючи на інші сервіси, оскільки контракт ізольований через мережеву взаємодію;
- Незалежне розгортання через слабку зв'язаність сервісів: прості сервіси простіше розгортати, з меншою ймовірністю відмови системи.

Основні недоліки при використанні мікросервісної архітектури:

- Складність розробки;
- Кінцева узгодженість – бізнес повинен дозволяти працювати з відстроченими даними. Цим доведеться платити за високу доступність.
- Класичний приклад кінцевої узгодженості — банківські картки, транзакції між якими можуть займати три дні. Через це є ймовірність перевищення кредитного ліміту, і тоді починає діяти найпростіший механізм компенсації – вам дзвонять із банку з проханням погасити перевищення.

Складність операційної підтримки – необхідність у кваліфікованих DevOps-інженерах, безперервному розгортанні та автоматичному моніторингу

1.8. Постановка задачі

Враховуючи вищезазначені факти актуальною є задача розробки мікросервісної системи обробки данних у реальному часі. За допомогою програмного продукту можна моніторити та контролювати спортивні події та корегувати виплати стосовно ставок на спортивні події.

Під час та до проведення спортивної події можна створити та налаштувати виплати та виграши для відповідних подій, які можуть статися під час або до спортивної події. Сервіс повинен підтримувати налаштування за замовчуваням для багатьох операторів, кожен з який повинен успадковувати глобальні налаштування з глобального оператора з можливістю сконфігурувати для відповідного користувача або повернутися до глобальних налаштувань.

Сервіс повинен підтримувати гнучке масштабування та швидку реплікацію відповідних сервісів. Під час аналізу ставок на спортивні події можна оцінити ризики по ним. Щоб запобігти втраті великої кількості грошей по ризикованим ставкам, доцільно розробити систему яка буде мати можливість конфігурувати виплати для відповідних користувачів та динамічно вимикати події під час спортивної гри, для того щоб заборонити користувачам ставити гроші на відповідну подію.

Враховуючи наведене вище, актуальною є розробка програмної системи ставок на спорт, яка використовується з метою контролю та моніторингу ставок у реальному часі, на основі наборів високонавантажених патернів і архітектур, що реалізують відповідні модулі програмних систем.

Таким чином, метою дипломної роботи є розробка мікросервісної архітектури у високонавантажених додатках в реальному часі та створення інтерфейсу прикладного програмування. Для досягнення поставленої мети необхідно проаналізувати та вирішити наступні задачі:

- провести аналітичний огляд предметної області;
- визначити критерії оцінки їх ефективності;
- розробити архітектуру для спілкування мікросервісів;
- розробити інфтерфейс прикладного програмування для застосування розробленої технології.

Функціонал запропонованого програмного продукту реалізує такі можливості:

- Налаштування виплат і максимально допустимих ставок;
- вирішення проблем з масштабуванням “вузьких місць”;
- Заморозка події у відповідних спортивник подіях для припинення ставок на них;
- Можливість комперсувати налаштування для економії пам’яті та видалення непотрібних данних з БД;
- Відображення інформації стосовно спортивної події у реальному часі з мінімальною затримкою.

Перевагами проектованої системи є висока ефективність за рахунок:

- зменшення до мінімуму мережевих затримок під час оновлення даних;
- підвищення точності отримуваних данних, оскільки запропонований сервіс співпрацює з довіреними вендорами;
- динамічне та гнучке розгортання;
- швидка реплікація сервісів у “вузьких місцях”.

Висновки до розділу 1

Розподілена система – це система, яка працює відразу на безлічі машин, що утворюють цілісний кластер. Кластер – це набір комп'ютерів/серверів, об'єднаних мережею, які взаємодіють між собою. Найважливіші плюси такого підходу розробки додатків – велика доступність та відмовостійкість. Можливості масштабування можна реалізувати за допомогою "Куба масштабованості".

Основні переваги мікросервісної архітектури:

- Чіткий поділ на модулі, мікросервіси посилюють модульну структуру;
- Висока доступність. Деякі сервіси можуть не працювати, при цьому у цілому система буде доступною але буде працювати без відповідних сервісів;
- Можливість використовувати правильний інструмент для відповідного завдання;
- Незалежне розгортання через слабку зв'язаність сервісів.

Мікросервісна архітектура у високонавантажених додатках у реальному часі реалізує бізнес-логіку для управління ставками на спортивні події, а також контролює налаштування та ризики для операторів. Більше того, програмний продукт повинен мати можливість швидко масштабуватися та забезпечувати цілісність та точність даних. Система має доступ, і дозволяє користувачеві безпосередньо вносити зміни в деякі з налаштувань, як, наприклад, виплата по спортивній події, аналіз ризиків і заморозка ризикованих подій.

Перевагами проектованої системи є висока ефективність за рахунок:

- зменшення до мінімуму мережевих затримок під час оновлення даних;
- підвищення точності отримуваних данних, оскільки запропонований сервіс співпрацює з довіреними вендорами;
- динамічне та гнучке розгортання;
- швидка реплікація сервісів у "вузьких місцях".

РОЗДІЛ 2
ОБҐРУНТУВАННЯ ВИБОРУ ІНСТРУМЕНТІВ РОЗРОБКИ ТА МОВИ ПРОГРАМУВАННЯ

2.1. Вибір мови програмування для розробки мікросервісної системи

2.1.1. Python

Python — це високорівнева, інтерпретована, інтерактивна й об'єктно-орієнтована мова сценаріїв. Python розроблений так, щоб його можна було легко читати. Він часто використовує ключові слова англійською мовою, він має менше синтаксичних конструкцій, ніж інші мови. Python обробляється під час виконання інтерпретатором. Вам не потрібно компілювати вашу програму перед її виконанням. Це схоже на PERL і PHP.

Python підтримує об’єктно-орієнтований стиль або техніку програмування, що інкапсулює код в об’єкти. Python – чудова мова для програмістів-початківців і підтримує розробку широкого спектру додатків від простої обробки тексту до WWW–браузерів до ігор.

Його високорівневі вбудовані структури даних у поєднанні з динамічним типізацією та динамічним зв’язуванням роблять його чудовим вибором для швидкої розробки додатків. Простий, легкий у засвоєнні синтаксис Python підкреслює читабельність і, отже, знижує витрати на обслуговування програми. Python підтримує модулі та пакети, що сприяє модульності програм і повторному використанню коду. Інтерпретатор Python і його стандартне API доступні у вихідній або двійковій формі безкоштовно для всіх основних платформ і можуть вільно поширюватися. Програмісти часто обирають Python через високу продуктивність, яку він забезпечує. Оскільки відсутній етап компіляції, цикл редагування-тестування-налагодження неймовірно швидкий.

Основні особливості та функції Python

Особливості Python Функції Python включають − Легкий у навчанні — Python легкий у навчанні і має просту структуру та чітко визначений синтаксис. Це дозволяє швидко опанувати мову. Код на мові Python зрозумілий навіть дитині. Вихідний код на мові Python досить простий в обслуговуванні. Велика частина бібліотеки Python портативна і сумісна між платформами на таких ОС як UNIX, Windows і Macintosh. Python підтримує інтерактивний режим, який дозволяє інтерактивне тестування та налагодження фрагментів коду. Python може працювати на широкому спектрі апаратних платформ і має однаковий програмний інтерфейс на всіх платформах. Ви можете додавати модулі низького рівня до інтерпретатора Python. Ці модулі дозволяють програмістам додавати або налаштовувати свої інструменти для підвищення ефективності програмного коду. Також Python надає прикладні інтерфейси до всіх основних комерційних баз даних (як реляційних так і нереляційних).

Порівняння Python з Java

- Python не портребує крапку з комою та фігурні дужки у програмі в порівнянні з Java, яка згенерує помилку синтаксису у разі коли крапка з комою або фігурні дужки будуть відсутніми;
- Код Python вимагає менше рядків коду, ніж Java, для написання программи;
- Python динамічно типізована мова програмування, це означає що тип значення змінної присвоюється лише під час виконання коду, інтерпретатор Python сам виявить тип даних у порівнянні з мовою Java, де потрібно явно задати тип даних;
- Python підтримує різні типи моделей програмування: імперативне, об'єктно-орієнтоване та процедурне програмування, порівняно з мовою

Java, яка повністю базується на моделях програмування на основі об'єктів і класів;

- Python легко читається, що корисно для початківців, які з нетерпінням чекають швидкого розуміння основ програмування в порівнянні з Java, яка має стрімку криву навчання через заздалегідь визначені складний синтаксис програми;
- Стислий та лаконічний синтаксис Python робить його набагато кращим варіантом для людей інших дисциплін, які хочуть використовувати мову програмування для аналізу даних, нейронної обробки, машинного навчання або статистичного аналізу в порівнянні з синтаксисом Java, який важкий для читання та розуміння;
- Python безкоштовний і з відкритим вихідним кодом означає, що його код безкоштовний для комерційних цілей, порівняно з Java, яка може вимагати ліцензії для використання для великомасштабної розробки додатків;
- Код Python вимагає менше ресурсів для запуску, оскільки він безпосередньо компілюється в машинний код у порівнянні з Java, який спочатку компілюється в байтовий код, а потім його потрібно скомпілювати в машинний код за допомогою віртуальної машини Java (JVM).

Порівнянн Python з Node.js

- Node.js найкраще підходить для асинхронного програмування та є ідеальною платформою, доступною зараз для роботи з веб–додатками в режимі реального часу. Python погано справляється з виликонавантажуваними додатками реального часу;

- Node.js використовує мову JavaScript. Найбільша перевага використання Python полягає в тому, що розробникам потрібно писати менше рядків коду;
- Node.js можна рекомендувати для великих проектів. Він ідеально підходить для великого проекту, оскільки може маштабуватися у кластер та використовувати багатоботочну обробку;
- Node.js найкраще підходить для середніх проектів. Python підходить для розробки невеликих проектів;
- Node.js є найкращим вибором, якщо наголос стоїть на розробці веб–додатка або веб–сайта. Python є ідеальною платформою для чисельних обчислень, машинного навчання та мережевого програмування.

2.1.2. Java language

Java - мультифункціональна об'єктно-орієнтована мова зі строгою типізацією. Сувора (сильна) типізація не дозволяє підставляти у виразах різні типи даних та дозволяє автоматично неявні перетворення, що додає мороки: якісь частини доводиться прописувати самому, в обмін на це збільшується надійність програми.

Java – це мова, створена за моделлю об'єктно-орієнтованого програмування. У ній існують класи та об'єкти. Класи – це типи даних, а об'єкти – представники класів. Программість сам декларує їх, називає та надає їм властивості та операції, які з ними можна виконувати. Це як конструктор, який дозволяє збудувати те, що ви хочете. Саме через об’єктру модель переважно і програмують на Java.

Використання Java

Java використовується у багатьох сферах. Найчастіше Java використовується у веб-розробці та додатках для Android, але і в інших сферах вона теж дуже популярна. На ній пишуть:

- додатки для Android;
- десктопні, промислові, банківські програми;
- програми для роботи з Big Data;
- веб-програми, веб–сервера, сервера додатків;
- вбудовані системи – від програмування дрібних чіпів до спеціальних комп'ютерів;
- корпоративний софт.

Переваги та недоліки Java

Переваги:

- Незалежність – код буде працювати на будь-якій платформі, яка підтримує Java;
- Надійність — чималою мірою досягається завдяки строгій статичній типізації;
- Мультифункціональність;
- Порівняно простий синтаксис;
- Java – основна мова для Android-розробки;
- Об'єктно-орієнтоване програмування (ООП) теж приносить багато переваг: паралельна технологія обробки, гнучкість розробки, повторне використання, код добре організований, та його легше підтримувати.

Недоліки:

- Низька швидкість (проти C та C++);
- Потребує багато пам'яті;
- Немає підтримки низькорівневого програмування (Java – високорівнева мова). Відсутня підтримка вказівників;
- З 2019 року оновлення для бізнесу та комерційного використання стали платними;
- Для ООП потрібний досвід, а планування нової програми займає багато часу;
- Автоматичне складання сміття (Garbage collection): з одного боку це звільняє інженера від потреби самостійно виділяти та чистити пам'ять, але з іншого боку, розробник не може контролювати процес, хоча іноді це важливо.

Java має статичну типізацію: ви повинні прописувати тип даних, коли вводите нову змінну. Також Java має комплексний код, з безліччю слів і знаків: англійською такий синтаксис називають словом «verbose», тобто «балакучий» код, багатослівний. Він гірше читається і може бути складним для новачків, хоча багато розробників почуваються комфортніше зі строгим синтаксисом.

Порівняння Java з Python:

- Python не є чисто об'єктно-орієнтованою мовою на відміну від Java;
- Python є інтерпретованою мовою, тоді як Java – це компільована мова;
- Python простий у використанні, тоді як Java не така проста;
- Програми на Python набагато коротші, ніж програми на Java;

- Python підтримує динамічний типізацію, що дуже корисно для програмістів, оскільки їм потрібно писати менше коду, завдяки чому економиться їх час і який зручний для користувача, а також для програміста. Java вимагає від розробника визначити тип кожної змінної перед її використанням, що забирає багато часу програміста;
- Багато великих організацій, таких як Google, Yahoo, NASA тощо, використовують Python, але зазвичай програми на Python працюють набагато повільніше, ніж програми на Java;
- Java має набагато більшу екосистему бібліотек на всі випадки використання, ніж Python;
- Java використовується для великих додатків, Python зачасту для машинного навчання.

Порівняння Java з JavaScript:

- Java є суворо типізованою мовою, і змінна повинна бути оголошена першою для використання в програмі. У Java тип змінної перевіряється під час компіляції. JavaScript — це слаботипізована мова і має більш розслаблений синтаксис і правила;
- Java - це об'єктно-орієнтована мова програмування. JavaScript – це мова сценаріїв на основі об'єктів;
- Програми Java можуть працювати на будь-якій віртуальній машині (JVM) або в браузері. Раніше код JavaScript працював лише у браузері, але тепер він може працювати на сервері через Node.js;

- Об’єкти Java базуються на класах, навіть ми не можемо створити жодну програму на java без створення класу. Об'єкти JavaScript базуються на прототипах;
- Програма Java має розширення файлу “Java” і переводить вихідний код у байт–код, який виконується JVM (віртуальна машина Java). Файл JavaScript має розширення “.js”, і він інтерпретується, але не компілюється, кожен браузер має інтерпретатор Javascript для виконання JS-коду;
- Java є окремою мовою. міститься на веб-сторінці та інтегрується з її вмістом HTML;
- Java має потоковий підхід до паралельності. Javascript має підхід на основі подій до паралельності;
- Java підтримує багатопотоковість. Javascript не підтримує багатопотоковість.

2.1.3. Javascript

JavaScript — це динамічно типізована однопоточна інтерпретована мова для веб-розробки.

Динамічна типізація мов означає, що змінна може містити будь-який тип даних, наприклад String або Number, протягом свого життя, і інтерпретатор JavaScript не буде скаржитися на це. Мова однопоточна, що означає, що ваш код JavaScript виконується синхронно або послідовно рядок за рядком, з додатковими можливостями роботи багатопоточно. Мова інтерпретована, що означає відсутність у потребі в компіляції.

Кожен браузер постачає інтерпретатор JavaScript, який також називається JavaScript Engine. V8 — це рушій JavaScript, розроблений Google і використовується

у браузері Google Chrome, SpiderMonkey — це рушій JavaScript, розроблений Mozilla для свого браузера Firefox.

JavaScript – це дуже проста мова для вивчення. JavaScript дуже добре підтримує як парадигму ООП так і може використовуватися у функціональному програмуванні. JavaScript є інтерпретованою мовою. Це означає, що нам не потрібно компілювати вихідний код JavaScript, перш ніж відправити його в браузер. Інтерпретатор може взяти необроблений код JavaScript і запустити його для вас.

Історія мови JavaScript

У перші дні Інтернету веб-браузери використовувалися для відображення статичних сторінок. Зазвичай ці сторінки були неінтерактивними. Щоб додати певну взаємодію, у 1995 році Брендан Айх ввів нову мову в браузер Netscape. Цією новою мовою був JavaScript (раніше називався LiveScript).

У нього були і інші конкуренти, такі як ActionScript, Silverlight та Flash, але JavaScript переміг. JavaScript був розроблений без урахування продуктивності. Його головною ціллю було працювати всередині браузера і надавати API для роботи з DOM. Але оскільки багато браузерів намагалися прийняти його по-своєму, його довелося стандартизувати.

Ecma International — це організація зі стандартів, яка стандартизує JavaScript, а Технічний комітет, яки керує цим стандартом. Цей стандарт відомий як EcmaScript. EcmaScript також використовується як синонім до JavaScript, оскільки торгова марка JavaScript належить Oracle Corporation.

Node.js – серверний JavaScript

JavaScript – це однопотокова мова, вона знає, як виконувати речі по черзі. Він не може виконувати асинхронні завдання або запускати код JavaScript у кількох потоках для ефективності.

Використовуючи JavaScript, не можна напряму отримати доступ до комп'ютера, на якому він працює, до файлової системи, мережі тощо. Стандарт ECMAScript не має специфікацій для цього.

Тому постачальники браузерів мають розширити механізм JavaScript за допомогою API, які можуть робити інші речі. Наприклад, DOM API відповідає за рендер HTML-коду в фактичні пікселі на екрані. XMLHttpRequest API дає нам можливість надсилати мережеві запити для отримання даних з віддаленого сервера у фоновому режимі.

Ці види API відповідають за виконання інших операцій, для виконаняня яких JavaScript не призначений. Ці API надаються браузером і називаються Web API. Ці API написані мовами низького рівня, такими як C або C++, але їхній інтерфейс доступний через JavaScript.Ці веб–API іноді виконують свою роботу в окремому потоці, дозволяючи іншим кодам JavaScript працювати нормально, коли завдання виконується у фоновому режимі. Після завершення роботи він інформує основний потік JavaScript. JavaScript необхідний не лише у браузері, можна використати рушій V8 JavaScript і встановити його на свій комп'ютер.

Концепція серверного JavaScript походить від цієї простої ідеї. Ви можете взяти будь-який механізм JavaScript, загорнути в програму, яка надає чистий інтерфейс, щоб взяти код JavaScript користувача та виконати його в механізмі JavaScript. Ви також можете надати API для виконання таких операцій, як введення файлової системи, мережа тощо, які не підтримуються на пряму рушієм JavaScript.

Архітектура Node.js

Node.js — це фреймворк, який “під капотом” містить рушій JavaScript V8, стандартну бібліотеку пакетів і деякі двійкові файли.

Як і веб-API у браузері, Node.js має стандартну бібліотеку, яка містить пакунки JavaScript, яка також може надавати інтерфейс для низькорівневих API. Наприклад, Node.js fs, модуль містить функцію readFile. Ця функція читає файл на диску.

Більшість із цих пакетів містить код, написаний на мові програмування низького рівня, для зв’язку з пристроєм, наприклад для доступу до файлової системи. Ці пакети експортують функції JavaScript та інші типи для виконання цього коду. Оскільки JavaScript не може спілкуватися з C++ чи якоюсь іншою мовою, Node.js має створити прив’язку, щоб полегшити це спілкування. Процес створення таких пакетів дуже складний. Node.js використовує різні потоки для виконання низькорівневих операцій, що не займають час. Таким чином, наш JavaScript не блокується, поки виконується така довготривала операція, як читання файлу. Оскільки ці операції виконуються у фоновому режимі після початку, нам потрібно підтвердження або функцію зворотнього виклику після завершення операції. Функція зворотнього виклику є функцією JavaScript, яка буде виконуватися, як тільки операція буде завершена.

Архітектура Node.js дуже складна і складається з різних частин, як показано на попередній діаграмі. Він також містить цикл подій, який полегшує виконання цих функцій зворотного виклику. Ви повинні подивитися наведене нижче відео про цикл подій, хоча воно знаходиться в контексті браузера, але все дуже схоже і в Node.js. Це розвіє ваші сумніви, що залишилися.

Вбудоване API

Node.js постачається з колекцією вбудованих пакетів, які називаються стандартною бібліотекою вузлів. Ці пакети необхідні для виконання низькорівневих операцій, таких як введення-виведення файлової системи та мережа. Нам не потрібно встановлювати їх за допомогою NPM.

Оскільки ці пакунки містять код низькорівневою мовою програмування, пристосований до певної версії Node.js, вони повинні бути відправлені як частина процесу встановлення. Ось список вбудованих модулів у Node.js.

JavaScript в порівнянні з Java:

- Java використовує статичну типізацію, JavaScript — динамічну;
- Java розроблена для підтримки великих програм із багатьма оточеннями;
- JavaScript гарно справляється з розподіленими системами;
- Java в основному обмежена сервером. JavaScript може працювати як на клієнті так і на сервері за допомогою технології серверного Javascript Node.js;
- Java базуэться на ООП падигмі, тоді як Javascript дозволяє писати програми як у ООП стилі так і у функціональному.

JavaScript у порівнянні з Python:

- Python використовується для невеликих проектів, Node.js у розподілених, масштабованих системах;
- Серверний Javascript багатопоточний, на відміну від Python–а;
- Код JavaScript дає можливість писати у повністю функціональному стилі, тоді як Python підтримує лише процедурну та ООП парадигму;

- Екосистема Python зосереджені на обробці та аналізі даних, тоді як бібліотек JavaScript для клієнтського та серверної розробки.

2.2. Вибір брокера повідомлень для системи.

Брокер повідомлень – це технологія, що забезпечує зв'язок між додатками та допомагає створити загальний механізм інтеграції для підтримки хмарних, мікросервісних, безсерверних та гібридних архітектур.

Брокер повідомлень являє собою програмне забезпечення для зв'язку між програмами, системами та службами, яке допомагає їм обмінюватися інформацією один з одним. Це робиться через переводу повідомлень з одного формального протоколу обміну повідомленнями в інший. Таким чином, незалежні служби можуть «спілкуватися» між собою безпосередньо, навіть якщо вони написані різними мовами або реалізовані на різних платформах.

Брокери повідомлень — це програмні модулі орієнтованого на обмін повідомленнями. Проміжне програмне забезпечення такого типу надає стандартизовані засоби обробки потоку даних між компонентами програми, дозволяючи розробникам не відволікатися на це і займатися основною логікою програм. Також брокер може виступати як розподілений рівень зв'язку, що дозволяє додаткам, що охоплюють декілька платформ, взаємодіяти один з одним.

Брокери повідомлень перевіряють, зберігають, маршрутизують повідомлення та доставляють їх у місце призначення. Вони виступають як брокери між різними додатками: для відправлення повідомлень відправникам необов'язково знати, де знаходяться одержувачі, чи активні вони і скільки їх всього. Це спрощує поділ процесів та послуг усередині систем.

Щоб забезпечити надійне зберігання повідомлень та їх гарантовану доставку, у брокерах користуються чергами повідомлень — допоміжною структурою або компонентом, який зберігає та впорядковує повідомлення, доки вони не будуть оброблені додатками-отримувачами. Повідомлення в черзі зберігаються в тому ж порядку, в якому вони були передані, доки не буде підтверджено їх отримання.

Брокери повідомлень працюють за принципом асинхронного обміну повідомленнями. Ця модель запобігає втраті цінних даних та підтримує функціонування систем навіть при нестабільному зв'язку або високих затримках, властивих загальнодоступним мережам. При асинхронному обміні повідомленнями повідомлення будуть доставлені один (і лише один) раз у тому порядку, в якому вони були відправлені.

До брокерів повідомлень можуть входити адміністратори черг для взаємодії між кількома чергами, а також інструменти маршрутизації даних, перекладу повідомлень, зберігання та керування станом клієнта.

Моделі брокерів повідомлень

Брокери повідомлень працюють за двома основними шаблонами (стилями) обміну повідомленнями:

- **Двуточний обмін повідомленнями:** цей шаблон використовується в чергах повідомлень із прямим взаємозв'язком між відправником та одержувачем. Кожне повідомлення у черзі надсилається лише одному одержувачу, який отримує його лише один раз. Двоточковий обмін повідомленнями передбачає лише одноразову обробку повідомлення. Прикладом застосування такого стилю може бути обробка платежів та фінансових транзакцій. У таких

системах і відправнику, і одержувачу потрібна гарантія, що кожен платіж буде відправлено один раз;

- **Модель видавець/передплатник:** цей шаблон обміну повідомленнями часто скорочується до pub/sub. Автор кожного повідомлення публікує його у відповідну тему, а одержувачі (яких може бути кілька) підписуються на відповідні теми, з яких хочуть отримувати повідомлення. Всі опубліковані в темі повідомлення надсилаються у всі підписані на цю тему програми. Цей обмін схожий на широкомовлення, коли між видавцем та одержувачами повідомлення встановлений зв'язок «один до багатьох». Якщо авіакомпанія розсилатиме новини про час посадки або терміни затримки рейсів, то цією інформацією могли б користуватися кілька сторін: наземні бригади, які виконують технічне обслуговування та заправку літаків, працівники багажних служб, бортпровідники та пілоти, що готуються до наступного рейсу, та оператори для інформування пасажирів Для таких ситуацій найкраще підходить стиль видавець/передплатник.

Брокери повідомлень у хмарних архітектурах

Хмарні програми дозволяють реалізувати переваги, властиві хмарним обчисленням: це гнучкість, масштабованість та їх швидке розгортання. Ці програми складаються з окремих невеликих, багаторазово використовуваних компонентів, які називаються мікросервісами. Кожен мікросервіс розгортається та працює незалежно від інших. Тобто будь-який з них можна оновлювати, масштабувати або перезапускати, не торкаючись інших сервісів у системі. Мікросервіси часто упаковуються в контейнери і працюють як єдине ціле у складі програми, однак у

кожного з них може бути свій, відмінний від інших, стек технологій, що включає базу даних та модель управління даними.

Щоб працювати у зв'язці з іншими компонентами, мікросервісам потрібні засоби обміну інформацією. Один із механізмів для створення спільної магістралі обміну повідомленнями – брокери повідомлень.

Брокери повідомлень часто використовуються для керування обміном повідомленнями між локальними системами та хмарними компонентами в гібридних хмарних середовищах. Використання брокера повідомлень надає більше контролю над зв'язком між службами та гарантує, що дані будуть пересилатися між компонентами програми безпечно, надійно та ефективно. Аналогічну роль брокери можуть грати в інтеграції мультихмарних середовищ, забезпечуючи зв'язок між завданнями та середовищами виконання, що працюють на різних платформах. Також ця технологія відмінно підходить для безсерверних обчислень, які мають на увазі запуск окремих хмарних служб за попереднім запитом.

Брокери повідомлень та API

Для зв'язку між мікросервісами зазвичай використовують REST API. Вони визначають набір принципів та обмежень, що враховуються розробниками під час розробки веб-служб. Будь-які служби, створені з урахуванням цих контрактів, згодом зможуть взаємодіяти між собою за допомогою набору уніфікованих спільних операторів та запитів. Програмний інтерфейс програм (API) вказує базовий код, який за умови відповідності правилам REST забезпечує взаємодію служб один з одним.

REST API взаємодіють за протоколом HTTP. Оскільки HTTP є стандартним транспортним інтернет-протоколом, REST API широко відомі, часто

використовуються та відмінно поєднуються один з одним. Але оскільки HTTP — це протокол запиту/відповіді, він найкраще підходить для ситуацій, коли потрібен асинхронний запит/ответ. Це означає, що служби, які надсилають запити через REST API, повинні розроблятися з розрахунком на негайну відповідь. Якщо клієнт не зможе отримати відповідь, то сервіс, що надіслав запит, буде заблокований в очікуванні відповіді на запит. Також в обидві служби має бути вбудована логіка обробки помилок.

Брокери повідомлень забезпечують асинхронний обмін повідомленнями між службами, при якому відправнику не потрібно чекати на відповідь одержувача. Це підвищує стійкість до відмов і надійність систем, в яких реалізовані такі служби. Також використання брокерів повідомлень спрощує масштабування систем, оскільки шаблон видавець/передплатник легко допускає зміну числа служб. Крім цього, брокери повідомлень відстежують стан отримувача повідомлення.

Брокери повідомлень та платформи потокової обробки подій

На відміну від брокерів повідомлень, які підтримують два або кілька шаблонів обміну повідомленнями (включаючи черги та видавець/передплатник), платформи потокової обробки подій працюють лише з шаблоном видавець/передплатник. Платформи потокової обробки подій призначені для великих обсягів повідомлень, тому розраховані на масштабування. Вони здатні впорядковувати потоки записів у категорії (теми) та зберігати їх протягом встановленого терміну. На відміну від брокерів, платформи не гарантують ні доставку повідомлень, ні відстеження статусу одержувачів.

Хоча можливості масштабування платформ потокової обробки подій ширші, ніж у брокерів повідомлень, ці платформи менш стійкі до збоїв (наприклад, немає функції

повторного надсилання повідомлень), а також обмежені можливості маршрутизації повідомлень і функції роботи з чергами.

Варіанти використання брокерів повідомлень

Брокери повідомлень можна реалізувати для широкого ряду завдань у різних галузях і корпоративних обчислювальних середовищах. Вони корисні в тих випадках, коли потрібен надійний зв'язок між програмами та гарантована доставка повідомлень. Найчастіше брокери повідомлень використовуються для наступних завдань:

- Фінансові транзакції та обробка платежів: вкрай важливо бути впевненим, що платіж буде відправлено один і лише один раз. Обробка таких транзакційних даних з використанням брокера повідомлень дає гарантію, що платіжна інформація не буде втрачена, ні випадково продубльована; забезпечує підтвердження отримання та дозволяє системам надійно взаємодіяти навіть при помилках у проміжних мережах;
- Обробка та виконання замовлень у сфері електронній комерції. Здатність брокерів повідомлень підвищувати стійкість до відмов, а також гарантія одноразового отримання повідомлень роблять їх чудовим варіантом для обробки онлайн-замовлень;
- Захист конфіденційних даних під час зберігання та передачі. Брокери повідомлень забезпечують обмін повідомленнями з функціями наскрізного шифрування;
- Високонавантажені системи реального часу з динамічним масштабуванням.

У міру того як організації модернізують додатки в ході освоєння хмарних технологій, брокери повідомлень набувають все більшого значення і відкриваються з

нового боку. Багато з найуспішніших компаній світу, використовують брокер повідомлень, призначений для підтримки сучасних середовищ гнучкої розробки, архітектур на основі мікросервісів та гібридної хмарної архітектури, а також різноманітних систем і вимог до зв’язку.

2.2.1. Apache Kafka

Apache Kafka — розподілений горизонтально масштабований стійкий до відмов брокер повідомлень. Програми (генератори) надсилають повідомлення (записи) на вузол Kafka (брокер), і зазначені повідомлення обробляються іншими програмами, так званими споживачами. Зазначені повідомлення зберігаються, а споживачі далі підписуються для отримання нових повідомлень. Kafka гарантує, що всі повідомлення будуть упорядковані саме в тій послідовності, в якій надійшли. Kafka не відстежує, які записи були считані споживачем і після цього видаляються, а просто зберігає їх протягом заданого часу. Споживачі самі опитують Kafka, чи не з'явилося в нього нових повідомлень і вказують, які записи їм потрібно прочитати.

2.2.2. RabbitMQ

RabbitMQ, як і Kafka, також розподілений горизонтально масштабований стійкий до відмов програмний брокер повідомлень.

Програми (publishers) посилають повідомлення на вузол RabbitMQ (exchange), причому RabbitMQ відсилає підтвердження, що повідомлення отримано. Одержувачі (consumers) постійно з'єднані з RabbitMQ TCP і чекають, коли RabbitMQ проінформує (push) їх про нове повідомлення. Одержувачі підтверджують отримання повідомлення або повідомляють про невдачу. Якщо доставка невдала,

RabbitMQ намагається відправити повідомлення до тих пір, поки воно не буде доставлене. Після успішної доставки повідомлення видаляється із черги.

2.2.3 Apache Pulsar

Pulsar - це розподілена платформа обміну повідомленнями в режимі pub-sub, з гнучкою моделлю обміну повідомленнями та інтуїтивно зрозумілим клієнтським API.

Pulsar поєднує деякі можливості Kafka та RabbitMQ, наприклад, організацію черги повідомлень та потокову передачу подій. Однак, насправді реалізувати обидва ці напрями на прийнятному рівні досить складно через різницю самих архітектурних моделей. Зокрема, черги повідомлень (Message Queue, MQ) використовуються для точкового зв'язку, забезпечуючи асинхронну взаємодію відправника та одержувача даних. У свою чергу, потокова обробка подій передбачає обробку даних в режимі онлайн з мінімальною часовою затримкою у спілкуванні між виробником (producer) і одержувачем (consumer). Pulsar складається з 3 основних компонентів:

- брокер – stateless-служба, до якої підключаються клієнти, які бажають обмінюватись повідомленнями;
- розподілений сервіс логів Apache BookKeeper, вузли якого (букмекери) зберігають фактичні повідомлення та позиції курсору (Cursor) для їх читання з топіка. BookKeeper використовує RocksDB як вбудовану базу даних, яка використовується для зберігання внутрішніх індексів, але не керується незалежно від BookKeeper.

сервіс синхронізації розподілених систем Apache ZooKeeper для зберігання метаданих брокерів та букмекерів.

Apache Pulsar має обмежену підтримку повідомлень, т.к. у ньому відсутні такі функції обміну даними, як транзакції, маршрутизація, фільтрація повідомлень та інші можливості, які зазвичай присутні у типових MQ–системах, наприклад, IBM MQ, RabbitMQ та ActiveMQ. Навіть MQ-адаптери Pulsar не надто допомагають вирішити цю проблему через низку функціональних обмежень. Таким чином, як і Kafka, Pulsar не повністю реалізує MQ–концепцію. Тому, якщо потрібно готове рішення для безпосереднього обміну повідомленнями, його слід шукати серед відповідних брокерів: RabbitMQ, IBM MQ, RabbitMQ, ActiveMQ, NATS та інші аналоги.

Pulsar також не повністю підтримує потокову обробку подій. Він забезпечує семантику суворо одноразової доставки повідомлень (exactly–once), про яку ми розповідали тут. Це значно скорочує кількість варіантів використання Apache Pulsar. Зокрема, із цим фреймворком не вдасться реалізувати обробку онлайн-платежів, т.к. за відсутності exactly-once гарантії будь-який збій може викликати дублювання чи втрату даних про фінансові проводки. Варто відзначити, що Pulsar надає функцію дедуплікації, яка гарантує, що повідомлення не буде збережено у брокері двічі. Однак це не заважає споживачеві прочитати повідомлення кілька разів, що не відповідає концепції exactly-once.

Крім того, Pulsar надає тільки деякі елементарні функції для потокової передачі, що підходить для простих зворотних викликів, але не стає повноцінною онлайн-обробкою. Тому не можна назвати Pulsar 100% альтернативою можливостям Kafka Streams або KSQL для створення stateful–додатків, які включають інформацію про стан та інші ідеї справжньої потокової обробки Big Data.

Нарешті, на відміну від транзакцій у Kafka, у Pulsar неможливо точно прив'язати повідомлення, зафіксовані до стану, записаного всередині потокового

процесора. Таким чином, Pulsar не має функцій потокового з'єднання, агрегування, віконної обробки, стійкості до відмови і обчислень на основі часу подій. А зворотним боком гнучкої багаторівневої архітектури за рахунок BookKeeper є зниження функціональних можливостей топіків порівняно з Kafka. Нагадаємо.

Некоректно напрямого порівнянювати Apache Pulsar з Kafka і RabbitMQ через різні концепції роботи з даними. Зокрема, RabbitMQ працює зі стійкою чергою, яка зберігає повідомлення на диск тоді і лише тоді, коли повідомлення ще не були використані. На відміну від Kafka і Pulsar, RabbitMQ не підтримує перемотування черг для повторного читання старих повідомлень. Крім того, RabbitMQ не передбачає розділів у топіці, використовуючи обмін для маршрутизації повідомлень у пов'язані черги за допомогою атрибутів заголовка, ключів маршрутизації або прив'язок обміну, з яких споживачі можуть обробляти повідомлення. Нарешті, у разі RabbitMQ накладні витрати на реплікацію даних серйозно знижують пропускну можливість всієї системи, адже цей фреймворк передбачає необмежене масштабування. Все це позначилося на процесі бенчмаркінгового тестування та його результатах, які вкотре показали першість Apache Kafka за пропускною здатністю та швидкістю обробки даних, а також співвідношенням ціна-якість (вартість записаного байта).

2.3. Порівняння Apache Pulsar, переваги та недоліки

Подібно Kafka, у Pulsar є розділи топіків, які теж є топіками. Як і в Apache Kafka, виробник може надсилати повідомлення циклічно, використовуючи алгоритм хешування або явно вибирати розділ. У Kafka кожне реєстр розділу повністю зберігається на одному брокері. Репліка реєстрів складається із серії файлів сегментів та індексів. Ця модель гарна тим, що вона проста та швидка: всі операції

читання та запису є послідовними. Однак для реалізації цього брокер має мати достатньо місця на жорсткому диску. Крім того, при масштабуванні кластера стає необхідним перебалансування, що потребує ретельного планування та виконання.

В Apache Pulsar кожен регістр з одного або декількох фрагментів може бути реплікований на кілька вузлів BookKeeper для підвищення продуктивності на читання. Кожен фрагмент тиражується в іншому наборі букмекерів, якщо їх достатньо. Дані топіки розподілені між декількома букмекерами. Топік розділений на регістри (Ledger) та фрагменти (Fragment) з чергуванням на обчислювані підмножини ансамблів фрагментів. При розширенні кластера потрібно просто додати букмекерів і вони почнуть отримувати записи при створенні нових фрагментів. Ребалансування розділів не потрібне. Кожен брокер Pulsar відстежує регістри та фрагменти топіка метаданими, які зберігаються в ZooKeeper. Тому, при збої ZooKeeper, Pulsar також відмовляє.

З точки зору моделі роботи з повідомленнями цікаві наступні подібності та відмінності Apache Pulsar від Kafka та RabbitMQ:

- Kafka використовує архітектуру на основі опитування, коли споживачі одержують повідомлення із сервера, а тривале опитування використовується для забезпечення миттєвого доступу до нових повідомлень;
- RabbitMQ використовує підхід на основі push, синонім традиційних систем обміну повідомленнями;
- Pulsar також використовує підхід на основі push, але з API, що імітує запити споживачів.

Архітектури на основі витягування (pull) зазвичай краще для роботи високонавантажених додатків з високою пропускною здатністю, дозволяючи

споживачам керувати своїм потоком даних, тобто. отримувати лише те, що потрібно. Push-архітектури вимагають, щоб у брокер були інтегровані управління потоком та буфферизацією (backpressure). З точки зору зберігання даних Pulsar трохи схожий на Kafka та RabbitMQ, але з низкою відмінностей:

- Kafka зберігає дані в розподіленому журналі фіксації, в кінець якого додаються нові записи. Читання є послідовними, починаючи зі зміщення дані копіюються з нуля з дискового буфера в мережевий буфер. Це підходить для потокової передачі подій;
- RabbitMQ та Pulsar використовують системи зберігання на основі індексів, зберігаючи дані в деревоподібній структурі, щоб забезпечити швидкий доступ для підтвердження окремих повідомлень. Швидкі окремі читання в деревоподібних структурах компенсуються збільшенням накладних витрат на запис даних, що проявляється у зменшенні пропускної спроможності запису або збільшенням затримки.

RabbitMQ зберігає повідомлення лише короткий період часу, а Pulsar та Kafka можуть робити це необмежено.

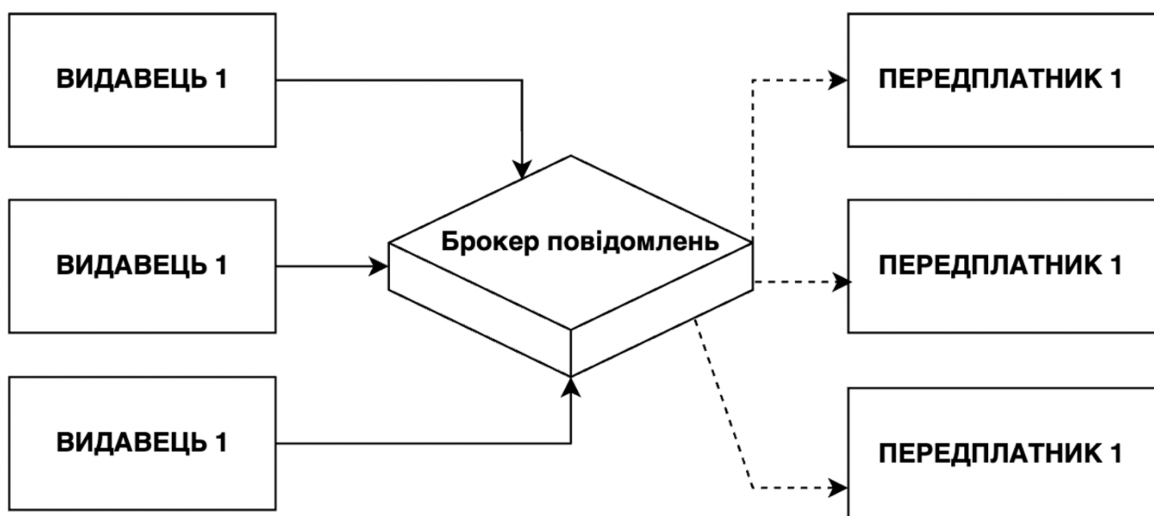


Рис. 2.1. Робота брокера повідомлень

На відміну від Kafka, де використовується модель монолітної архітектури, яка тісно пов’язує обслуговування та зберігання, Pulsar реалізує багаторівневу структуру, яка дозволяє керувати цими функціями на окремих рівнях. Брокер Pulsar виконує обчислення на одному рівні, а букмекер управляє stateful–сховищем на іншому рівні. Наявність BookKeeper в архітектурі Pulsar забезпечує більш гнучку масштабованість, меншу операційну навантаження, швидкість та стабільність високопродуктивної обробки Big Data.

Особливості роботи Apache Pulsar:

- Аналогічно Kafka, Pulsar зберігає повідомлення в топіках – структурі журналу, де кожне повідомлення має зсув, для відстеження якого використовується курсор (Cursor). Видавець (producer) відправляє свої повідомлення в заданий топік, і Pulsar гарантує, що після підтвердження повідомлення воно не буде втрачено;
- Споживач отримує повідомлення з топіка через підписку (Subscription) – логічну сутність, яка відстежує поточне зміщення споживача (курсор), а також надає деякі додаткові гарантії залежно від типу підписки;
- Підписка може бути ексклюзивною (Exclusive), коли лише один споживач може читати відповідний топік, загальною (Shared) підпискою, коли конкуруючі споживачі можуть одночасно читати топік. А при підписці на відмову (Fail-Over Subscription) для споживачів застосовується активний/резервний шаблон: при відмові активного споживача його замінює резервна копія, активний споживач одночасно може бути лише один. До одного топіка може бути прикріплено кілька підписок. Підписки не містять даних, тільки метадані та курсор;

- Pulsar забезпечує семантику черги та журналу (логу), дозволяючи споживачам розглядати топік як чергу, яка видаляє повідомлення після підтвердження споживачем, або як журнал, де споживачі можуть перемотувати курсор. При цьому модель сховища даних залишається тим самим логом;
- Якщо для топіка не задана політика зберігання даних через простір імен, то повідомлення видаляються після того, як повідомлення було підтверджено усім підписках, прикріплених до цього топику. Але, якщо задана політика зберігання даних у топіці, повідомлення видаляються після того, як вони виходять за кордон політики, наприклад, за розміром або часом;
- Повідомлення живуть певний час – вони видаляються, якщо вийшов час (Time to Live, TTL). Це означає, що повідомлення можуть бути видалені, перш ніж будь-який споживач отримає можливість їх прочитати. Термін дії застосовується лише до непідтверджених повідомлень а також підходить для архітектури і семантики черг;
- TTL застосовуються до кожної передплати окремо, що означає логічне видалення даних. Фактичне видалення відбудеться пізніше, залежно від інших підписок та політики зберігання даних;
- Споживачі підтверджують отримання повідомлень по одному або оптом (кумулятивно), що краще для пропускної спроможності, але додає обов’язкову обробку дублікатів після збоїв споживачів.

•

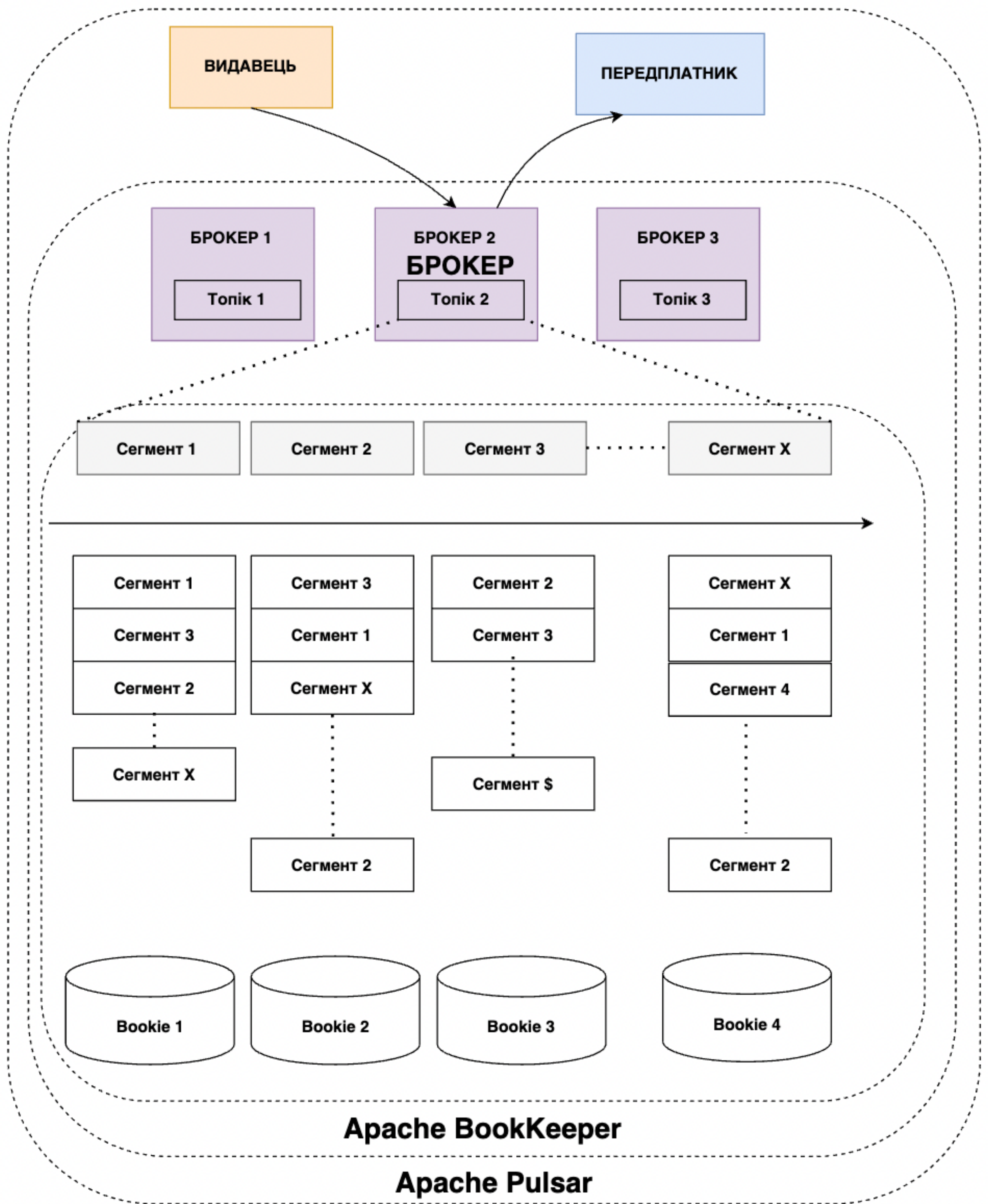


Рис. 2.2. Архітектура Pulsar

Таким чином, Pulsar поєднує переваги Apache Kafka і RabbitMQ, додаючи свої покращення та вирішуючи відповідні проблеми.

2.3. Вибір клієнтського фреймворку для розробки

Angular, React і Vue — одні з найпопулярніших бібліотеки та фреймворки JavaScript, які допомагають розробникам створювати складні, реактивні та сучасні користувацькі інтерфейси. За допомогою додаткових бібліотек, таких як React Native, Ionic (з Angular або з React) або NativeScript, можна створювати програми для мобільних пристроїв за допомогою Angular, React і Vue.

Не існує найкращого фреймворка чи бібліотеки. Усі три технології дуже популярні з різних причин. Усі мають свої сильні та слабкі сторони, їх використання напряму залежить від конкретно поставленої задачі та команди розробників.

2.3.1. Angular

Angular — це фреймворк, розроблений Google: Google користується Angular в своїх проектах, що означає про його постійну підтримку і вдосконалення.

React — це бібліотека, створена Facebook. Facebook написаний з використанням React. Vue — це "окремий" стартап, який не був створений всередині великої компанії. Раніше це було персональне шоу (Еван Ю, його засновник), але ті часи давно минули. Зараз у нього є спеціальна команда основних учасників, які працюють на Vue. Порівнюючи загальну філософії кожної технології, можна сказати, що Angular безумовно є основним фреймворком із трьох. Іноді його ще називають "платформою", а не фреймворком.

Angular по умовченню містить підтримку багатьох речей. Це допомагає повністю контролювати інтерфейс користувача, реагувати на введення користувача,

перевіряти данні, які введені у формах, маршрутизувати, керувати станом, надсилати запити Ajax Http, реалізувати усі можливості PWA.

Усі технології мають на меті полегшити розробнику створювати реактивні складні інтерфейси користувача. Angular надає повний набір інструментів для цього. Він не обмежений лише підтримкою DOM маніпуляцій а також додає вищезгадані функції, щоб реалізувати усі можливі аспекти роботи додатка.

Крім того, Angular містить офіційний CLI, який полегшує створення та редагування проектами на Angular а також керувати ними, підтримувати їх у актуальному стані, додавати залежності та навіть займатися розгортанням.

Основне завдання Angular полягає в створенні повторно використовуваних компонентів користувацького інтерфейсу. Їх можна повторно поєднувати з іншими компонентами для створення повноцінного інтерфейсу користувача з цих компонентів, які були створені на Angular.

2.3.2. React

React на відміну від Angular надає розробнику лише бібліотеку для редагування елементів в DOM і ефективного керування ними. React напряму пов’язаний зі створенням користувацьких інтерфейсів за допомогою компонентів. React надає усі інструменти, які потрібні для маніпуляціями елементами інтерфейсу, дає можливість контролювати, що потрібно бути відображено, в який спосіб і за яких обставин. У React відсутня вбудована підтримку валідацій форм. Сама бібліотека не має вбудований маршрутизатор (для зміни та рендеру різних компонентів на основі змін URL–адреси) і не має власний Http–клієнта. React підтримує управління внутрішнім станом компонентів та передачу стану від одного

компонента іншим. Для додаткових функцій (таких як маршрутизація та валідація форм)потрібно встановлювати додаткові бібліотеки.

2.3.3. Vue.js

Vue — це бібліотека, який поєднує особливості React і Angular. Бібліотека не така «громіздка», як Angular, але вона містить більше функцій ніж React. Vue постачається з вбудованим керуванням станом, і також з вбудованим маршрутизатором. Однак по замовченню Vue не підтримує валідацію форм та функціональні можливості клієнта Http.

Як і Angular і React, ядро Vue оперує створенням інтерфейсів користувача шляхом поєднання компонентів, які можна повторно використовувати. Але окрім цього, бібліотека надає більше можливостей у порівнянні з React, і трохи менше, у порівнянні з Angular. Але не завжди більше можливостей це кращий вибір. Все це залежить від проекту, і від особистих уподобань.

Деякі функції, такі як керування станом і маршрутизація, будуть потрібні майже у кожному проекті, незалежно від того, наскільки великий чи маленький проект. Angular і Vue мають повну вбудовану підтримку для наступних операцій, React має лише вбудовану підтримку управління станом (не ідеально для високочастотних змін) і не має вбудованої підтримки маршрутизації.

Можна стверджувати, що простота React і його сильна зосередженість на компонентах і візуалізації інтерфейсу є його великою перевагою. Там, де Angular потрібно з'єднати багато речей і забезпечити їх безперебійну роботу, з React цього робити не потрібно.

Для інших “інструментів”, які можуть знадобляться, існує душе активна спільноту React. Спільнотою були розроблені бібліотека (додаткові пакети, які ви можете додати до свого проекту), таких як маршрутизатор React, Redux або Formik.

Можна стверджувати, що React зосереджений на тому, щоб надати найкращу можливу бібліотеку рендерингу інтерфейсу, а його спільнота зосереджується на окремих бібліотеках, які доповнюють цю бібліотеку.

Команда Angular включає величезну команду досвідчених розробників. Це означає про відсутність проблем з версіями або несумісністю бібліотек. Різні модулі завжди працюватимуть безперебійно, оскільки їх розроблює єдина команда.

2.3.4. Порівняння та вибір клієнтського фреймворку

Angular був розроблений Google, React — Facebook, Vue — проект із відкритим кодом, який створила спільнотою. Всі інструмент представляють собою компонентні бібліотеки та фреймворки, де Angular — повноцінний фреймворк, тоді як Vue та React звичайні бібліотеки. Angular змушує розробляти на TypeScript, змішуючи логіку HTML + TypeScript, React використовує JavaScript з синтаксисом, який має назву "JSX" (комбінація "HTML" і JavaScript), аналогічно Vue використовує нативний JavaScript і розроблюється на HTML + JavaScript. Vue та React найлегші для вивчання, Angular найважчий у навчанні через потребу у TypeScript та обмеженням фреймворку. Усі три технології пропонують чудову продуктивність роботи додатку, але React найшвидший із них. React найпопулярніша бібліотека для побудови користувацьких інтерфейсів з найбільшою спільнотою розробників, Angular та Vue менш популярні.

2.4. Вибір реалізації бази данних для системи.

2.4.1 MongoDB

MongoDB — це документно-орієнтована база даних NoSQL, яка використовується для зберігання даних великих масштабів. MongoDB з'явилася приблизно в середині 2000-х років. Mongo відноситься до категорії баз даних NoSQL.

MongoDB безкоштовна у використанні, має відкритий вихідний кодом; однак принципи проектування БД у ній відрізняються від традиційних реляційних систем. MongoDB часто описується як нереляційна (або NoSQL) система, MongoDB використовує значно інший підхід до зберігання даних, представляючи інформацію у вигляді серії документів, подібних до JSON (фактично зберігаються як двійкові JSON або BSON), на відміну від таблиці та рядків у реляційних системах.

Документи MongoDB складаються з серії пар ключ/значення різних типів, включаючи масиви та вкладені документи; однак основна відмінність полягає в тому, що структура пар ключ/значення в даній колекції може відрізнятися від документа до документа у одній таблиці. Цей підхід додає гнучкості у розробці рівня данних у додатку.

NoSQL СУБД використовує динамічні схеми, що дозволяю створювати записи без попереднього визначення структури, наприклад полів або типів та їх значень. MongoDB дозволяє змінювати структуру записів, які називаються документами, додаючи нові поля або видаляючи наявні.

2.4.2 MySQL

MySQL — це популярна, безкоштовна і відкрита система керування реляційними базами даних (RDBMS), розроблена Oracle. Як і інші реляційні

системи, MySQL зберігає данні у виді таблиць і рядків, забезпечуючи цілісність користуючись мовою структурованих запитів (SQL) для доступу до даних.

Схеми баз даних і моделі даних повинні бути заздалегідь висзначеними, і дані повинні відповідати цій схемі, щоб зберігатися в ній. Цей жорсткий підхід до зберігання даних забезпечує певний ступінь безпеки, але жертвує гнучкістю. Якщо потрібно зберегти новий тип або формат даних, повинна відбутися міграція схеми, яка бути складною та дорогою зі збільшенням розміру бази даних.

У ядрі СУБД (система управління базою даних) лежить математична модель реляційної алгебри. Це робить адміністрування баз даних легшим і гнучкішим.

Працюючи з MySQL, потрібно заздалегідь визначити схему бази даних на основі вимог і налаштувати правила та обмеження, які допоможуть керувати відносинами між полями в таблицях.

Ключові різниці

MongoDB представляє дані як документи JSON, тоді як MySQL представляє дані в таблицях і рядках. У MongoDB не потрібно визначати схему. Користуючись MySQL потрібно визначити типи та обмеження таблиць та стовпців. MongoDB не підтримує напряму операцію JOIN (через оператор $lookup Aggregation Framework), на відміну від MySQL. MongoDB використовує JavaScript як мову запитів, а MySQL використовує мову структурованих запитів SQL. MongoDB — ідеальний вибір, якщо є неструктуровані або структуровані дані з потенціалом швидкого зростання, тоді як MySQL — чудовий вибір, якщо є структуровані дані та потрібна традиційна реляційна база даних.

Якщо більшість ваших сервісів розгорнені через хмарні технології, MongoDB найкраще підходить під таку архітектуру, але якщо безпека та консистентність даних є пріоритетом, то MYSQL — найкращий варіант.

Основні причини використання MongoDB:

- MongoDB дуже гнучка і адаптивна до реальних ситуацій і вимог бізнесу;
- Можливість робити запити для повернення певних полів у документах а не всієї записи;
- MongoDB підтримує запити на основі полів, діапазонів, регулярні вирази для пошуку даних;
- MongoDB — це дуже проста система СУБД, яку можна легко масштабувати;
- MongoDB допомагає використовувати внутрішню пам'ять для зберігання робочих тимчасових наборів даних, що робить її роботу набагато швидшою;
- MongoDB пропонує первинні та вторинні індекси для будь–якого поля;
- MongoDB підтримує реплікацію бази даних;
- Можливість використовувати MongoDB як систему зберігання файлів, названою GridFS;
- MongoDB пропонує різні методи для виконання операцій агрегації над даними, як конвеєр агрегації, який містить команди map–reduce, count і т.п;
- MongoDB дозволяє зберігати файли будь-якого типу та будь-якого розміру;
- MongoDB в основному використовує функції JavaScript замість стандартних процедури;
- MongoDB підтримує спеціальний тип індексів, TTL (Time–To–Live) для зберігання даних, які можуть бути видаленими через певний час;

- Схема динамічної бази даних, що використовується в MongoDB, називається JSON;
- Індекси можуть бути створені для підвищення ефективності пошуку в MongoDB. Будь-яке поле в документі MongoDB можна проіндексувати;
- Реплікація – MongoDB може забезпечити високу доступність за допомогою наборів реплік;
- MongoDB може працювати на кількох серверах, балансуючи навантаження та/або дублюючи дані, щоб підтримувати роботу системи в разі збою обладнання.

Причини використання MYSQL

MySQL використовують з наступних причин:

- Підтримка таких функції, як реплікація Master–Slave, масштабування Формування звітів про навантаження, розподіл даних тощо.
- Дуже низькі накладні витрати на читання за допомогою механізму зберігання MyISAM, який використовується у системах з перевагою на читання.
- Підтримка механізму зберігання у пам’яті часто використовуваних таблиць
- Запити кешуються для операцій які часто повторюються.

Особливості MongoDB

Кожна база даних містить колекції, які, у свою чергу, містять документи. Кожен документ може відрізнятися кількістю полів, розміром і змістом. Структура документа MongoDB залежить від того, як розробники створюють свої класи та об’єкти на своїх відповідних мовах програмування. Рядки не повинні мати визначену схему. Натомість поля можна створювати на льоту. MongoDB дозволяє

легше представляти ієрархічні відносини, зберігати масиви та інші складніші структури.

Особливості MYSQL

Серед особливостей СУБД MySQL можна виділити наступні:

- MySQL — це СУБД, яку підтримує велика спільнота розробників
- Сумісний з різними платформами, які використовують усі основні мови програмування та проміжне програмне забезпечення
- Пропонує підтримку багатоверсійного контролю паралельності
- Відповідає стандарту ANSI SQL
- Дозволяє SSL реплікацію базуючись на логах і тригерах.
- Об'єктно-орієнтований і сумісний з ANSI-SQL2008
- Багатошарова конструкція з незалежними модулями
- Підтримує багатопоточність, використовуючи потоків ядра
- Пропонує вбудовані інструменти для аналізу часу виконання запитів.
- Може обробляти будь-яку кількість даних, до 50 мільйонів рядків і більше
- MySQL працює на багатьох версіях UNIX і Linux.

2.4.3. Ключові відмінності

Проаналізувавши особливості відповідних СУБД можно виділити основні властивості:

- MongoDB представляє дані як документи JSON. MySQL представляє дані в таблицях і рядках.
- У MongoDB не потрібно визначати схему. Замість цього можливо вводите документи, навіть не обов'язково мати однакові поля у кожних сутністях.

- MySQL вимагає визначення таблиці та стовпців, перш ніж дає можливість щось зберігати, і кожен рядок у таблиці повинен мати однакові стовпці.
- MongoDB має попередньо визначену структуру, яку можна визначити та дотримуватися, але також, якщо потрібні різні документи в колекції, вона може мати різні структури.
- MySQL використовує мову структурованих запитів (SQL) для доступу до бази даних. Неможливість змінювати схему.
- Підтримувані мови – C++, C. Підтримувані мови – C++, C та JavaScript Постійну розробку здійснює MongoDB, Inc. Постійну розробку здійснює корпорація Oracle.
- MongoDB підтримує вбудовану реплікацію, шардінг і автоматичну виборку. MySQL підтримує реплікацію “master–slave” і “master” реплікацію.
- Якщо більшість сервісів базується у хмарній архітектурі, MongoDB найкраще підходить для цього. Якщо безпека даних є пріоритетом, то обирають MYSQL. MongoDB не накладає обмежень на розробку схеми. MySQL вимагає визначення схем таблиць та типів стовпців, перш ніж можна щось зберігати.
- Кожен рядок таблиці повинен мати однакові стовпці.
- MongoDB використовує JavaScript як мову запитів. MySQL використовує мову структурованих запитів (SQL).
- MongoDB не підтримує JOIN. MySQL підтримує операції JOIN.
- Швидко обробляє великі неструктуровані дані. MySQL є досить повільним у порівнянні з MongoDB при роботі з великими базами даних.
- MongoDB – ідеальний вибір, якщо є неструктуровані та/або структуровані дані з потенціалом швидкого зростання. MySQL обирають, якщо у є структуровані дані та потрібна традиційна реляційна база даних.

Недоліки використання MongoDB

MongoDB не повністю підтримує принципи ACID (атомність, консистенція, ізоляція та довговічність) у порівнянні з багатьма іншими системами СУБД. Транзакції у MongoDB є складними для написання і розуміння. У MongoDB не передбачено використання збережених процедур або функцій, відповідна логіка реалізовується через механізм потоків змін у таблицях.

Недоліки використання MySQL

Транзакції, пов'язані з системним каталогом, не відповідають вимогам ACID. Іноді збій сервера може пошкодити системний каталог і роботу всієї СУБД. Збережені процедури не кешуються. Таблиці MYSQL, які використовуються для процедури або тригера, найчастіше попередньо заблоковані.

Основні відмінності між MongoDB та MySQL?

Основні відмінності між цими двома СУБД є суттєвими. Вибір того, який із них використовувати, насправді є архітектурним питанням.

MySQL — це зріла система реляційних баз даних, що пропонує знайоме середовище баз даних для досвідчених IT–фахівців.

MongoDB — це добре налагоджена нереляційна система баз даних, яка пропонує підвищену гнучкість і горизонтальну масштабованість, але за рахунок деяких функцій безпеки реляційних баз даних, таких як цілісність даних.

Зручність у використанні MongoDB і MySQL

MongoDB є привабливим варіантом для розробників. Його філософія зберігання даних проста і відразу зрозуміла кожному, хто має досвід програмування особливо на мові Javascript.

MongoDB зберігає дані в колекціях без структурованої схеми. Цей гнучкий підхід до зберігання даних робить його особливо придатним для розробників, які, можливо, не є експертами з адміністрації баз даних, але хочуть використовувати базу даних для розробки своїх програм.

Порівняно з MySQL ця гнучкість є значною перевагою: щоб отримати максимальну віддачу від реляційної бази даних, спочатку необхідно зрозуміти принципи нормалізації, посилальної цілісності та дизайну реляційної бази даних.

Завдяки можливості зберігати документи з різними схемами, включаючи неструктуровані набори даних, MongoDB надає розробнику гнучкий інтерфейс для команд, які створюють програми, яким не потрібні всі функції безпеки, які пропонуються реляційними системами. Прикладом такої програми є веб-додаток, який не залежить від структурованих схем; він може легко обслуговувати неструктуровані, напівструктуровані або структуровані дані з однієї колекції MongoDB.

MySQL обирають розробники, які мають великий досвід використання традиційних SQL БД, розробки додатків для реляційних баз даних або тих, хто переписує або оновлює існуючі програми, які вже працюють з реляційною системою. Реляційні бази даних також можуть бути кращим вибором для програм, які вимагають дуже складних, але жорстких структур даних і схем баз даних у великій кількості таблиць.

Звичайним прикладом такої системи може бути банківська програма, яка вимагає дуже сильної довідкової цілісності та гарантій транзакцій, які необхідно забезпечити для підтримки точної цілісності даних на кожний момент часу.

MongoDB також частково підтримує властивості ACID транзакцій (атомічність, узгодженість, ізольованість та довговічність). Це забезпечує більшу гнучкість у створенні моделі транзакційних даних, яка може горизонтально масштабуватися в розподіленому середовищі і не впливає на продуктивність багатодокументних транзакцій.

Масштабування MongoDB проти MySQL

Ключовою перевагою дизайну MongoDB є те, що базу даних надзвичайно легко масштабувати. Налаштування розподіленого кластера дозволяє частину бази даних, яка називається шардом, також налаштовувати як набір реплік. У розподіленому кластері дані розподіляються між багатьма серверами. Цей надзвичайно гнучкий підхід дозволяє MongoDB горизонтально масштабувати систему як на читання, так і на запис, щоб задовольнити потреби будь-яких програм.

Набір реплік — це реплікація групи серверів MongoDB, які зберігають ті самі дані, що забезпечує високу доступність та аварійне відновлення данних у разі помилок.

З системою баз даних MySQL можливості масштабованості значно обмежені. Зазвичай у є варіанти: вертикально масштабувати БД або додавання нових реплік на читання. Вертикальне масштабування передбачає додавання додаткових ресурсів до існуючого сервера бази даних, яке обмежано максимально допустимими можливостями системи.

Реплікація БД на читання передбачає додавання копій бази даних лише для читання на інші сервери. Однак це зазвичай обмежується п’ятьма репліками, які можна використовувати лише для читання даних. Це може спричинити проблеми з додатками, які інтенсивно записують, або регулярно записують і читають для бази даних, оскільки додаткові репліки зазвичай відстають від головної. До MySQL була додана підтримка багатоголовної реплікації, але її реалізація більш обмежена, ніж функціональність, доступна в MongoDB.

Продуктивність MongoDB у порівнянні з MySQL

Оцінити продуктивність двох абсолютно різних систем баз даних дуже складно, оскільки обидві системи управління підходять до завдання зберігання та пошуку даних абсолютно по-різному.

MySQL оптимізовано для високопродуктивних об’єднань декількох таблицях, які були належним чином проіндексовані. У MongoDB об’єднання підтримуються за допомогою операції $lookup, але вони менш потрібні через те, як зазвичай використовуються документи MongoDB; вони дотримуються ієрархічної моделі даних і зберігають більшу частину даних в одному документі, таким чином усуваючи необхідність об’єднання кількох документів.

MongoDB також оптимізовано для продуктивності запису та має спеціальний API insertMany() для швидкої вставки даних, віддаючи пріоритет швидкості над безпекою транзакцій. У MySQL потрібно вставляти рядок за рядком.

Спостерігаючи за деякими високорівневими запитами двох систем, можна побачити, що MySQL швидше читає велику кількість записів, тоді як MongoDB значно швидше вставляє або оновлює велику кількість записів.

Гнучкість MongoDB проти MySQL

Безсхемний дизайн документів MongoDB дозволяє надзвичайно легко створювати та покращувати програми з часом, без необхідності запускати складні та дорогі процеси міграції схем, як це було б із реляційною базою даних.

У MongoDB є більш динамічні варіанти оновлення схем у колекціях, наприклад створення нових полів на основі конвеєра агрегації або оновлення вкладених полів масиву. Ця перевага особливо важлива, оскільки бази даних постійно збільшуються. На відміну від цього, більші бази даних MySQL повільніше переносять оновлення схем та збережених процедур, які можуть залежати від оновлених схем. Гнучкий дизайн MongoDB робить це набагато меншим.

Варто зазначити, що обидві бази даних встановити на Linux та Windows, і обидва мають широку підтримку мов програмування для популярних мов, таких як Java, node.js та Python.

MongoDB пропонує для використання MongoDB Atlas, адміністроване хмарне рішення, для динамичного розгортання та масштабування БД.

MongoDB проти MySQL у плані безпеки даних

MongoDB використовує популярну модель контролю доступу на основі ролей з гнучким набором дозволів. Користувачам призначається роль, і ця роль надає їм певні дозволи на набори даних і операції з базою даних. Весь зв'язок шифрується за допомогою TLS, і можна записувати зашифровані документи в колекції даних MongoDB, використовуючи Primary key, який ніколи не буде відомий навіть для MongoDB, досягаючи повне шифрування даних.

MySQL підтримує ті ж функції шифрування, що й MongoDB. Модель аутентифікації MySQL також схожа з MongoDB. Користувачам можуть бути надані

ролі, а також привілеї, що надає їм дозволи на певні операції з базою даних і на певні набори даних.

Висновки до розділу 2

У розділі проведено аналіз технологій для реалізації проектованої системи. Найбільш доцільним є вибір технологій JavaScript, React.js, нереляційна база данних MongoDB та архітектурний шаблон брокера повідомлень Pulsar є найкращим стеком технологій для інформаційної системи валідації телефонних номерів.

Node.js є середовищем виконання JavaScript з відкритим кодом. Fastify.js є фреймворком, який базується на технології Node.js, і найчастіше використовується для розробки веб-додатків через свою простоту, великої кількості плагінів та високої продуктивності.

Node.js чудово себе показує для розробки мікросервісної архітектури з можливістю швидко масштабуватися.

Бібліотеку для побудови користувацьких інтерфейсів React.js було обрано через потребу системи у багаторазовому використанні компонентів, високій продуктивності.

Нереляційна СУБД MongoDB зберігає данні у JSON форматах, тим самим зменшуючи кількість інформації, яка зберігається у самому додатку. Оновлення даних у реальному часі базується на великій кількості записів у БД, через що і було обрано MongoDB. Предметну область легко структурувати, виділити сутності та зв’язки між ними.

Так як розробляється високопродуктивний додаток а також присутній досвід роботи з мовою JavaScript та програмуванням на основі подій, було обрано архітектурний шаблон комунікації сервісів через брокер повідомлень.

Використання цього стеку технологій дозволяє:

- покращити стійкість системи;
- збільшити швидкість оновлення данних у реальному часі;

- забезпечити сервіс легким маштабуванням;
- полегшити розробку, використанням однієї мови для клієнтсьої та серверної частини.

Під час роботи при великому навантаженні (спортивних подій) данний стек технологій може забезпечити стабільну роботу системи, зменшити кількість помилок до мінімума, які може отримати клієнт та збільшити реактивність інтерфейсу, що стане перевагами данної системи, порвінюючи з конкурентами.

РОЗДІЛ 3
РОЗРОБКА МІКРОСЕРВІСНОГО ДОДАТКУ ОБРОБКИ ДАННИХ У РЕАЛЬНОМУ ЧАСІ

3.1. Загальна концепція роботи додатку

Для того щоб контролювати налаштування виплат стосовно ставок користувач додаток використовує шаблони (Templates), які прив'язані до відповідної ліг групи (League Group), яка включає у себе багато майстер ліг (Master League). Майстер ліга включає у себе багато спортивних подій (Event). Процес контролювання ставок на потрібні події, які можуть статися під час або перед матчем і налаштовується за допомогою потрібного шаблону. Система складається з 51 сервісів, які дублюють інфраструктуру для кожного оператора, кожен з яких успадковує налаштування від глобального оператора з можливістю перевизначати налаштування з глобальних шаблонів а також можно конфігурувати потрібні спортивні події власноручно.

Для економії пам'яті у сервісі присутня функція компресії шаблонів. Сервіс може відображати виплати у різних цифрових форматах, присутня можливість переходити на глобальний рівень опреатора, шукати події за відповідною назвою та спортом. Додаток транслює оновлення інформації стосовно спортивної події у реальному часі на всі оператори з мінімальними затримками. Ліг групам та майстер лігам у них можна змінювати порядок та пріорітет. Присутня можливість порівнювати шаблони з відображенням налаштувань, які не зходяться. Додаток використовується для налаштувань поточних спортивних подій а також спортивних подій, які будуть проводитися у майбутньому. Пошук шаблонів та подій реалізовується за ім'ям та за допомогою додаткових фільтрів. Сервіс допомагає користувачам (операторам) заморожувати можливість ставок на події, аналізувати ризики та попереджати користувача про підозрілі ставки.

Контролювати налаштування виплат та ставок на спортивні події можна скориставшився вкладкою “Configaration”. Вибрати потрібник шаблон (Templates) налаштувань зі списку, який прив’язаний до певної спортивної події.

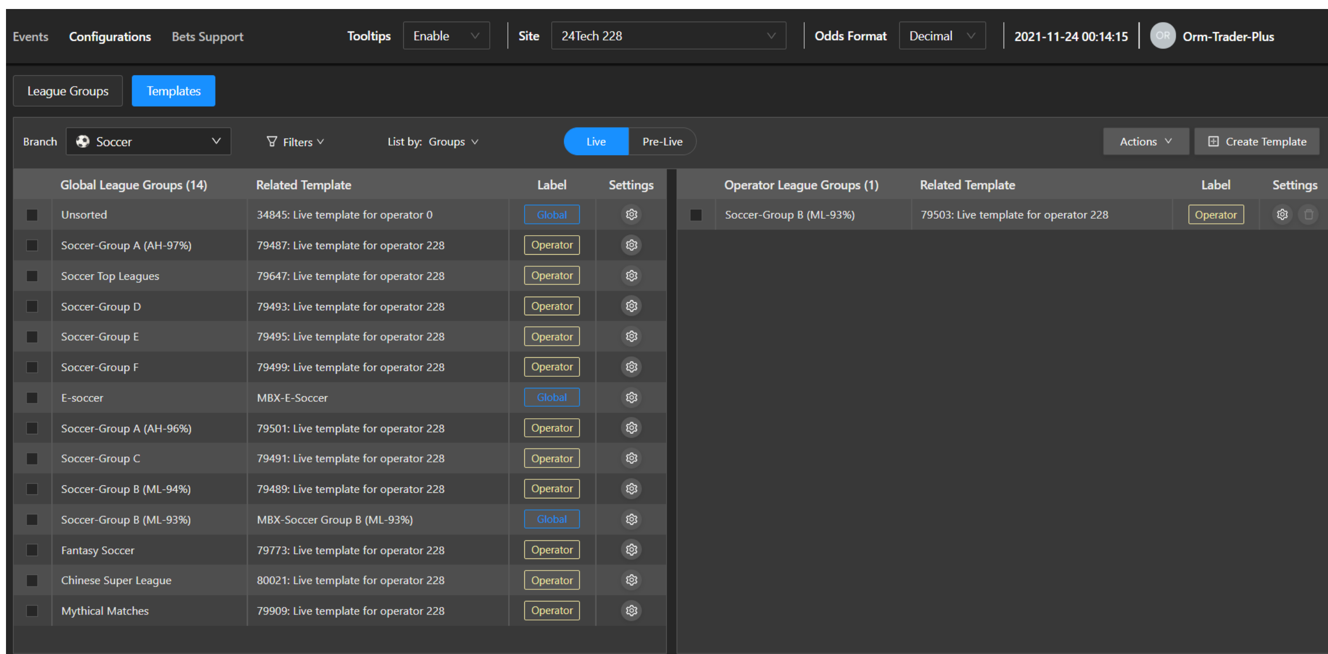


Рис. 3.1. Сторінка Configuration для обирання відповідних шаблонів

Події пов’язані з шаблонами, через сутність LeagueGroup. Кожна ліг група містить прив’язку до відповідного (Template) шаблону, певну кількість MasterLeagues, які у свою черну складаються з різних спортивних подій.

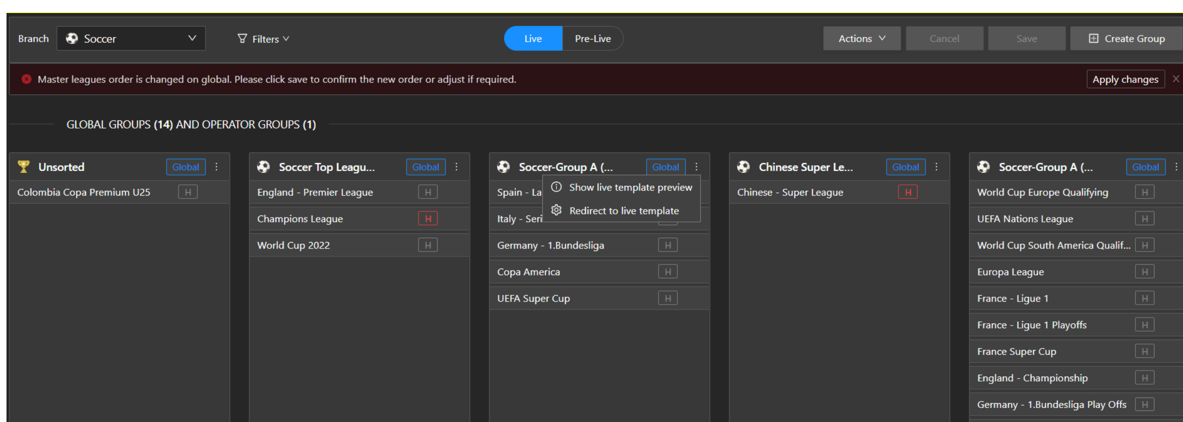


Рис. 3.2. Сортування ліг груп

Користувач може змінювати налаштування шаблона. Така зміна буде використовувати відповідні значення для всіх подій, які знаходяться у Майстер Лігах відповідної Ліг Групи, яка і зв’язана з цим шаблоном.

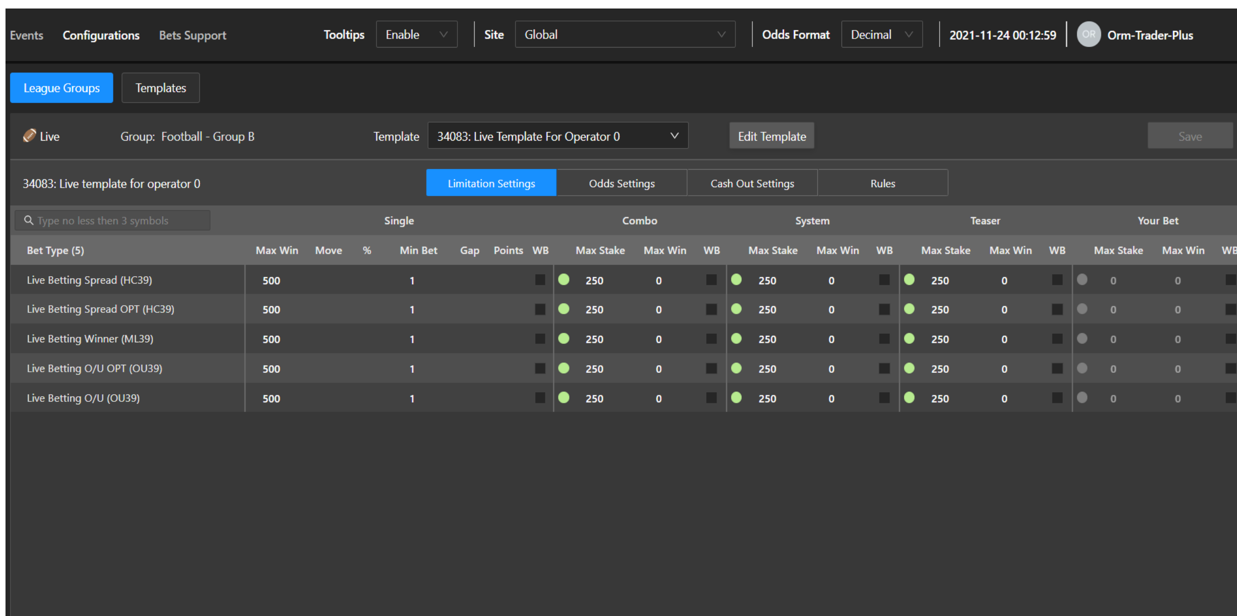


Рис. 3.3. Відкритий темплейт

Кожнен шаблон містить відповідні налаштування для подій. (Максимальні/мінімальні виплати, чи можна повернути ставку, якщо так то який відцоток грошей повернеться користувачу).

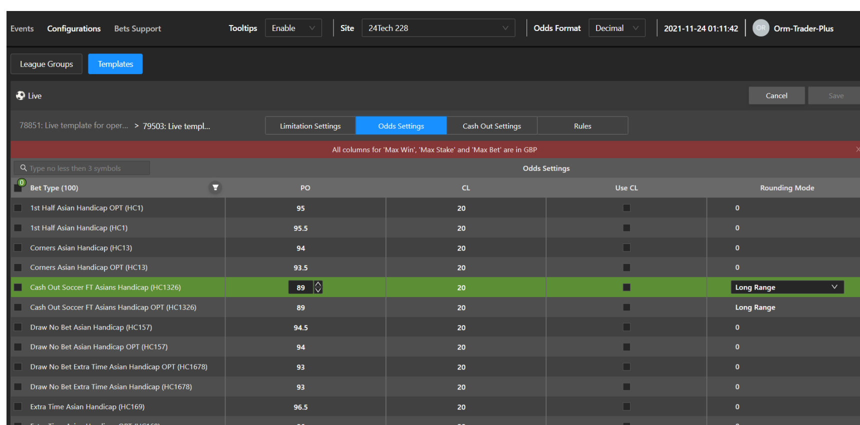


Рис. 3.3. Налаштування вірогідностей шаблону

В залежності від контексту (Глобальний або операторський) можно використовувати різні шаблони, ліг групи. Операторські сутності використовують наслідування від глобальних перевизначаючи лише відповідні параметри, з можливістю повернутися на значення, яке вказано у батьківській сутності.

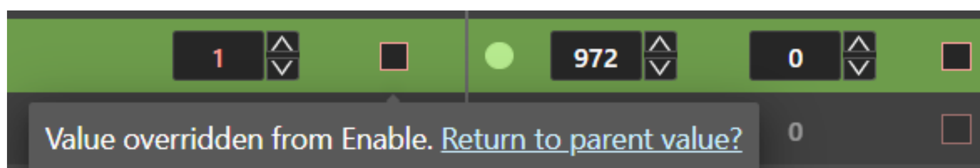


Рис. 3.4. Приклад перевизначених параметрів на шаблоні

Для оптимізації пам’яті БД, існує функція компрессії шаблонів. Обравши 2 шаблона можна порівняти їх, зробивши одного з них батьківським, видаливши всі записи, які однакові з батьківським у дочірнього з БД.

Templates list > Compare templates | Limitation Settings | Odds Settings | Cash Out Settings | Rules | Show only difference | Cancel | Compress

79495: Live template for operator 228 Operator | Make As Parent

Bet Type (78)	Max Win	Move	%	Min Bet	WB	M
1st Half Asian Handicap OPT (HC1)	432			1		
1st Half Asian Handicap (HC1)	432			1		
Corners Asian Handicap (HC13)	200			1		
Corners Asian Handicap OPT (HC13)	200			1		
Draw No Bet Asian Handicap (HC157)	400			1		
Draw No Bet Asian Handicap OPT (HC157)	300			1		
Draw No Bet Extra Time Asian Handicap OPT ...	432			1		
Draw No Bet Extra Time Asian Handicap (HC1...	432			1		
Extra Time Asian Handicap (HC169)	432			1		
Extra Time Asian Handicap OPT (HC169)	432			1		
Corners Extra Time Asian Handicap (HC1754)	432			1		
Corners Extra Time Asian Handicap OPT (HC1...	432			1		

79499: Live template for operator 228 Operator | Make As Parent

Bet Type (77)	Max Win	Move	%	Min Bet	WB	M
1st Half Asian Handicap (HC1)	300			1		
1st Half Asian Handicap OPT (HC1)	300			1		
Corners Asian Handicap (HC13)	200			1		
Corners Asian Handicap OPT (HC13)	200			1		
Draw No Bet Asian Handicap OPT (HC157)	200			1		
Draw No Bet Asian Handicap (HC157)	200			1		
Draw No Bet Extra Time Asian Handicap OPT ...	400			1		
Draw No Bet Extra Time Asian Handicap (HC1...	400			1		
Extra Time Asian Handicap OPT (HC169)	400			1		
Extra Time Asian Handicap (HC169)	400			1		
Corners Extra Time Asian Handicap (HC1754)	400			1		
Corners Extra Time Asian Handicap OPT (HC1...	400			1		

Рис. 3.5. Сторінка компрессії шаблонів

На сторінці /events можна за допомогою пошуку по подіям (які відбуваються наразі (live) та ті, які будуть відбуватися через деякий час (pre– live)) відкрити налаштування відповідної події, та налаштувати значення для відповідних ситуацій для ставок (кількість голів, хто переможе і т.п). Значення по замовченню беруться з

шаблона, який прив’язаний до події, але ми можемо перевизначити деякі параметри локально для цієї події.

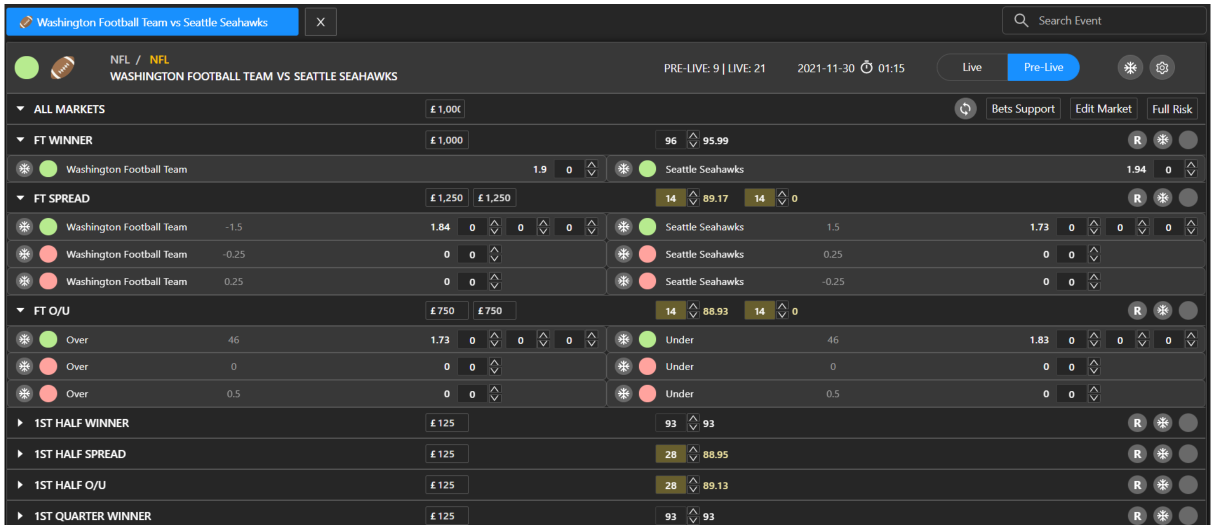


Рис. 3.6. Відкрита подія з налаштуваннями

Жовтуватим кольором показані параметри, які були змінені конкретно для цієї події. Інші значення беруться з налаштувань шаблона.

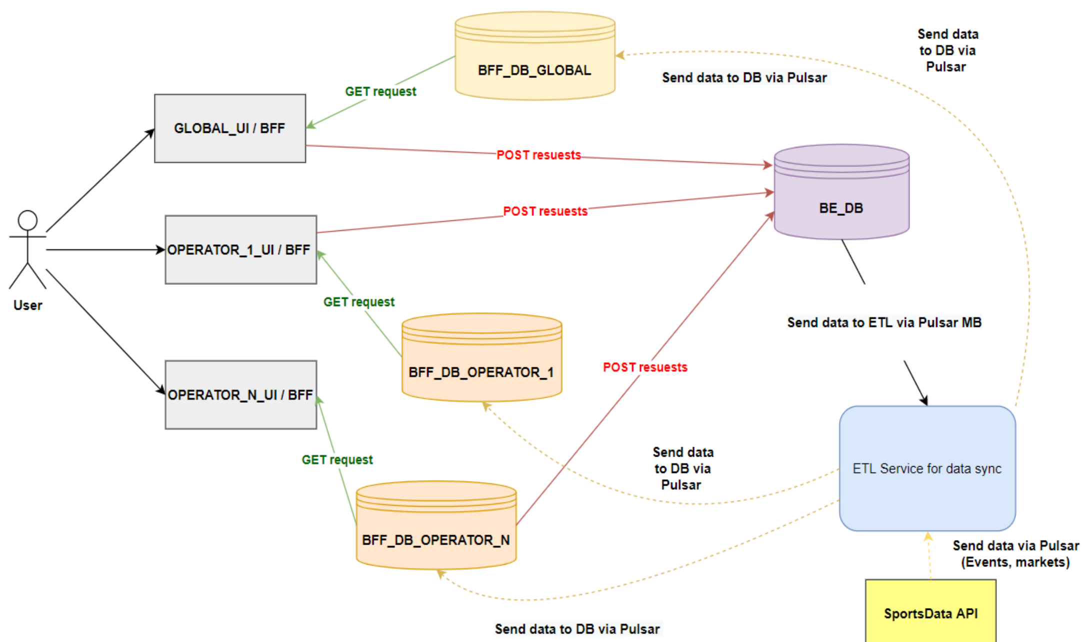


Рис. 3.7. Схема роботи мікросервісного додатку

Система складається з єдиного BE-сервісу та приблизно 50 BFF– сервісів

3.2. Віртуалізація за допомогою Docker

Перед розробкою мікросервісної системи по-перше потрібно підняти додаток у Docker – контейнері, щоб досягнути масштабованості за допомогою сервісу оркестрації та масштабування контейнерів Kubernetes.

Для того, щоб запакувати додаток у контейнер треба описати його Dockerfile. Dockerfile – список команд, які треба виконату для створення ємуляції відповідної ОС у unix-контейнері, завантаження потрібних бібліотек для роботи додатка, відкриття портів на хост-машину і т.п.

```
FROM main-harbor.btigroup.io/docker-snapshots/node:12-stretch

RUN wget --user-agent=Mozilla -O apache-pulsar-client.deb "https://archive.apache.org/dist/pulsar/pulsar-2.7.2/DEB/apache-pulsar-client.deb"
RUN wget --user-agent=Mozilla -O apache-pulsar-client-dev.deb "https://archive.apache.org/dist/pulsar/pulsar-2.7.2/DEB/apache-pulsar-client-dev.deb"

RUN apt install -y ./apache-pulsar-client.deb
RUN apt install -y ./apache-pulsar-client-dev.deb


ARG APPUSER=appuser

ENV PROJECT_DIR /opt/operator_risk_management_bff
ENV APPUSER appuser

RUN groupadd -g 12000 ${APPUSER}
RUN useradd -u 12000 -g ${APPUSER} -d ${PROJECT_DIR} -m ${APPUSER}

USER ${APPUSER}

RUN chmod -R 764 ${PROJECT_DIR}

WORKDIR ${PROJECT_DIR}

COPY --chown=${APPUSER} package.json ./

RUN npm config set @bti:registry http://proget-dev.btigroup.io/npm/npm-packages/
RUN npm install

COPY --chown=${APPUSER} . .

RUN npm run build

RUN rm -r /opt/operator_risk_management_bff/src

EXPOSE 8001

CMD [ "node", "./dist/runner.js" ]
```

Рис. 3.8. Конфігурація сервісів інфраструктури

Інфраструктура піднімається за допомогою відповідного docker-compose.yaml файлу за допомгою команди:

```
Алекс@DESKTOP-RIMNE1H MINGW64 ~/Desktop/hlr_js
$ docker-compose up -d
```

Docker-compose використовується для того, щоб прописати мережевий драйвер, порти, файли та папки, які треба підв’язати у контейнер, а також вказати шлях до Dockerfile, з якого і будуть білдити проект.

```yaml
version: '3.7'

services:
  operator_risk_management_bff:
    container_name: operator_risk_management_bff
    command: nodemon -L
    build:
      context: ../
      dockerfile: docker/Dockerfile.dev
      network: host
    environment:
      - CONFIG_DIR=docker
    volumes:
      - ../src/:/app/src
    ports:
      - "8001:8001"
```

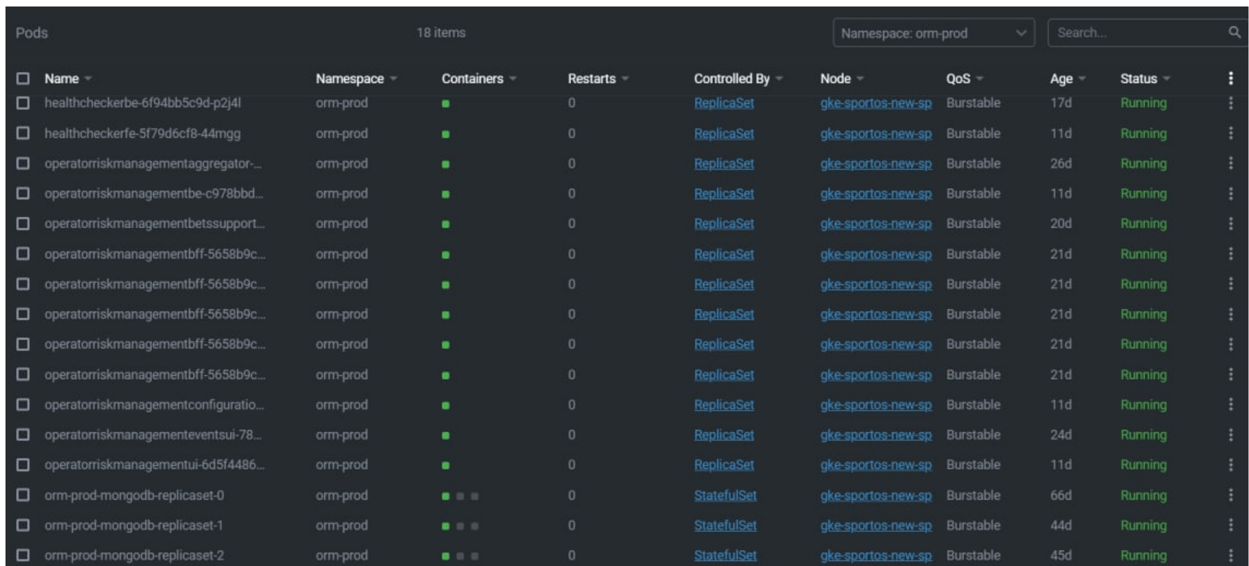
Рис. 3.9. Docker-compose файл для запуску інфраструктури

Перелік піднятих сервісів, їх імена та порти, які вони слухають можна побачити за допомогою команди: docker ps

```
PS C:\Users\Administrator\Desktop\testR> docker ps
CONTAINER ID   IMAGE                              COMMAND                  CREATED        STATUS
   PORTS                                                    NAMES
b87834800082f  docker_operator_risk_management_bff  "docker-entrypoint.s…"  24 hours ago   Up 3 minutes
   0.0.0.0:8001->8001/tcp                                   operator_risk_management_bff
a34f833e3633   zookeeper                          "/docker-entrypoint.…"   47 hours ago   Up 10 hours
   2183/tcp, 2888/tcp, 3888/tcp, 8080/tcp, 0.0.0.0:2183->2181/tcp   docker_zoo3_1
5fa645c9558d   zookeeper                          "/docker-entrypoint.…"   47 hours ago   Up 10 hours
   2888/tcp, 3888/tcp, 0.0.0.0:2181->2181/tcp, 8080/tcp              docker_zoo1_1
3842d8bbca98   zookeeper                          "/docker-entrypoint.…"   47 hours ago   Up 10 hours
   2182/tcp, 2888/tcp, 3888/tcp, 8080/tcp, 0.0.0.0:2182->2181/tcp   docker_zoo2_1
```

Рис. 3.10. Список піднятої інфраструктури

Сервіс масштабування Kubernetes дозволяю масштабувати контейнери, які запаковані у сервіси Docker.


Pods 18 items Namespace: orm-prod Search...

Name	Namespace	Containers	Restarts	Controlled By	Node	QoS	Age	Status
healthcheckerbe-6f94bb5c9d-p2j4l	orm-prod		0	ReplicaSet	gke-sportos-new-sp	Burstable	17d	Running
healthcheckerfe-5f79d6cf8-44mgg	orm-prod		0	ReplicaSet	gke-sportos-new-sp	Burstable	11d	Running
operatorriskmanagementaggregator-...	orm-prod		0	ReplicaSet	gke-sportos-new-sp	Burstable	26d	Running
operatorriskmanagementbe-c978bbd...	orm-prod		0	ReplicaSet	gke-sportos-new-sp	Burstable	11d	Running
operatorriskmanagementbetssupport...	orm-prod		0	ReplicaSet	gke-sportos-new-sp	Burstable	20d	Running
operatorriskmanagementbff-5658b9c...	orm-prod		0	ReplicaSet	gke-sportos-new-sp	Burstable	21d	Running
operatorriskmanagementbff-5658b9c...	orm-prod		0	ReplicaSet	gke-sportos-new-sp	Burstable	21d	Running
operatorriskmanagementbff-5658b9c...	orm-prod		0	ReplicaSet	gke-sportos-new-sp	Burstable	21d	Running
operatorriskmanagementbff-5658b9c...	orm-prod		0	ReplicaSet	gke-sportos-new-sp	Burstable	21d	Running
operatorriskmanagementbff-5658b9c...	orm-prod		0	ReplicaSet	gke-sportos-new-sp	Burstable	21d	Running
operatorriskmanagementconfiguratio...	orm-prod		0	ReplicaSet	gke-sportos-new-sp	Burstable	11d	Running
operatorriskmanagementeventsui-78...	orm-prod		0	ReplicaSet	gke-sportos-new-sp	Burstable	24d	Running
operatorriskmanagementui-6d5f4486...	orm-prod		0	ReplicaSet	gke-sportos-new-sp	Burstable	11d	Running
orm-prod-mongodb-replicaset-0	orm-prod		0	StatefulSet	gke-sportos-new-sp	Burstable	66d	Running
orm-prod-mongodb-replicaset-1	orm-prod		0	StatefulSet	gke-sportos-new-sp	Burstable	44d	Running
orm-prod-mongodb-replicaset-2	orm-prod		0	StatefulSet	gke-sportos-new-sp	Burstable	45d	Running

Рис. 3.11. Масштабування контейнерів BFF за допомогою сервісу Kubernetes

3.3. Розробка додатку

При розробці серверної частини було використано Node.js фреймворк Fastify.js. Сервер запускає index.js файл, який містить перелік конфігурацій раутингу та налаштувань. Fastify дозволяє, в залежності від переданному йому оточення (test, stage, prod) відключати та включати плагіни розширення роботи.

Ініціювання серверу відбувається шляхом реєстрації відповідних плагінів перед викликом методу listen(PORT, HOST), який створює веб-сервер на відповідному порту.

```
export const createServer = async ({ dbConfig }) => {
  const {
    getSwaggerConfig: swaggerConfig,
    getAuth0Config: auth0Config,
    getCorsConfig: corsConfig
  } = ConfigService;

  server
    .register(fastifyCors, corsConfig) FastifyInstance<Server, RawRequestDefaultExpr
    .register(fastifyCookies) FastifyInstance<Server, RawRequestDefaultExpression<Serve
    .register(fastifyAuth0, auth0Config) FastifyInstance<Server, RawRequestDefaultE
    .register(fastifySwagger, swaggerConfig) FastifyInstance<Server, RawRequestDe

    // Hooks
    .addHook(ReqResHookTypes.onRequest, Hooks.requestHooks.parseReqId)

    // Plugins
    .register(Plugins.dbPlugin, dbConfig) FastifyInstance<Server, RawRequestDefault
    .register(Plugins.servicesPlugin) FastifyInstance<Server, RawRequestDefaultExpre
    .register(Plugins.authPlugin, PRIVATE_ROUTES) FastifyInstance<Server, RawRe
    .register(Plugins.consumersPlugin);

  // API Routes
  registerMainRoutes(server);

  return server;
};
```

Рис. 3.12. Ініціалізація сервера

До необхідних плагінів для правильної роботи інформаційної системи потрібно віднести:

- плагін регістрації сервісів;
- плагін підключення споживачів повідомлень через message broker;
- реєстрації cookies та auth0;
- підключення до БД.

```
import fp from 'fastify-plugin';

import Database from '../utils/db';

import { DatabasesConfigType } from '../modules/config';
import { ServerType } from '../types';

const plugin = async (fastify: ServerType, opts: DatabasesConfigType, next: Function) => {
  const operatorMainDB = new Database(opts.operatorMainDB);
  await operatorMainDB.getDatabase(opts.operatorMainDB.database);

  fastify.decorate( property: 'db', operatorMainDB);
  next();
};

export default fp(plugin);
```

Рис. 3.13. Реєстрація плагіну підключення до БД

Для обробки та проксювання відповідних запитів з клієнтської частини необхідно реалізувати серверний раутинг. Методи та маршрути запитів групуються за сутностями Template, Events і т.п.

```
const registerMainRoutes = (fastify: ServerType, options :{} = {}) => {
  fastify.register( plugin: (fastify: ServerType, _opts: object, next: Function) => {
    /* eslint-disable */
    // prettier-ignore
    fastify
      .register(HealthRoutes,         { prefix: API.PREFIX.HEALTH,         logLevel: 'silent' })
      .register(EventsRoutes,         {                                    logLevel: 'silent' })
      .register(BetsRoutes,           { prefix: API.PREFIX.BETS,           logLevel: 'silent' })
      .register(SportsRoutes,         { prefix: API.PREFIX.SPORTS,         logLevel: 'silent' })
      .register(TemplatesRoutes,      { prefix: API.PREFIX.TEMPLATES,      logLevel: 'silent' })
      .register(LeagueGroupsRoutes,   { prefix: API.PREFIX.LEAGUE_GROUPS,  logLevel: 'silent' })
      .register(MasterLeaguesRoutes,  { prefix: API.PREFIX.MASTER_LEAGUES, logLevel: 'silent' })
      .register(MarketTypesRoutes,    { prefix: API.PREFIX.MARKET_TYPES,   logLevel: 'silent' })
      .register(LoggerRoutes,         { prefix: API.PREFIX.LOGS,           logLevel: 'silent' })
      .register(AccountRoutes,        { prefix: API.PREFIX.ACCOUNT,        logLevel: 'silent' })
      .register(OperatorRoutes,       { prefix: API.PREFIX.OPERATOR,       logLevel: 'silent' });
    /* eslint-enable */
    next();
  }, options);
};
```

Рис. 3.14. Плагін реєстрації серверного раутигу

Клієнтський раутинг складається з 3 частин:

- головної сторінки;
- всіх сторінкам, які відповідають схемі /назва_сторінки;
- сторінкам, які відповідають /сторінка/префікс_сторінки.

Відповідним регулярним виразом можна покрити всі можливі сторінки додатку. Якщо ж сторінка буде відсутня, сервер віддасть сторінку, яка відповідає статус– коду 404 Not Found.

3.4. СУБД MongoDB

Робота додатку напряму зв’зана з БД MongoDB. Створення відповідних сутностей, виставлення індексації полів може значно прискорити роботу додатку, додавши миттєву реакцію на дії користувача.

Проаналізувавши предметну область можна виділити 4 сутності:

- шаблон (Template);
- налаштування вірогідностей, обмежень
- подія (Event);
- ліг група (LeagueGroup);
- маркет (Market).

```
export interface TemplateType {
  _id: string;
  Name: string;
  SportId: number;
  IsActive: boolean;
  IsGlobal: boolean;
  IsLive: boolean;
  IsDefault: boolean;
  ParentTemplate: {
    _id: string;
    Name: string;
    Operator: string;
  } | null;
  IsBetBuilderEnabled?: boolean | null;
  UpdatedAt?: string | Date;
  CreatedAt?: string | Date;
  FeedMetaData?: {
    ID?: number;
    SiteID?: number;
    BranchID?: number;
    LeagueID?: number;
  };
  RemovedAt?: string | Date;
  IsRemoved?: boolean;
}
```

Рис. 3.15. Декларація сутності шаблон для БД

Правильно оформивши структуру сутностей, можна проставити відповідні ключі, для утворення зв’язків між ними.

Таблиця 3.1

Ключі таблиць

Шаблон	**Налаштування вірогідностей**	**Налаштування обмежень**
-Код шаблона **PK** - Ім'я - Спорт -Дата створення - Дата оновлення	-Код налаштування **PK** - Код шаблона **FK** - Виплата - Дата створення	- Код налаштування **PK** - Код шаблона **FK** - Максимальний виграш - Мінімальний виграш - Дата створення
Подія	**Маркет**	**ЛігГрупа**
-Код події **PK** - Код ліг групи **FK** - Ім'я - Спорт - Дата створення	-Код маркета **PK** - Код події **FK** - Ім'я - Налаштування - Дата створення	- Код ліг групи **PK** - Код шаблона **FK** - Спорт - Назва - Дата створення

Для кращого представлення зв'язків між сутностями потрібно представити E-R діаграму.

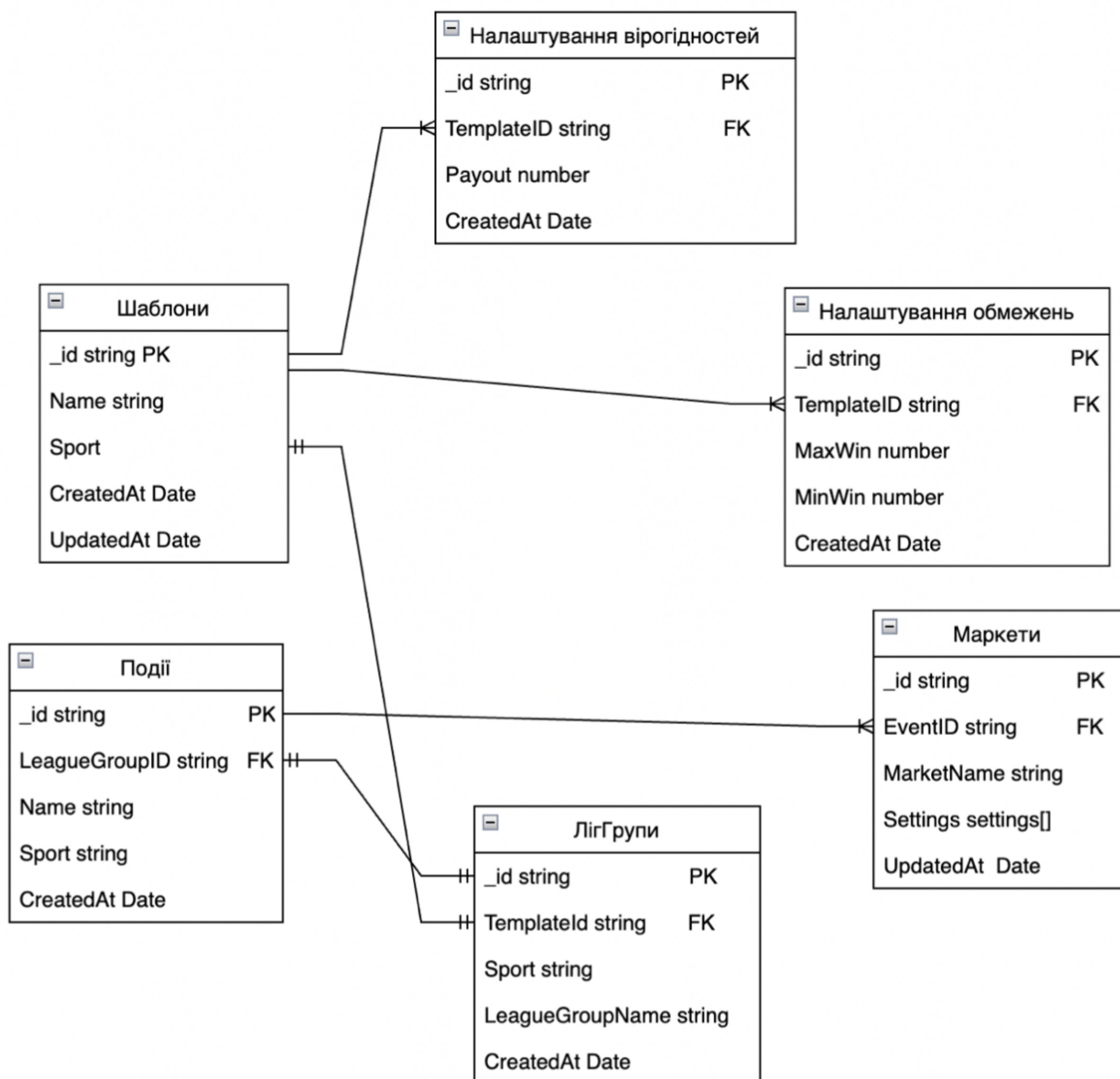


Рис. 3.16. Нормалізована E– R діаграма сутностей у БД

Так як робота розробляємої системи напряму залежить від швидкості пошуку сутностей гарною практикою є використання індексів (indexes) відповідних для відповідних полів сутності.

Для створення індексів використовуємо команду db.createIndex().

```
db.getCollection('Templates').createIndex({_id: 1})
```

Рис. 3.17. Створення індекса для колекції

Для виставлення індекса по зростанню треба виставити значення 1. Для використання індексу за спаданням використовуємо значення -1.

MongoDB надає можливість створювати TTL-індекс, вказавши після декларування поля, який буде використовуватися для індексації, параметр “expireAfterSeconds”, який видалить відповідні записи по полю, через час у секундах, який вказаний як значення даного поля.

```
db.getCollection('Templates').createIndex({CreatedAt: 1}, { expireAfterSeconds: 60});
```

Рис. 3.18. Використання TTL-індекса

3.5. Оновлення подій у реальному часі за допомогою SSE та MongoDB ChangeStreams

Для того, щоб досягти оновлення данних у реальному часі додаток використовує технологію Server Sent Events (SSE) з комбінацією з можливістю підписуватися на оновлення колекцій у БД через брокер повідомлень Pulsar.

MongoDB використовує концепцію Change Streams, яка реалізовує всі основні типи операцій з колекцією (insert, update, replace і т.п). Технологія використовується на сервері за допомогою бібліотеки mongo для мови JavaScript.

```
subscribeToCollectionUpdates = async (collectionName: string): Promise<void> => {
  const collection = await this.getCollection(this.db.defaultDatabaseName, collectionName);

  if (collectionName === Collections.MARKETS) this.pollUpdatedMarketBuffer();

  const pipeline = [ { $match: { $and: [ { operationType: {
    $in: collectionName === CollectionsForUpdates.EventSettings
      ? ['update', 'replace', 'insert']
      : ['update', 'replace']
  } }] }}];

  const stream = collection.watch(pipeline, options: { fullDocument: 'updateLookup' });

  stream.on( event: 'change', listener: async (change) => {
    const eventId = collectionName === 'Events' ? change.fullDocument._id : change.fullDocument.EventId;

    if (this.isClientExistForEventId(eventId)) {
      const payload = change.fullDocument;

      if (collectionName === CollectionsForUpdates.EventSettings) {
        this.recieveEventSettingsUpdates(change, eventId, payload);
      } else if (collectionName === CollectionsForUpdates.Markets) {
        this.recieveMarketUpdates(change, eventId, payload);
      } else {
        this.recieveEventUpdates(change, eventId, payload);
      }
    }
  });

  this.changeStreams[collectionName] = stream;
};
```

Рис. 3.19. Метод підписки на оновлення даних у колекції

Оновлення сутності транслюється на клієнтську частину за допомогою технології SSE. Технологія одностороння, тобто після того, як буде встановле з’єднання данні про оновлення сутностей будуть відправлятися з серверної частини на клієнтську, для можливості відправляти запити з клієнта на сервер треба використовувати інші доступні технології (HTTP, XMLHttpRequest і т.п).

```
export const establishSSE = (eventId?: string) => {
  const url = `${getBaseUrl()}/sse/events/${eventId}/updates`;

  source = new EventSource(url);

  source.addEventListener( type: 'error',  listener: e => {
    connectionCounter += 1;
    if (source && connectionCounter > SSE_CONNECTION_LIMIT_COUNT) source.close();
  });

  source.addEventListener( type: 'close',  listener: () => { connectionCounter = 0; source = null; });

  return source;
};

export const addEventListener = (type: any, cb: (event: MessageEvent) => void): void => {
  if (!source) throw new Error('EventSource is not initialized');

  source.addEventListener(
    type,
    listener: (msg: MessageEvent) => {
      if (!msg || !msg.data || msg.data === SSE_PING_MESSAGE) return;

      cb( event: { ...msg, data: JSON.parse(msg.data) });
    },
    options: false
  );
};
```

Рис. 3.20. Клієнтська підписка на серверні оновлення

Надходження оновлень можна побачити на UI а також, відкривши інформацію про запит. Запит не закінчується (оновлення постійно поступають по сокету) через постійний ping-сигнал “:keep-alive” для того щоб сервер не закрив сокет, якщо через 60с нові данні не надійдуть. Данні передаються з заголовком Content-Type: ‘text/event-stream’ як звичний текстовий формат передачі інформації.

× Headers EventStream Initiator Timing Cookies

Id	Type	Data
	message	{"messageType":"EVENT_UPDATES","payload":{"_id":"250424318465699840","BetslipLine":{"EN":"Washington Football Team vs Seattle Seahawks"},"BetslipLinePattern":{"Parameters":{"PhraseClass{0}...
	message	{"messageType":"EVENT_SETTINGS_UPDATES","payload":{"GameSettings":{"isBetBuilderEnabled":true,"isCashOutEnabled":true,"isExposureEnabled":false,"isVIPExcludeEnabled":false,"isWBComboEn...
	message	:keep-alive

Рис. 3.21. Надходження оновлень через сокет

Висновки до розділу 3

Розробка мікросервісної системи обробки даних у реальному часі складалася з:

1. Розробки серверної частини за допомогою Node.js (Fastify.js).
2. Клієнтська частина інтерфейсу користувача (React.js).
3. Контейнеризація та масштабування додатку за допомогою Docker та Kubernetes.
4. Розробка рівня доступу до даних за допомогою СУБД MongoDB.
   3.1. Структура таблиць та зв’язки між ними.
   3.2. Використання TTL-індексу для видалення непотрібних записів.
   3.3. Використання технології MongoDB Change Stream для підписки на оновлення даних.

Для розробки мікросервісної системи була використана мова JavaScript, серверний фреймворк Node.js та бібліотека для створення клієнтских інтерфейсів React.

За результатами тестування роботи мікросервісної системи було отримані наступні результати, швидкість роботи розробленої системи вища за відповідних аналогів на 20 %, система швидше відправляє оновлення. Системою можна користуватися багатьом користувачам одночасно через високу відмовостійкість, покращена реплікація була досягнена за допомогою сервісу Kubernetes.

ВИСНОВКИ

Розподілена система – це система, яка працює відразу на безлічі машин, що утворюють цілісний кластер. Кластер – це набір комп’ютерів/серверів, об’єднаних мережею, які взаємодіють між собою. Найважливіші плюси такого підходу розробки додатків – велика доступність та відмовостійкість. Можливості масштабування можна реалізувати за допомогою “Куба масштабованості”.

Основні переваги мікросервісної архітектури:

- Чіткий поділ на модулі, мікросервіси посилюють модульну структуру;
- Висока доступність. Деякі сервіси можуть не працювати, при цьому у цілому система буде доступною але буде працювати без відповідних сервісів;
- Можливість використовувати правильний інструмент для відповідного завдання;
- Незалежне розгортання через слабку зв'язаність сервісів.

Мікросервісна архітектура у високонавантажених додатках у реальному часі реалізує бізнес-логіку для управління ставками на спортивні події, а також контролює налаштування та ризики для операторів. Більше того, програмний продукт повинен мати можливість швидко масштабуватися та забезпечувати цілісність та точність даних. Система має доступ, і дозволяє користувачеві безпосередньо вносити зміни в деякі з налаштувань, як, наприклад, виплата по спортивній події, аналіз ризиків і заморозка ризикованих подій.

Перевагами проектованої системи є висока ефективність за рахунок:

- зменшення до мінімуму мережевих затримок під час оновлення даних;
- підвищення точності отримуваних данних, оскільки запропонований сервіс співпрацює з довіреними вендорами;

- динамічне та гнучке розгортання;
- швидка реплікація сервісів у “вузьких місцях”.

Порівняльний аналіз технологій для реалізації проектованої системи показав, що найбільш доцільним є вибір технологій JavaScript, Node.js, React.js, документна база данних MongoDB та архітектурний шаблон брокера повідомлень Pulsar, що є найкращим стеком технологій для мікросервісної системи обробки даних у реальному часі.

Використання зазначеного стеку технологій дозволяє:

- покращити стійкість системи;
- збільшити швидкість оновлення даних;
- забезпечити сервіс легким масштабуванням;
- полегшити розробку, користуючись однією мовою програмування для клієнтсьої та серверної частини.

Розробка мікросервісної системи обробки даних у реальному часі складалася з:

1. Розробки серверної частини за допомогою Node.js (Fastify.js).
2. Клієнтська частина інтерфейсу користувача (React.js).
3. Контейнеризація та масштабування додатку за допомогою Docker та Kubernetes.
4. Розробка рівня доступу до даних за допомогою СУБД MongoDB.
   3.1. Структура таблиць та зв’язки між ними.
   3.2. Використання TTL-індексу для видалення непотрібних записів.
   3.3. Використання технології MongoDB Change Stream для підписки на оновлення даних.

Для розробки мікросервісної системи була використана мова JavaScript, серверний фреймворк Node.js та бібліотека для створення клієнтських інтерфейсів React.

За результатами тестування роботи мікросервісної системи було отримані наступні результати, швидкість роботи розробленої системи вища за відповідних

аналогів на 20 %, система швидше відправляє оновлення. Системою можна користуватися багатьом користувачам одночасно через високу відмовостійкість, покращена реплікація була досягнена за допомогою сервісу Kubernetes.

СПИСОК БІБЛІОГРАФІЧНИХ ПОСИЛАНЬ ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Why use an SMS Text Messaging Service for Business? [Electronic resource]. Access mode: https://www.grouptexting.com/sms-for-business/(lastaccess:04.06.2020). – Title from the screen.

2. Mobile Country Code [Electronic resource]. – Access mode: https://ru.wikipedia.org/wiki/Mobile_Country_Code/(lastaccess:04.06.2020). – Title from the screen.

3. Mobile Network Code [Electronic resource]. – Access mode: https://ru.wikipedia.org/wiki/MNC/(lastaccess:04.06.2020). – Title from the screen.

4. Number Validation. Validate numbers, check porting status and more [Electronic resource]. – Access mode: https://www.sms77.io/en/products/number-validation/(lastaccess:05.06.2020). – Title from the screen.

5. HLR Lookup [Electronic resource]. – Access mode: https://www.amdtelecom.net/services/numbers-insight/hlr-lookup/(lastaccess: 04.06.2020). – Title from the screen.

6. JavaScript [Electronic resource]. – Access mode: https://developer.mozilla.org/en-US/docs/Web/JavaScript/(lastaccess: 04.06.2020). – Title from the screen.

7. React. A Javascript library for building user interfaces [Electronic resource]. – Access mode: https://reactjs.org/(lastaccess:04.06.2020). – Title from the screen.

8. PostgreSQL: The World's Most Advanced Open Source Relational Database [Electronic resource]. – Access mode: https://www.postgresql.org/(lastaccess:06.06.2020). – Title from the screen.

9. Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine [Electronic resource]. – Access mode: https://nodejs.org/en/(lastaccess:04.06.2020). – Title from the screen.

10. SQL Enumerated Types [Electronic resource]. – Access mode: https://doc.nuodb.com/nuodb/latest/reference-information/sql-reference-information/sql-data-types/sql-enumerated-types/(lastaccess: 04.06.2020). – Title from the screen.

11. Fastify. Fast and low overhead web framework, for Node.js [Electronic resource]. – Access mode: https://www.fastify.io/(lastaccess:04.06.2020). – Title from the screen.

12. Beginners Guide to ReactJS.js [Electronic resource]. – Access mode: https://medium.com/zenofai/beginners-guide-to-reactjs-3ca07f56d526/(lastaccess: 04.06.2020). – Title from the screen.

ДОДАТКИ

Додаток А

Код серверної частини

```
import {basename, resolve} from 'path'
import {Worker} from 'worker_threads'
import {APP_API_JOB, IS_OA, SERVER_API, WORKERS} from '../configs'
import {DBG, validate as vali, ERR, hub, queue$, LOG, workers$, sys$, store, compose, copyObj, sanitize, clearObj } from '../modules'
import {estimateJob, saveJob, setJob, uploadJob} from '../modules/job'
import {sendReport} from '../modules/report'
import {periodic} from 'most'
import request from 'superagent'

const reportArr = [],
  uploadArr = [],
  validateArr = []

let isReporting = false,
  isUploading = false,
  isValidating = false

const checkTitle = () => {
  const job = store.get('job')
  let {title, file: {name}} = job

  if (!title)
    title = String(name).split('.').shift()

  store.set('job', {title}, 'JOB')

  return job
}
```

```
const approveJob = async (saveJobByOA) => {

  store.set('job', {pending: true, status: 'validate'}, 'JOB')

  const status = 'approve'
  const job = IS_OA ? compose(copyObj, checkTitle)() : saveJobByOA

  const title = IS_OA ? sanitize.trim(job.title) + new Date().getTime() : sanitize.trim(job.title)

  if (vali.isLength(title, {min: 3, max: 100})) {

    const isExits = await request
      .post(SERVER_API + APP_API_JOB)
      .set('Content-Type', 'application/json')
      .send({title})
      .then(({body}) => body)
      .catch(ERR)

    LOG({isExits})

    job.id = IS_OA ? job.title : job.job_id



    clearObj(job, ['spark', 'processing', 'columns', 'schema', 'config', 'error', 'pending'])

    isExits && !IS_OA
      ? store.set('job', {title, pending: false, error: 'Not Unique job Name'}, 'JOB')
      : hub.emit('WORKER', 'job-approve', {job: {...job, status, title}})

  } else {
    store.set('job', {title, pending: false, error: 'job Name must be at least 3 characters'}, 'JOB')
  }
}

const WorkerMsg = ({id, cb}) => data => {
```

```
    if (data.job && id !== 'REPORT')
      store.set('job', data.job)

    if(IS_OA){
      if (data && data.job && data.job.estimate && !data.job.error && data.job.raw && id !== 'REPORT') {
        console.log({DA: data.job.raw})
        approveJob(data.job).catch(e => console.log('ERROR', e))
      }
    }

    return cb(data)
  }
  const WorkerError = ({id}) => err => ERR(WORKER ${id} Error, err)
  const WorkerExit = ({id}) => code => code ? ERR(new Error(Worker ${id} exit code:${code})) : DBG(Worker ${id} exit code:${code})
  const WorkerOnLine = ({id}) => () => LOG(WORKER ${id} Online)


  const WorkerReportExit = ({id}) => code => {
    isReporting = false
    return code ? ERR(new Error(Worker ${id} exit code:${code})) : DBG(Worker ${id} exit code:${code})
  }

  const WorkerReportOnLine = ({id}) => () => {
    isReporting = true
    return LOG(WORKER ${id} Online)
  }

  const WorkerUploadExit = ({id}) => code => {
    isUploading = false
    return code ? ERR(new Error(Worker ${id} exit code:${code})) : DBG(Worker ${id} exit code:${code})
  }

  const WorkerUploadOnLine = ({id}) => () => {
    isUploading = true
    return LOG(WORKER ${id} Online)
```

```
}

const WorkerValidateExit = ({id}) => code => {
   isValidating = false
   return code ? ERR(new Error(Worker ${id} exit code:${code})) : DBG(Worker ${id} exit code:${code})
}

const WorkerValidateOnLine = ({id}) => () => {
   isValidating = true
   return LOG(WORKER ${id} Online)
}

const startWorker = (path, cb) => workerData => {
   const
      id = basename(path).split('.').shift().toUpperCase(),
      w = new Worker(path, {workerData}),
      arg = {id, path, cb, workerData}

   w.on('message', WorkerMsg(arg))
   w.on('error', WorkerError(arg))
   w.on('online', WorkerOnLine(arg))
   w.on('exit', WorkerExit(arg))
   return w
}

const startAsyncWorker = ({workerData, path, cb, fnOnExit, fnOnOnline}) => {
   const
      id = basename(path).split('.').shift().toUpperCase(),
      w = new Worker(path, {workerData}),
      arg = {id, path, cb, workerData}

   w.on('message', WorkerMsg(arg))
   w.on('error', WorkerError(arg))
   w.on('exit', fnOnExit(arg))
   w.on('online', fnOnOnline(arg))
}
```

```
const
  upload = resolve(WORKERS, 'cmj', 'upload.js'),
  validate = resolve(WORKERS, 'cmj', 'validate.js'),
  estimate = resolve(WORKERS, 'cmj', 'estimate.js'),
  report = resolve(WORKERS, 'cmj', 'report.js'),
  mailer = resolve(WORKERS, 'cmj', 'mailer.js')

const startEstimateWithProductId = ({estimates, job, tasks, spark}) => {
  const {product_id: rate_plan_id} = store.get({cid: 'balance'}).findOne({id: job.user_id})
  return startWorker(estimate, setJob)({estimates, job, tasks, spark, rate_plan_id})
}
const workers = new Map([
  ['job-upload', workerData => hub.emit('QUEUE', {type: 'upload', workerData})],
  ['job-validate', workerData => hub.emit('QUEUE', {type: 'validate', workerData})],
  ['job-estimate', IS_OA ? startEstimateWithProductId :  startWorker(estimate, setJob)],
  ['job-approve', saveJob],
  ['job-report', workerData => hub.emit('QUEUE', {type: 'report', workerData})],
  ['mailer', startWorker(mailer, LOG)]
])

const arrLookup = () => {
  if (!isReporting) {
    if (reportArr.length && !isReporting) {
      let {workerData} = reportArr.shift()
      startAsyncWorker({
        workerData,
        path: report,
        cb: sendReport,
        fnOnExit: WorkerReportExit,
        fnOnOnline: WorkerReportOnLine
      })
    }
  }

  if (!isUploading) {
    if (uploadArr.length && !isUploading) {
```

```
            let {workerData} = uploadArr.shift()
            startAsyncWorker({
                workerData,
                path: upload,
                cb: uploadJob,
                fnOnExit: WorkerUploadExit,
                fnOnOnline: WorkerUploadOnLine
            })
        }
    }

    if (!isValidating) {
        if (validateArr.length && !isValidating) {
            let {workerData} = validateArr.shift()
            startAsyncWorker({
                workerData,
                path: validate,
                cb: estimateJob,
                fnOnExit: WorkerValidateExit,
                fnOnOnline: WorkerValidateOnLine
            })
        }
    }
}

periodic(1000)
    .since(sys$)
    .observe(arrLookup)
    .catch(ERR)

queue$
    .observe(el => {
        switch (el.type) {
            case 'report':
                reportArr.push({workerData: el.workerData})
                break
```

```
case 'upload':
            uploadArr.push({workerData: el.workerData})
            break
         case 'validate':
            validateArr.push({workerData: el.workerData})
            break
      }
   })
   .catch(ERR)

workers$
   .observe(([worker, data]) => workers.get(worker)(data))
   .catch(ERR)
```

Додаток Б

Код клієнтськоï частини

```
import {Component, format, getPager, store, linkEvent, sys$} from '../../modules'
import {Icon, JobComment, JobReport, Pager, QueueInfo} from '../../components'
import {setJobStatus} from '../../actions'

const Nav = ({coll, job}) =>
  job && coll.count() > 1
    ? coll.chain()
      .simplesort('created')
      .mapReduce(({id}) => id, r => <Pager {...getPager('jobs', r, job.id)} />)
    : null

const BarMap = new Map([
  ['pause', {label: 'pause', ico: 'pause'}],
  ['unhealthy', {label: 'restart', ico: 'play'}],
  ['ready', {label: 'restart', ico: 'play'}],
  ['stopped', {label: 'stop', ico: 'stop'}],
])

const ControlBarItem = ({id, status, disabled}) => {
  const {label, ico} = BarMap.get(status)
  return <button disabled = {disabled} className='ic' onClick={linkEvent({id, status}, setJobStatus)}><span>{label}</span><Icon id={ico}/>
  </button>
}

const ControlBar = ({stat, job}) => {

  const {id, status} = job
  let dom = null

  const isDisabled = stat.length ? !
    stat.filter(e => e.id === job.provider)[0].isHealth : false
```

```
    if (['processing', 'ready', 'pause', 'unhealthy'].includes(status)) {

      const action =
        ['pause', 'unhealthy'].includes(status)
        ? <ControlBarItem disabled={isDisabled} id={id} status='ready'/>
        : <ControlBarItem id={id} status='pause'/>

      const stop = <ControlBarItem id={id} status='stopped'/>

      dom = <control-bar>{action}{stop}</control-bar>
    }

    return dom
  }

  const Job = ({job, stat}) => {
    let {file, validation, created, started, completed, statistic, status, estimate, queue} = job
    const size = format.fileSize(file.size)

    const {all, invalid, valid, duplicate} = validation
    const {processed, left, cost, errors, repeats, roaming, no_roaming, ported, no_ported, absent, no_absent, active, no_active} = statistic
    const costProcessing = cost ? <h4>{format.currency(cost)}</h4> : null
    const Q = queue && ['ready', 'hold'].includes(status) ? <QueueInfo queue={queue}/> : null

  return <job>
      <card>
        <section>
          <column>
            <file-info>
              <h2>{file.name}</h2>
              <div>{size}</div>
              <div>{file.ext}</div>
            </file-info>
          </column>
```

```
  <column>
    <ul>
      <li><b>all:</b><span>{all}</span></li>
      <li><b>valid:</b> <span>{valid}</span></li>
      <li><b>invalid:</b><span>{invalid}</span></li>
      <li><b>duplicate:</b> <span>{duplicate}</span></li>
    </ul>
  </column>
  <column>
    <status>
      <h3>{status}</h3>
      {costProcessing}
      <estimate>estimate: {format.currency(estimate)}</estimate>
      <div>{Q}</div>
      <ControlBar stat={stat} job={job}/>
    </status>
  </column>
</section>
<section>
  <column>
    <ul>
      <li><b>created:</b><span>{format.date(created)}</span></li>
      <li><b>started:</b><span>{started ? format.date(started) : null}</span></li>
      <li><b>completed:</b><span>{completed ? format.date(completed) : null}</span></li>
      <li>
        <b>processing:</b><span>{completed ? format.dateDistanceStrict(started, completed) : null}</span>
      </li>
    </ul>
  </column>
  <column>
    <ul>
      <li><b>ported:</b><span>{ported}</span>/<span>{no_ported}</span></li>
      <li><b>roaming:</b><span>{roaming}</span>/<span>{no_roaming}</span></li>
      <li><b>absent:</b><span>{absent}</span>/<span>{no_absent}</span></li>
      <li><b>active:</b><span>{active}</span>/<span>{no_active}</span></li>
    </ul>
```

```
                </column>
                <column>
                    <ul>
                        <li><b>left:</b><span>{left}</span></li>
                        <li><b>processed:</b><span>{processed}</span></li>
                        <li><b>errors:</b><span>{errors}</span></li>
                        <li><b>repeats:</b><span>{repeats}</span></li>
                    </ul>
                </column>
            </section>
            <JobReport job={job}/>
        </card>
        <JobComment job={job} />
    </job>
}

export class job extends Component {

    componentDidMount() {
        sys$.observe(() => this.forceUpdate())
    }

    render() {

        const cid = 'jobs'

        const {sub} = store.get('router')
        const {stats} = store.get( 'sys', this)
        const coll = store.get({cid}, this)

        const job = coll.by('id', sub)

        const title = job ? job.title : 'job not found'

        const navProps = {coll, job}
```

```
        return <main>
            <header className='complex'>
                <h1>{title}</h1>
                <Nav {...navProps}/>
            </header>
            {job ? <Job stat={stats} key={job.id} job={job}/> : null}
        </main>
    }
}
```