НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
Інститут прикладного системного аналізу
Кафедра системного проектування

До захисту допущено:
Завідувач кафедри
_____ В.Є. Мухін
«___»_____20__ р.

Дипломна робота

на здобуття ступеня бакалавра

за освітньо-професійною програмою «Інтелектуальні сервіс-орієнтовані розподілені обчислювання»

спеціальності 122 «Комп'ютерні науки»

на тему: «Хмарні сервіси з використанням методів асинхронного програмування»

Виконав:
студент IV курсу, групи ДА-81
Переяславський Сергій Костянтинович _____

Керівник:
к.т.н., доцент, Харченко Костянтин Васильович _____

Консультант з економічного розділу:
к.е.н., доцент, Рощина Надія Василівна _____

Консультант з нормконтролю:
к.т.н., доцент, Кирюша Богдан Анатолійович _____

Рецензент:
к.т.н, доцент, Тимощук Оксана Леонідівна _____

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2022 року

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Інститут прикладного системного аналізу

Кафедра системного проектування

Рівень вищої освіти – перший (бакалаврський)

Спеціальність – 122 «Комп'ютерні науки»

Освітньо-професійна програма «Інтелектуальні сервіс-орієнтовані розподілені обчислювання»

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ В.Є. Мухін
«___»_____20__ р.

ЗАВДАННЯ
на дипломну роботу студенту
Переяславському Сергію Костянтиновичу

1. Тема роботи «Хмарні сервіси з використанням методів асинхронного програмування», керівник роботи Харченко Костянтин Васильович, доцент, к.т.н., затверджені наказом по університету від «____» _____ 20__ р. №_____

2. Термін подання студентом роботи 15.06.2022

3. Вихідні дані до роботи Java Core, JVM, Spring Boot, Spring Framework, Spring MVC, Spring WebFlux, MongoDB

4. Зміст роботи

1. Вступ

2. Дослідження засобів реактивного програмування на мові Java

3. Аналіз застосування методів реактивного програмування при побудові інформаційних систем

4. Аналіз наявних рішень реактивних систем на мові Java

5. Застосування реактивних систем в мікросервісній архітектурі

6. Використання хмарної інфраструктури для розгортання сервісу

7. Функціонально-вартісний аналіз програмного продукту

8. Висновки

5. Перелік ілюстративного матеріалу (із зазначенням плакатів, презентацій тощо)

1. Презентація роботи

6. Консультанти розділів роботи*

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Економічний	доцент, к.е.н., Рощина Н.В.		

7. Дата видачі завдання 01.02.2022

Календарний план

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітка
1	Отримання завдання	01.02.2022	
2	Складання плану роботи	15.02.2022	
3	Дослідження наукової літератури по темі дипломної роботи	01.03.2022	
4	Огляд засобів реактивного програмування на мові Java	15.03.2022	
5	Аналіз наявних рішень реактивних систем на мові програмування Java	05.04.2022	
6	Дослідження застосування фреймворку Spring в реактивних системах	26.04.2022	
7	Реалізація додатку для методичного використання	17.05.2022	
8	Оформлення дипломної роботи	31.05.2022	
9	Оформлення допуску до захисту і подання роботи	15.06.2022	

Студент С. К. Переяславський

Керівник К. В. Харченко

* Якщо визначені консультанти. Консультантом не може бути зазначено керівника дипломної роботи.

АНОТАЦІЯ

бакалаврської роботи Переяславського Сергія Костянтиновича

на тему «Хмарні сервіси з використанням методів асинхронного програмування»

Метою даної роботи є дослідження методів асинхронного програмування, їхній аналіз, порівняння та застосування при побудові додатків які є одним з компонентів в мікросервісній архітектурі та розгортання сервісу в хмарній системі.

В процесі виконання роботи були досліджені особливості та стандарти реактивного підходу до програмування, була розглянута стандартна реалізація на мові програмування Java, описані основні концепції, які застосовуються при реактивному підході до побудови систем, розглянуті існуючі реалізації, проведено порівняння, визначені їхні переваги та недоліки.

В результаті дипломної роботи на прикладі побудованого мікросервісного додатку було розглянуто застосування фреймворку Spring у поєднанні з реактивними бібліотеками, проведений аналіз підходу, описані характеристики фреймворку, особливості застосування, тестування та розміщення серед інших сервісів. Окремо було розглянуто розгортання сервісу в хмарній системі Heroku з використанням бази даних MongoDB. Цю роботу рекомендується використовувати у якості методичних рекомендацій при розробці інформаційних систем, які розміщуються в реактивному мікросервісному середовищі.

Загальний обсяг роботи: 115 сторінок, 32 рисунки, 7 таблиць, 25 посилань, 1 додаток.

Ключові слова: реактивне програмування, фреймворк Spring, Spring WebFlux, Project Reactor, брокер повідомлень, мікросервісний додаток, хмарна інфраструктура, Heroku, MongoDB.

ABSTRACT

on Serhii Pereiaslavskyi bachelor's thesis

«Cloud services based on asynchronous programming methods»

The goal of this work is to research the tools of asynchronous programming, their analysis, comparison and usage in construction of applications that are the components of microservice architecture and service deployment in the cloud system.

The paper contains the features and standards of the reactive approach to programming, the standard implementation in Java programming language, the basic concepts used in the reactive approach while building the systems, existing implementations were considered, comparisons were made, the advantages and disadvantages of such an approach were identified.

As a result of the thesis on the example of the built microservice application was considered the use of Spring framework in combination with reactive libraries, approach analysis, framework characteristics, application features, testing approach and its placement among other services. The deployment of the service in the Heroku cloud system using the MongoDB database was considered separately. It is recommended to use this work as methodological recommendations in the development of information systems that are used in a reactive microservice environment.

The total volume of work: 115 pages, 32 figures, 7 tables, 25 links, 1 appendix.

Keywords: reactive programming, Spring framework, Spring WebFlux, Project Reactor, message broker, microservice application, cloud infrastructure, Heroku, MongoDB.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ОС – Операційна Система

СУБД – Системи Управління Базами Даних

ЦП – Центральний Процесор

API – Application Programming Interface

CRUD – Create, Read, Update, Delete

HTTP – HyperText Transfer Protocol

JDBC – Java Database Connectivity

JSON – JavaScript Object Notation

JVM – Java Virtual Machine

REST – Representational State Transfer

SOA – Service Oriented Architecture

SQL – Structured Query Language

ВСТУП

Асинхронне програмування — це форма паралельного програмування, яка дозволяє інструкціям виконуватися окремо від основного потоку програми. Коли виконання інструкцій завершено, про це повідомляється потік, в якому необхідно було обробити інструкції (а також про те, чи були вони завершені або невдало). Використання цього засобу має численні переваги, наприклад, покращена продуктивність програми та покращена швидкість реагування на запити. Асинхронна модель дозволяє одночасно виконувати декілька задач.

Реактивне програмування — це парадигма асинхронного програмування, яка орієнтована на асинхронне оброблення вхідного потоку даних і поширення подій, які виникають в процесі обробки [1].

Реактивне програмування часто асоціюють з паралельними обчисленнями і високою продуктивністю настільки, що важко розділити ці поняття, хоча насправді вони в принципі абсолютно різні. Це неминуче призводить до плутанини. Реактивне програмування також часто поєднують з функціональним реактивним програмуванням. Реактивне програмування — це стиль архітектури, що включає інтелектуальну маршрутизацію та реакцію на події, які поєднуються для зміни поведінки. Резонанс застосування такого підходу з’явився (не випадково) разом із застосуванням мікросервісів і повсюдним поширенням багатоядерних процесорів.

Основна ідея реактивного програмування полягає в тому, що є певні типи даних, які представляють значення "змінювані з плином часу". Крім того, самі обчислення, що включають змінювані з часом дані, також є значенням що змінюються з часом.

Завдяки реактивному програмуванню створюється програмне забезпечення, яке реагує на події замість того, щоб вимагати вводу від користувачів. Подія – це сигнал про те, що щось сталося. Загальновизнано, що події є сигналами

«реального часу», тобто вони генеруються одночасно з умовою, про яку вони сигналізують, і вони також повинні оброблятися в режимі реального часу. Ці події найкраще візуалізувати як «потоки», які можуть протікати через кілька елементів обробки, бути зупиненими та обробленими на даному етапі шляху, або розгалужуються та паралельно обробляються. У більшості випадків ця обробка чутлива до часу, а це означає, що програми вимагають іншого стилю програмування - саме так і з’явився реактивний підхід до програмування.

Реактивне програмування та реактивні системи, з якими воно має справу, складаються з комбінації функцій «спостерігача» та «обробника». Перший розпізнає важливі умови або зміни та генерує повідомлення, щоб повідомити про зміни, на які очікує обробник, а другий належним чином їх обробляє. Припущення в реактивному програмуванні полягає в тому, що немає контролю над кількістю або часом подій, тому програмне забезпечення має бути стійким і високомасштабованим, щоб керувати змінними навантаженнями.

Реактивне програмування дозволяє асинхронно обробляти та комбінувати потоки елементів даних, що надходять з різних систем і джерел. Насправді програми, написані відповідно до цієї парадигми, реагують на елементи даних у міру їх появи, що дозволяє їм мати швидший відклик у взаємодії з користувачами. Більше того, реактивний підхід можна застосувати не тільки до побудови окремого компонента або програми, а й до координації багатьох компонентів у цілісну реактивну систему. Системи, сконструйовані таким чином, можуть обмінюватися та маршрутизувати повідомлення в різних мережевих умовах і забезпечувати доступність при великому навантаженні, враховуючи збої та відключення

1 ДОСЛІДЖЕННЯ ЗАСОБІВ РЕАКТИВНОГО ПРОГРАМУВАННЯ НА МОВІ JAVA

1.1 Маніфест реактивного підходу до програмування

Особливості та переваги, які характеризують реактивні програми та системи, викристалізовані в так званому «Реактивному маніфесті» - документі, який описує основні принципи реактивного підходу при програмуванні [2]:

- Реактивні системи повинні мати малий відклик [2]
  Система повинна реагувати вчасно. Системи, що реагують, зосереджені на забезпеченні швидкого та стабільного часу реагування, тому вони забезпечують стабільну якість обслуговування.
- Реактивні системи повинні бути стійкими [2]
  У випадку, якщо система зіткнеться з будь-яким збоєм, вона повинна реагувати. Стійкість досягається шляхом реплікації, ізоляції та делегування. Помилки можуть виникати в кожному з компонентів, тому, коли виникає помилка, вона не має впливати на інші компоненти або систему в цілому.
- Реактивні системи повинні бути еластичними [2]
  Реактивні системи можуть реагувати на зміни та залишатися з малим відкликом при різному робочому навантаженні. Вони досягають еластичності завдяки апаратним і програмним платформам.
- Реактивні системи повинні оперувати повідомленнями [2]
  Щоб встановити принцип стійкості, реактивні системи повинні встановити межу між компонентами, покладаючись на асинхронну передачу повідомлень.

Кожна з цих характеристик є важливою при проектуванні реактивних систем. Усі вони залежать одна від одної, але не як шари стандартної багаторівневої

архітектури, навпаки, вони описують властивості, які застосовуються у всьому стеку технологій, де використовуються реактивний підхід.

1.2 Види реактивності в системах

Реактивність на програмному рівні

Основна особливість реактивного програмування для компонентів прикладного рівня дозволяє виконувати інструкції асинхронно. Асинхронна і неблокуюча обробка потоків подій є важливою для максимізації швидкості використання сучасних багатоядерних процесорів і потоків, які змагаються за можливість їхнього використання. Для досягнення цієї мети реактивні фреймворки та бібліотеки поділяють потоки (відносно дорогі та дефіцитні ресурси) серед більш легких конструкцій, таких як ф'ючерси (CompletableFuture); актори (Actors); і (частіше) цикли подій, які надсилають послідовність зворотних викликів, призначених для агрегації, трансформації та керування подіями, які підлягають обробці.

Ці методи мають перевагу не тільки в тому, що вони дешевші, ніж потоки, але й з точки зору розробників: вони підвищують рівень абстракції реалізації паралельних та асинхронних додатків, дозволяючи розробникам зосередитися на бізнес-вимогах замість того, щоб мати справу. з типовими проблемами низькорівневих проблем багатопоточності, такими як синхронізація, умови перегонів (race conditions) та взаємного блокування (deadlocks) [3].

Реактивність на системному рівні

Реактивна система — це архітектура програмного забезпечення, яка дозволяє кільком додаткам працювати як єдина узгоджена, стійка платформа, а також дозволяє цим додаткам бути достатньо відокремленими, щоб, коли одна з них виходить з ладу, це не похитнуло стійкість всієї системи. Основна відмінність

між реактивними додатками та системами полягає в тому, що перший тип зазвичай виконує обчислення на основі потоків даних і називаються вони керованими подіями. Останній тип призначений для створення додатків і полегшення спілкування між ними.

Інша важлива відмінність між повідомленнями та подіями полягає в тому, що повідомлення спрямовуються до визначеного єдиного місця призначення, тоді як події — це факти, які будуть отримані компонентами, які зареєстровані для їх спостереження. У реактивних системах також важливо, щоб ці повідомлення були асинхронними, щоб операції надсилання та отримання були відокремлені від відправника та одержувача відповідно. Це роз'єднання є вимогою для повної ізоляції між компонентами і є основоположним для підтримки системи реагування як на збої (стійкість), так і на велике навантаження (еластичність).

Реактивність в Java

Ідея реактивності в Java побудована на патерні проектування Observer [4]:

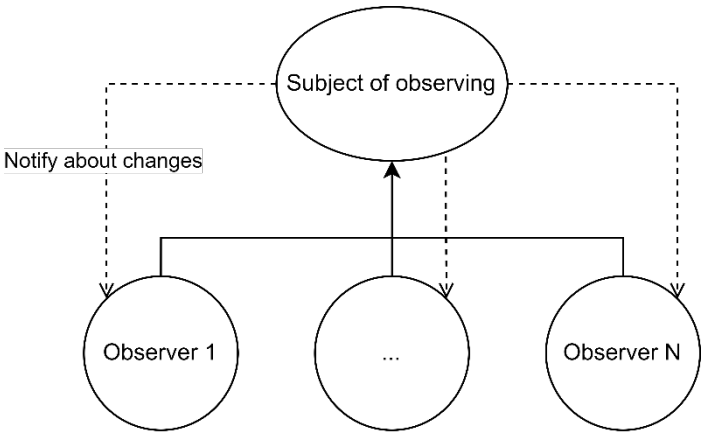

Рис. 1.1 – схематичне зображення патерну проектування Observer

Суб'єкт, за яким стежать спотерігачі (Observer) називається видавцем (той, за ким «слідкують» – Observable) – визначає залежність один-до-багатьох між об'єктами (багато підписників, один видавець) при якій зміна

спостережуваного суб'єкту повідомляється усім, хто спостерігає за ним, після чого відбувається обробка відповідно до описаних інструкцій.

Цей шаблон полегшує одночасні операції, оскільки його не потрібно блокувати під час очікування поки видавець випустить елементи.

Контракт Observer вимагає імплементації таких методів:

- onNext: щоразу, коли Observable публікує подію, цей метод викликається у Observer'а, який приймає в якості параметра об'єкт, що виробляється, щоб над ним була виконана певна дія
- onCompleted: цей метод викликається після останнього виклику методу onNext, що вказує на те, що послідовність подій, пов'язаних із Observable, завершена і в ньому не виявлено жодних помилок
- onError: цей метод викликається, коли при публікації подій сталася помилка

Розглянемо простий приклад з використанням бібліотеки io.reactivex.rxjava2:rxjava:

```
  public static void main(String[] args) {
    Observable<Long>   observableDataStream   =   Observable.interval(100,
TimeUnit.MILLISECONDS)
        .take(5)
        .share();

    System.out.println("First subscribed");
    observableDataStream.subscribe(item     ->     System.out.printf("First
received: {%s} %n", item));
    sleep(200);

    System.out.println("Second subscribed");
    observableDataStream.subscribe(item    ->    System.out.printf("Second
received: {%s} %n", item));
    sleep(1000);
  }

  private static void sleep(long millis) {
    try {
      TimeUnit.MILLISECONDS.sleep(millis);
    } catch (InterruptedException ignored) {}
  }
```

В даному прикладі імітується 1 виробник елементів, на який підписуються 2 споживачі, які будуть друкувати стрічку тексту використовуючи отриманий

елемент. Споживачі будуть отримувати елемент кожні 100 мілісекунд. Перший споживач приступить до очікування елементів від виробника одразу, другий – через 200 мілісекунд після першого. Перелік `observableDataStream` можна вважати потоком подій, які будуть оброблятися переданим споживачем. Для кожного підписника при публікації елемента у потоці викликається метод `.onNext()`, в який передається елемент, після чого над ним виконуються операції відповідно до переданого споживача (в даному прикладі – друк у термінал).

Результат виконання:

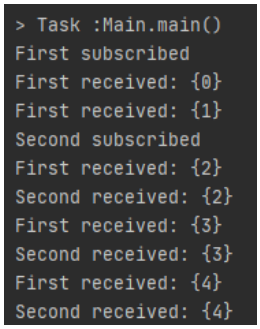

Рис. 1.2 – Результат виконання сценарію

Дані, які отримує споживач, можуть бути заздалегідь підготовленими для нього. Для цього можна використовувати оператори, які будуть застосовуватись до кожного елемента послідовності – це можуть бути предикати (пропускати елемент якщо він не задовольняє умову) – filter, трансформації – map (обробка/зміна даних), агрегація даних – reduce і т.д.

Попередній приклад був розширений завдяки додаванню операторів при побудові об’єкта Observable:

```
.filter(item -> item % 2 == 0)
.map(item -> item * 10);
```

які у зв’язці між собою випускають в 10 разів більший парний елемент послідовності від 0 до 9:

```
> Task :Main.main()
First subscribed
First received: {0}
Second subscribed
First received: {20}
Second received: {20}
First received: {40}
Second received: {40}
First received: {60}
Second received: {60}
First received: {80}
Second received: {80}
```

Рис. 1.3 – Результат виконання сценарію

Описуваний функціонал можна розширювати, передаючи як аргументи методу підписки відповідні обробники при таких подіях як: поява елемента, помилки виконання (exception), завершення публікацій від виробника, підписка чергового споживача:

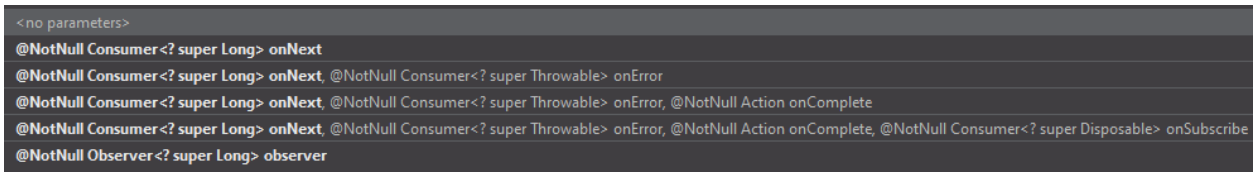

Рис. 1.4 – Аргументи методу підписки

Модифікуємо приклад так, щоб помилка сталася після шостого елемента з послідовності, відповідно додаючи необхідну трансформацію, в якій вона відбудеться і обробник для ситуації, коли сталася помилка, в якому буде друкуватись клас помилки і її повідомлення:

```
.map(item -> {
          if (item > 5) {
            throw new RuntimeException(format("Error on item: {%s}",
item));
          }
          return item;
        }
)

observableDataStream.subscribe(
item -> System.out.printf("First received: {%s} %n", item),
ex -> System.out.printf("Error occurred (1): {%s}, message: {%s} %n",
ex.getClass(), ex.getMessage()));

observableDataStream.subscribe(
```

```
item -> System.out.printf("Second received: {%s} %n", item),
ex -> System.out.printf("Error occurred (2): {%s}, message: {%s} %n",
ex.getClass(), ex.getMessage()));
```

```
> Task :Main.main()
First subscribed
First received: {0}
Second subscribed
First received: {1}
First received: {2}
Second received: {2}
First received: {3}
Second received: {3}
First received: {4}
Second received: {4}
First received: {5}
Second received: {5}
Error occurred (1): {class java.lang.RuntimeException}, message: {Error on item: {6}}
Error occurred (2): {class java.lang.RuntimeException}, message: {Error on item: {6}}
```

Рис. 1.5 – Результат обробки помилок

Можна побачити, шо штучно згенерована помилка була оброблена відповідно до переданих обробників помилок при підписці для споживачів – у терміналі надрукувався клас помилки, яка сталася і повідомлення, яке вона несе.

Використання реактивного програмування з Java має переваги. Воно дозволяє програмам Java демонструвати вищу продуктивність при менших вимогах до пам’яті та ресурів. Це стало можливим завдяки уникненню блокування викликів, які призводять до перемикання контексту в ОС.

Реактивне програмування працює на Java, покладаючись на потоки даних, які утворюють основу програми. Вони передають виклики, повідомлення, події та помилки в процесі обробки.

Реактивне програмування дозволяє спостерігати за потоками, а потім реагувати на події, коли вони з’являються. Розробники можуть створювати потоки для всього, що може змінитися в коді, наприклад подій в кешу або зміна кількості кліків миші, щоб зробити програму асинхронною.

Існують різні бібліотеки, які можна використовувати для застосування реактивного програмування в програмі Java.

1.3 Висновки до розділу

У даному розділі були визначені поняття реактивного програмування, особливості застосування, його становлення, був наданий перелік правил, які характеризують реактивні програми та системи, на базовому рівні були описані шаблони взаємодії в реактивних програмах між об’єктами, які несуть в собі потоки даних, і тими, які ці потоки даних споживають, були наведені приклади реалізацій і використання існуючого API в Java і продемонстровані можливості низькорівневих реалізацій інструментів мови програмування.

2 АНАЛІЗ ЗАСТОСУВАННЯ МЕТОДІВ РЕАКТИВНОГО ПРОГРАМУВАННЯ ПРИ ПОБУДОВІ ІНФОРМАЦІЙНИХ СИСТЕМ

Реактивне програмування все частіше застосовується при розробці інформаційних систем. Це парадигма програмування, заснована на передачі даних з одного джерела або з кількох джерел. Концепція не завжди зрозуміла, а її важливість недооцінюється. Як видно з назви, реактивне програмування орієнтоване на реакцію, потік даних і принцип реагування на збудження, тобто кожне збудження пов’язане зі своїми наслідками і навпаки. Найдоречнішим прикладом є електронні таблиці; де модифікація клітинки (події) ініціює наступну модифікацію всіх комірок, які її за нею спостерігали (використовували її значення для обчислення свого).

Це парадигма, яка означає, що більшість проблем, які можна вирішити за допомогою реактивного програмування, також можна вирішити іншими типами програмування; об’єктно-орієнтоване, процедурне, функціональне тощо. Найважливіше – розпізнати, який підхід є найбільш прийнятним, оскільки це рішення впливає на елегантність та якість отриманого рішення.

2.1 Застосування реактивного підходу

Реактивне програмування все частіше використовується в популярних компаніях, сервіси яких працюють під великим навантаженням. Більшість сучасних додатків є реактивними, тобто вони реагують на події, які до них надходять, і реагують на них відповідними обчисленнями. Таким чином користувачі покращують свої відносини з системою та краще взаємодіють з нею. Завдяки цьому відповідь на запит формується набагато швидше.

Реактивне програмування засноване на передачі даних з одного або кількох джерел. Всі без винятку дані обробляються через визначений потік операцій над ними. Потік представляє дані, які надходять упорядковано. Потік

випромінює три типи сигналів. Він видає такі значення як (власне) значення, помилку та сигнал сигнал, що вказує на відсутність подальшого надходження даних з потоку.

Компанії та організації впроваджують цей тип реактивності на всіх рівнях. Він особливо зустрічається в графічних інтерфейсах користувача. Сучасні додатки мають кілька незамінних якостей. Їх висока доступність дуже цінна для бізнесу: вони швидко реагують на будь-яку подію.

Реактивні програми також стають все більш стійкими в міру їх розробки. Система залишається доступною, навіть якщо виникає помилка. Навіть якщо вона перевантажена, система не руйнується. Саме тому реактивне програмування відповідає найсучаснішим вимогам.

2.2 Концепції реактивного програмування

Щоб зрозуміти, як працює реактивне програмування, необхідно знати основні поняття, до яких воно відноситься.

У реактивному програмуванні часто йдеться про зміну значень у часі. Під цією концепцією мається на увазі, що в будь-який момент виконання програми кожен вираз у програмі повинен залишатися коректним. Значення змінних змінюються з часом під час виконання програми.

Іншою важливою концепцією реактивного програмування є потік подій. Це концепція, згідно з якою, коли змінна змінює своє значення, змінні, які від неї залежать, також оновлюються. Після цього налаштовується подія/повідомлення. При цьому типі програмування повинні бути враховані всі події, які можуть відбутися.

Моніторинг залежностей також є частиною концепцій, що використовуються в реактивному програмуванні. Усі залежності між змінними, як прямі, так і

непрямі, необхідно відстежувати. Може з'явитися нова залежність і тоді її потрібно зберегти. Пізніше вона може поширювати оновлення значень.

Що стосується концепції автоматичного поширення змін, то це означає, що коли змінна оновлюється, всі інші залежні змінні інформуються, а також оновлюються. Поширення оновлень має продовжуватися автоматично, щоб таким чином оновлювалися всі змінні, незалежно від того, чи є вони прямо чи опосередковано залежними.

2.3 Переваги та недоліки реактивного програмування

Переваги застосування реактивного підходу до програмування:

Реактивне програмування все частіше використовується через його численні переваги. Зокрема, це допомагає уникнути несправностей, оскільки вони викликані проблемою дизайну, вони більше не можуть виникати, оскільки змінна не була оновлена. Час, який економиться під час виконання, є значним і дозволяє зосередитися на інших завданнях.

Більше того, з точки зору додатків, багато з них набагато легше писати в реактивному програмуванні. Тому робота спрощується. Програмісту також не доведеться турбуватися про порядок запуску подій.

- Підтримка потоків даних

  Як було зазначено раніше, мабуть, одна з найбільших переваг реактивного програмування полягає в тому, наскільки легко розробникам створювати потоки даних. Потоки можуть видавати значення, помилку та завершений сигнал. Ці події оброблюються асинхронно, і визначаються функції, які виконуються, коли видається значення, помилка або завершений сигнал. За допомогою реактивного програмування розробники можуть створювати, фільтрувати та комбінувати ці потоки для досягнення бажаних цілей.

- Високий рівень абстракції програм

  Оскільки реактивне програмування збільшує рівень абстракції розроблюваного коду, це дає змогу розробникам більше зосередитися на взаємозалежності подій, які визначають бізнес-логіку і їм більше не доведеться постійно мати справу з великою кількістю деталей реалізації. Переваги цього очевидні в сучасних мобільних і веб-додатках, які дуже інтерактивні з великою колекцією подій інтерфейсу користувача, пов’язаних із подіями даних. За допомогою реактивного програмування додатки можуть досягати більшого в режимі реального часу, покращуючи роботу і досвід користувача в цьому процесі.

Недоліки застосування реактивного підходу до програмування:

Проте програмісти відзначають, що в реактивному програмуванні все ще є можливості для вдосконалення. На даний момент це залишається важко реалізувати. Крім того, оскільки цей тип програмування є асинхронним, потоки може бути важко відслідковувати, і не завжди легко їх зрозуміти. Тому реактивне програмування потребує вирішення цих проблем у майбутньому.

- Високі вимоги до освоєння

  Реактивне програмування має репутацію такого підходу, який важко освоїти не маючи досвіду. Тому деяким розробникам може бути важко навчитися його застосовувати на перших етапах. З часом вони зможуть звикнути до ідеї реактивного програмування, але проходження початкового блоку може виявитися неприємним для багатьох людей. Той факт, що все має бути асинхронним потоком, ще більше посилює цю проблему. Розробникам, які не звикли працювати з цією парадигмою, буде важко вийти на ринок.
- Відсутність якісних освітніх ресурсів

Навіть ті, хто готовий випробувати це і вивчити реактивне програмування, будуть розчаровані відсутністю хороших освітніх ресурсів в цій сфері. Доступна інформація або занадто складна, або надто загальна, і велика її частина може не мати великого сенсу для тих, хто тільки починає.

2.4 Порівняння реактивного програмування з імперативним

Парадигми реактивного та імперативного програмування принципово відрізняються. Java є перш за все об’єктно-орієнтовною мовою. Реактивне програмування реєструє зворотний виклик, і його структура відповідає за правильний виклик цього зворотного виклику.

У імперативної парадигмі код виконується на основі операторів, які детально описують кожен крок, який програма має зробити, перш ніж завдання може бути завершено. В імперативному програмуванні дані вилучаються, тоді як вони переміщуються в реактивному.

В імперативній парадигмі для зміни стану програми використовуються оператори. У програмному коді перераховані команди, які комп’ютер повинен виконати для досягнення цілей завдання. Це визначає, як програма повинна працювати. Код виконується лише в порядку операторів за інструкціями.

Реактивне програмування відрізняється тим, що воно повністю покладається на асинхронні потоки даних. Ці потоки можна змінювати, створювати або комбінувати на льоту (під час виконання програми). Виконання здійснюється вже не у визначеному порядку, оскільки воно більше не базується на масивах, а на окремих потоках.

У більшості реалізацій необхідні як реактивна, так і імперативна парадигми. Написати бізнес-логіку легше в імперативному стилі, тоді для обробки подій краще підходить реактивна парадигма.

Таким чином, розробники можуть писати код програми в імперативному стилі, але використовувати реактивний стиль для виконання вибраних функцій.

Реактивний підхід до програмування допомагає розробникам підвищити продуктивність програми, оскільки цей підхід дозволяє розроблюваній системі набагато швидше обробляти великі обсяги даних. Основні властивості реактивного підходу:

- більша гнучкість
- стійкість і масштабованість
- більша пристосованість до помилок при обробці
- ефективність використання ресурсів
- менша затримка виконання при обробці

Код, написаний при реактивному підході є компактним і лаконічним, що полегшує читання та масштабування. Таким чином, зміни, модифікації та оновлення можна зробити спрощеним способом. Ці особливості надають цьому підходу додаткову перевагу – менші витрати часу на розробку.

Хоча розуміння концепції та вивчення реактивного підходу може зайняти деякий час і вимагає багато сумлінної роботи, навчання тощо, парадигма є надзвичайно корисною сьогодні. Вона має ряд переваг, які полегшують роботу розробників, покращує продуктивність додатків і, крім того, покращує досвід користувачів.

2.5 Висновки до розділу

У даному розділі було досліджено застосування реактивного підходу при програмуванні, описані основні концепції та особливості взаємодії компонентів, визначені переваги та недоліки застосування та проведена паралель між реактивним та імперативним підходами.

3 АНАЛІЗ НАЯВНИХ РІШЕНЬ РЕАКТИВНИХ СИСТЕМ НА МОВІ JAVA

3.1 Наявні рішення реактивних систем

Реактивні мікросервіси — це мікросервіси, написані з використанням моделі реактивного програмування.

Reactive Streams [6] — це ініціатива щодо забезпечення стандарту для обробки асинхронного потоку з неблокуючим зворотним тиском.

Обробка потоків даних, особливо «справжніх» даних, обсяг яких не визначений заздалегідь, вимагає особливої обережності та уваги в асинхронній системі. Найважливіша проблема полягає в тому, що споживання ресурсів потрібно ретельно контролювати, щоб швидке джерело даних не перевантажувало цільовий потік. Асинхронність потрібна для того, щоб уможливити паралельне використання обчислювальних ресурсів на мережевих хостах спільної роботи або кількох ядрах ЦП на одній машині.

Основна мета Reactive Streams — керувати обміном даними потоку через передачу елементів іншому потоку або пулу потоків — при цьому гарантувати, що сторона, яка їх отримує, не змушена буферизувати довільні обсяги даних. Іншими словами, зворотний тиск є невід'ємною частиною цієї моделі, щоб дозволити обмежувати черги, які є посередниками між потоками. Переваги асинхронної обробки були б зведені нанівець, якби сигнали зворотного тиску були синхронними, тому було взято до уваги повну неблокуючу та асинхронну поведінку всіх аспектів реалізації Reactive Streams.

Ця специфікація має на меті дозволити створити багато відповідних реалізацій, які завдяки дотриманню правил зможуть плавно взаємодіяти, зберігаючи переваги та характеристики в усьому циклі обробки.

Слід зазначити, що точна природа маніпуляцій з потоками (перетворення, розщеплення, злиття тощо) не охоплюється цією специфікацією. Reactive Streams займаються лише посередництвом потоку даних між різними компонентами API [6]. При їх розробці було зроблено увагу, щоб усі основні способи об'єднання потоків були виражені.

Підсумовуючи, Reactive Streams є стандартом і специфікацією для потоково-орієнтованих бібліотек для JVM, які:

- оброляють потенційно необмежене число елементів
- виконуються асинхронно
- обробляють елементи, передаючи їх між компонентами
- мають обов'язко неблокуючий протитиском

Reactive Streams ідеально підходять для опису потоку та перетворення даних через систему. Однак є деякі функції, надані Actors, які можуть бути досить корисними, і яких потоки (поки що) не надають. Саме ці функції знайшли корисними, надаючи перевагу Akka Actors [7] для його реактивної структури.

Першою перевагою акторів є гарантія паралельності, яку вони надають для спільного, змінюваного стану.

Кожен актор є об’єктом Java або Scala зі своєю власною поштовою скринькою. Фреймворк Akka гарантує, що рівно один потік витягне повідомлення з поштової скриньки актора і викличе в ньому функцію обробки повідомлення в будь-який момент часу. Завдяки цій гарантії можна помістити змінюваний стан всередині актора і більше ніколи не турбуватися, чи правильно була розроблена синхронізація. Якщо у потоковому API є спільний, змінюваний стан, доведеться шукати інструменти за межами потоків, які допоможуть зробити систему потокобезпечною.

Reactive Streams — це потужна техніка для вирішення випадків використання, з якими розробники стикаються в системах які реагують на події які

виникають. Це надає розробнику набагато більше гнучкості, ніж його аналоги і, що ще важливіше, дозволяє розробнику зосередитися на бізнес-логіці, а не витрачати продуктивність на роботу з основним механізмом виконання.

3.1.1 RxJava

RxJava [8] — це реалізація бібліотеки ReactiveX для віртуальної машини Java для імплементації асинхронних програм і програм на основі подій за допомогою спостережуваних послідовностей.

Основними блоками RxJava є Observables і Subscribers. Observable використовується для видачі елементів, а Subscriber — для споживання цих елементів.

Observables часто не починають видавати елементи, доки хтось на них не підписується. Це працює так: Subscriber підписується на Observable, потім Observable викликає Subscriber.onNext() для будь-якої кількості елементів, якщо щось піде не так, то викличеться Subsciber.onError() і якщо все закінчиться нормально, то викликається метод Subscriber.onCompleted().

RxJava є найбільш загальним з усіх реалізованих бібліотек, не має залежностей від інших бібліотек. Він дуже популярний на Android в основному через підтримку іншими бібліотеками/фреймворками. Версія 1.x не підтримує Reactive-Streams, але має обгортки для неї, тоді як 2.x сумісна з Reactive-Streams. Багато популярних мов програмування на основі JVM мають адаптери для цього (Scala, Kotlin, Clojure тощо).

3.1.2 Akka

Модель актора є основою для Akka. Актори - це незалежні набори коду, які спілкуються один з одним за допомогою повідомлень. Поштова скринька актора отримує повідомлення та ставить їх у чергу за допомогою одного потоку керування.

Набір інструментів спочатку був написаний на Scala, але сьогодні його також можна використовувати на Java. Akka є відкритим кодом і підтримує специфікацію Reactive Streams. Ідея набору інструментів полягала в тому, щоб представити високу доступність і масштабованість платформи Erlang для JVM.

Крім того, Akka виділяється ієрархією дерева акторів – особливою формою організації акторів, яка передбачає існування стосунків «батько-дитина» між акторами різних рівнів і таким чином забезпечує особливу стійкість до помилок і самовідновлення. Батько-актор піклується про дочірній блок: коли батько-актор отримує сповіщення про збій дитини, він може відновити, перезапустити, зупинити свою дочірню частину або зупинитися, щоб передати відповідальність за обробку помилок на наступний рівень. Оскільки актори представляють повністю ізольовані одиниці та не мають жодних змінюваних станів, такі раптові зупинки одного з них не вплинуть на інших акторів, і вони зможуть продовжувати роботу як зазвичай.

3.1.3 Vert.x

Vert.x [9] — це набір інструментів для створення розподілених реактивних систем над віртуальною машиною Java з використанням асинхронної та неблокуючої моделі розробки. Як набір інструментів, Vert.x можна використовувати в багатьох контекстах: в окремому додатку або вбудованому в програму Spring. Vert.x та його екосистема – це просто файли jar, які використовуються як і будь-яка інша бібліотека: необхідно лише розмістити їх у залежності свого проекту. Однак, оскільки Vert.x є набором інструментів, він не надає універсального рішення, а надає блоки для створення власного рішення. Багатство екосистеми робить Vert.x неймовірно гнучким – будь то прості мережеві утиліти, складні сучасні веб-додатки, мікросервіси

HTTP/REST, обробка подій великого обсягу чи повноцінний серверний додаток для шини повідомлень.

Eclipse Vert.x базується на моделі обробки циклу подій і не використовує моделі актора, як це робить Akka. Він набагато простіший, ніж Akka, але не має можливостей для обробки помилок. Деякі незалежні порівняння показують, що він швидший, ніж інші альтернативи. Власне, він використовується в деяких ігрових застосунках у режимі реального часу.

Крім того, він описується більше як набір інструментів, ніж як обмежувальна структура. Він доступний для таких мов як Java, JavaScript, Groovy, Ruby, Ceylon, Scala і Kotlin.

3.1.4 Project Reactor

Project Reactor [10] (або просто Reactor) - це бібліотека Reactive для створення неблокуючих додатків на JVM, заснована на специфікаціях Reactive Streams. Reactor – це основа реактивного стека в екосистемі Spring, і він розробляється в тісному співробітництві з Spring. WebFlux – веб-фреймворк з реактивним стеком Spring, використовує Reactor в якості базової залежності.

Project Reactor базується на реалізації фреймворку Spring. Він реалізує шаблони реактивного програмування і дотримується специфікації Reactive Streams. Через це легко помітити багато подібностей між Stream і Flux [11] (або його одноелементною версією – Mono [12]). Основні характеристики, які відрізняють Flux та Mono від Stream API, полягають у тому, що перші два дотримуються шаблону «Видавець-Підписник» і реалізують зворотний тиск.

Наприклад, якщо створюється об’єкт Flux, який бере елементи з бази даних, перетворює їх, застосовуючи передану операцію, і фільтрує їх за деякими критеріями, то нічого не відбувається. Усі ці операції виконуватимуться лише тоді, коли хтось підпишеться на Flux.

Дана бібліотека має найбільшу популярність серед всіх інших при розробці реактивних систем на мові програмування Java з використанням фреймворку Spring. Її популярність обумовлена високою підтримкою спільноти (даний проект є open-source), легким вбудовуванням у вже існуючу інфраструктуру Spring та розповсюдженим використанням у сторонніх бібліотеках.

3.2 Застосування Project Reactor у Spring

Розглянемо детальніше реалізацію Project Reactor. Spring включає Reactor у деякі зі своїх популярних модулів, таким чином дотримуючись шаблонів реактивного програмування, коли їх використовують розробники. Дотримуючись розумного підходу, вони не позбавляються від попереднього стилю програмування в більшості цих модулів. Це означає, що розробники які використовують ці бібліотеки мають змогу не використовувати підхід реактивного програмування.

Spring 5 має реактивну альтернативу стандартному підходу Spring Web MVC [13]: Spring WebFlux [14]. Він заснований на Project Reactor. В результаті це означає, що WebFlux дотримується стандартних концепцій реактивного програмування: видавців, підписників тощо і має на порядок більше переваг за стандартні блокуючі підходи при обробці запитів. Перша причина – повільне підключення до Інтернету. Повільне з’єднання дратує, але це стає ще гіршим із класичним блокуванням викликів HTTP. Скажімо, хтось шукає найкращі мобільні телефони у своєму улюбленому додатку для інтернет-магазину. Він вводить терміни запиту та натискає «Пошук». З’єднання користувача з Інтернетом повільне, тому йому потрібно почекати 15 секунд і дивитися на порожню сторінку. Потім з’явиться повна перша сторінка з результатами. Це відбувається тому, що веб-сервер не піклується про повільне з’єднання, тому він надсилає відповідь на 300 кілобайт повного JSON із підсумками, описами та цінами перших 50 елементів, які відповідають

вказаним критеріям. З боку користувача — на стороні клієнта — проблема пов’язана з класичними підходами веб-клієнтів, які використовують блокування викликів HTTP (навіть у різних потоках), які чекають, поки не буде отримана повна відповідь HTTP, щоб обробити дані, а потім відтворити сторінку. Кращою альтернативою було б, що сервер замість того, щоб повертати 50 елементів разом, надсилав клієнту елементи один за іншим, як тільки вони будуть знайдені. Потім клієнт міг адаптувати свій інтерфейс, щоб реагувати на ці окремі натискання елементів і відображати їх у міру їх появи. Необхідно зазначити, шо перемикання лише однієї з цих двох сторін зв’язку не виправить цю ситуацію: обидві мають використовувати інший підхід.

Щось подібне до проблеми повільного підключення може статися, якщо сервер зайнятий. Уявімо, що багато користувачів одночасно шукають товари в інтернет-магазині, а швидкодія бази до цього не готова. База даних стає повільнішою, і запити отримують 50 результатів за кілька секунд. Тепер вузьким місцем у взаємодії клієнт-сервер є не веб-інтерфейс, а підключення до бази даних. Проте результат той самий: щоб отримати список продуктів, користувач повинен дочекатися повного виконання запиту. Кращого результату можна було б досягти, якби підключення до бази даних замість того, щоб повертати результати запиту відразу (блокування), відкривало б потоки з клієнтами бази даних (інший серверний рівень) і повертало результати, коли воно їх знаходило.

З реактивним веб-інтерфейсом користувач бачить на екрані три елементи за секунду, а не всі відразу через п’ятнадцять секунд. Це перша перевага без блокування: швидша доступність даних, що в прикладі означає кращий досвід роботи з користувачем. Розробники повинні прагнути до цього і відмовитися від блокування викликів. Згідно з дослідженням Google, до 53% користувачів мобільних пристроїв залишають сайт, якщо він завантажується більше 3 секунд. [https://www.thinkwithgoogle.com/consumer-insights/consumer-trends/mobile-site-load-time-statistics/]

У сценарії повного реактивного стека база даних отримує результати, як тільки вони стають доступними. Те саме для веб-інтерфейсу (скажімо, HTTP). Рівень бізнес-логіки також повинен бути підготовлений до обробки елементів реактивним способом, напр. за допомогою реактивних бібліотек. Нарешті, щоб замкнути цикл і отримати переваги від повного реактивного стека, сторона клієнта повинна мати можливість обробляти дані, як тільки вони надходять, дотримуючись шаблону споживача.

Реактивні веб-інтерфейси мають додаткову перевагу з іншими асинхронними підходами: краще використання потоків. У системі, що не блокує, немає потоків, які чекають на завершення. Натомість сервер швидко обробляє клієнтський потік, щоб менше потоків сервера було зайнято. Веб-клієнти отримають сповіщення, коли нові дані стануть доступними за схемою "Публікація-Підписка". Ця перевага дуже актуальна, якщо є серверна програма, яка іноді може працювати повільно, таким чином накопичуючи багато заблокованих потоків і, зрештою, не в змозі обробити більше.

Одним з найпопулярніших модулів де Spring використовує Reactor є веб-фреймворк. Починаючи з Spring 5, розробникам надається можливість використовувати WebFlux, який поставляється з основними оновленнями, як новий спосіб оголошення ендпоінтів контролера та прозорою підтримкою подій, надісланих сервером, за допомогою Reactor API.

Spring Data також включає реактивні шаблони через свій модуль Reactive з включенням ReactiveCrudRepository. Це означає, що додаток може виконувати реактивні запити до таких баз даних, як MongoDB, Neo4J, Redis, Cassandra тощо, які вже мають реактивну версію свого драйвера. Потім драйвер бази даних надсилає почергово дані як потік, який контролюють підписники, замість того, щоб одночасно передавати всі результати запиту.

3.2.1 Об’єкти Flux та Mono

Об’єкт Flux

Flux<T> — це стандартний Publisher<T>, який представляє асинхронну послідовність з 0 до N елементів, що видаються. Може передавати виконання до наступного елемента, завершувати своє виконання сигналом або помилкою. Як і в специфікації Reactive Streams, ці три типи сигналів перетворюються на виклики методів onNext, onComplete і onError. Завдяки такому великому діапазону можливих сигналів Flux є реактивним типом загального призначення.

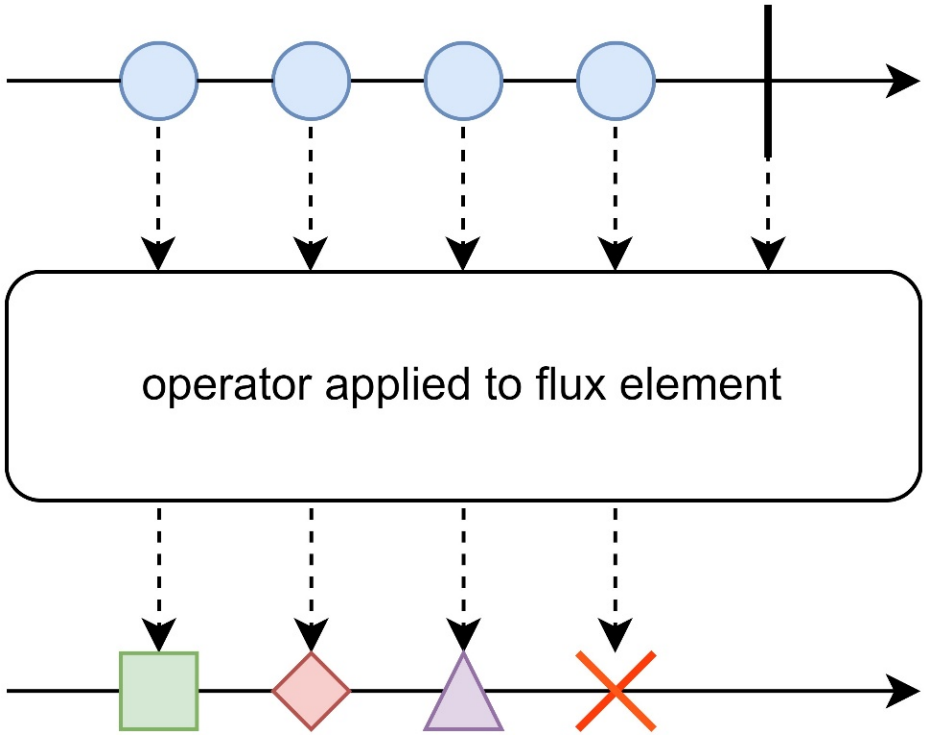

Рис. 3.1 – Взаємодія з об’єктом Flux

Дане зображення показує зміну елементів об’єкту Flux з плином часу. Вертикальна лінія означає подію про успішне завершення виконання об’єкту Flux. До кожного елемента (позначений синім кругом) може бути застосований оператор, який тим чи іншим чином застосовує визначені операції, тим самим змінюючи вхідні дані на інші (позначені квадратом, ромбом та трикутником. Елементи, які пройшли через визначений оператор

можуть бути вхідними для наступного, тим самим ці операції можуть повторюватись поки ланцюжок описаних дій не дійде до свого кінця. В процесі виконання може виникнути критична ситуація (exception – позначена червоним хрестом), відповідно якої згідно описаного блоку можна обробити цю ситуацію або взагалі негайно змусити Flux завершити свою обробку, тим самим передавши керування виконанням до стандартної гілки обробки помилок.

Об’єкт Mono

Mono<T> — це спеціалізований Publisher<T>, який видає щонайбільше один елемент через сигнал onNext, а потім завершується сигналом onComplete (успішне Mono, зі значенням або без нього) або видає лише один сигнал onError (невдалий сигнал Mono).

Очікується, що більшість реалізацій Mono негайно викличуть onComplete на своєму Subscriber після виклику onNext. Mono.never() є винятком: він не видає жодного сигналу, що технічно не заборонено, хоча й не дуже корисно за межами тестів в реальному додатку. Така комбінація як onNext і onError явно заборонена, бо одночасно Mono не може видати як успішний, так і помилковий елемент.

Mono пропонує лише підмножину операторів, доступних для Flux, а деякі оператори (зокрема ті, які поєднують Mono з іншим Publisher) можуть повертати Flux: наприклад, Mono#concatWith(Publisher) повертає Flux, а Mono#then(Mono) повертає інший Mono.

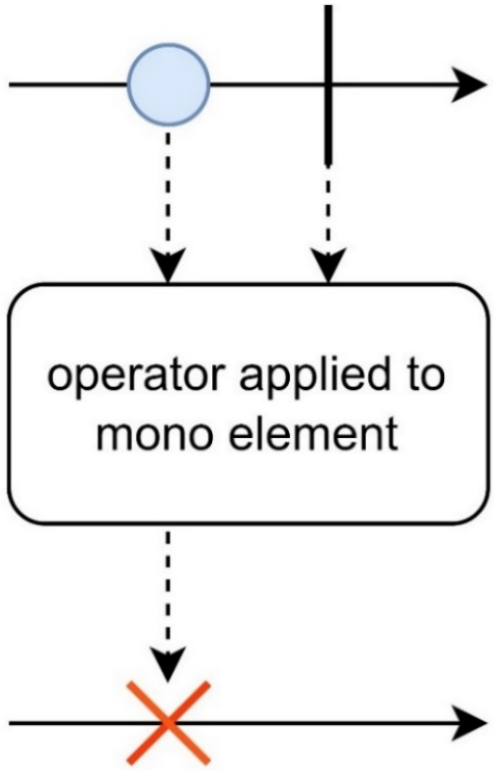

Рис. 3.2 – Взаємодія з об’єктом Mono

Дане зображення подібно до обробки елементів в потоці Flux може застосовувати оператори до єдиного вхідного елемента, або передавати управління та змінювати свою поведінку при помилці, яка може виникнути під час обробки елемента.

3.2.2 Особливості WebFlux

Spring WebFlux поставляється з додатковою новою функцією під назвою «Функції маршрутизації». Ідея полягає в тому, щоб застосувати функціональне програмування до веб-шару та позбутися декларативного способу реалізації контролерів і RequestMapping. Однак, Spring 5 все ще дозволяє використовувати анотації @Controller і @RequestMapping, щоб існувала варіативність вибору і залишалась звична всім реалізація.

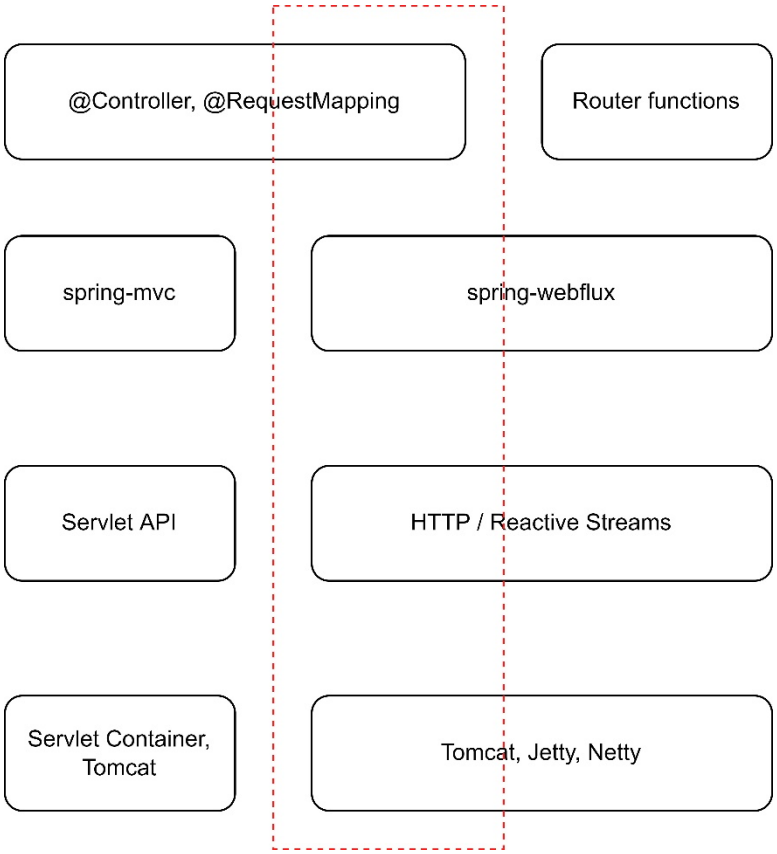

Рис. 3.3 – Стек технологій для Spring-сервісів

На відміну від моделі на основі анотацій, де фреймворк Spring MVC сканує шлях до класів для анотованих класів @Controller, а конфігурація шукає методи, анотовані @RequestMapping, функціональний підхід до побудови веб-фреймворку використовує HandlerFunction і RouterFunctions. RouterFunction є замінником анотації @RequestMapping, а HandlerFunction є реалізацією того, як буде оброблятися запит.

WebFlux створено з урахуванням неблокування при обробці запитів, тому він має модель паралельного програмування, відмінну від Spring MVC.

Spring MVC передбачає, що потоки будуть заблоковані, і використовує великий пул потоків, щоб продовжувати виконання програми під час блокування. Цей більший пул потоків робить MVC більш ресурсозалежним і накладає більше вимог на апаратну реалізацію, оскільки апаратне забезпечення комп’ютера має підтримувати більше потоків одночасно.

Натомість WebFlux використовує невеликий пул потоків, оскільки передбачає, що при обробці ніколи не доведеться пропускати виконання, щоб уникнути блокування. Ці потоки, які називаються обробниками циклу подій, мають фіксовану кількість і проходять через вхідні запити швидше, ніж потоки MVC. Це означає, що WebFlux використовує ресурси комп'ютера ефективніше, оскільки потоки завжди активно працюють і розподіляють обов'язки між собою.

3.2.2.1 Обробка запитів

Примітивний приклад з використанням стандартних контролерів:

```
@RestController
@RequestMapping
class MessageController {

    @GetMapping
    Flux<Message> allMessages(){
        return Flux.just(new Message("Example message from controller"));
    }

}
```

Зберігаючи ту ж функціональність та використовуючи Router Functions можна переписати на:

```
@Bean
public RouterFunction<ServerResponse> routes() {
    return route(
         GET("/"),
         (ServerRequest req) -> ok().body(
                BodyInserters.fromObject(
                     List.of(
                            new Message("Example message from controller")
                     )
                )
        );
}
```

в результаті чого контролери, які описані у Spring MVC-стилі:

```
@RestController
@RequiredArgsConstructor
@RequestMapping(value = "/posts")
class PostController {

    private final PostRepository posts;
```

```
    public PostController(PostRepository posts) {
        this.posts = posts;
    }

    @GetMapping(value = "")
    public Flux<Post> all() {
        return posts.findAll();
    }

    @GetMapping(value = "/{id}")
    public Mono<Post> get(@PathVariable(value = "id") UUID id) {
        return posts.findById(id);
    }

    @PostMapping(value = "")
    public Mono<ResponseEntity<?>> create(@RequestBody Post post) {
        return posts.save(post).map(p ->
ResponseEntity.created(URI.create("/posts/" + p.getId())).build());
    }

}
```

перетворюються на опис об’єкта маршрутизації, в якому встановлюється відповідність між ендпоінтом, який може обробляти запити, та методом, який викликається:

```
@Bean
public RouterFunction<ServerResponse> routes(PostHandler postController) {
    return route(GET("/posts"), postController::all)
        .andRoute(POST("/posts"), postController::create)
        .andRoute(GET("/posts/{id}"), postController::get)
        .andRoute(PUT("/posts/{id}"), postController::update)
        .andRoute(DELETE("/posts/{id}"), postController::delete);
}



@Component
@RequiredArgsConstructor
class PostHandler {

    private final PostRepository posts;

    public Mono<ServerResponse> all(ServerRequest req) {
        return ServerResponse.ok().body(posts.findAll(), Post.class);
    }

    public Mono<ServerResponse> create(ServerRequest req) {
        return req.bodyToMono(Post.class)
            .flatMap(post -> posts.save(post))
            .flatMap(p  ->  ServerResponse.created(URI.create("/posts/"  +
p.getId())).build());
    }

    public Mono<ServerResponse> get(ServerRequest req) {
        return posts.findById(req.pathVariable("id"))
```

```
            .flatMap(post   ->   ServerResponse.ok().body(Mono.just(post),
Post.class))
            .switchIfEmpty(ServerResponse.notFound().build());
    }

    public Mono<ServerResponse> update(ServerRequest req) {
        return Mono
            .zip(
                (data) -> {
                    Post p = (Post) data[0];
                    Post p2 = (Post) data[1];
                    p.setTitle(p2.getTitle());
                    p.setContent(p2.getContent());
                    return p;
                },
                posts.findById(req.pathVariable("id")),
                req.bodyToMono(Post.class)
            )
            .cast(Post.class)
            .flatMap(post -> posts.save(post))
            .flatMap(post -> ServerResponse.noContent().build());

    }

    public Mono<ServerResponse> delete(ServerRequest req) {
        return
ServerResponse.noContent().build(posts.deleteById(req.pathVariable("id")))
;
    }

}
```

В результаті – використовується менше анотацій, а це означає, що час на обробку зменшується.

Високорівнево обробка запиту полягає в наступному: коли HTTP-запит надходить на сервер, цей запит делегується RouterFunction для вирішення відповідної HandlerFunction для обробки HttpRequest, після чого формує відповідь на нього.

Завдяки функціональному підходу розробники більше стурбовані тим, як реалізувати свою бізнес-логіку, ніж на описі підтримуваних ендпоінтів та класів, в яких вони міститимуться.

3.2.2.2 Обробка помилок

Під час обробки запитів обов’язково виникатимуть помилки, які можуть відноситись до перевищення часу очікування запиту до сторонньої бази даних, до критичної ситуації в самому сервісі – виклик методу у null-об’єкта, або бізнес-логіки сервісу, які мають бути оброблені належним чином та містити детальний опис помилки, яка виникла, при цьому не призвести програму до краху.

У Java і помилки, і винятки є підкласами класу java.lang.Throwable. Помилка відноситься до забороненої операції, виконаної користувачем, що призводить до некоректної роботи програми.

Помилка «вказує" на серйозні проблеми, які розумна програма не повинна намагатися вловити» в той час як виняток «вказує на умови, які розумна програма може захотіти перехопити».

Помилка разом із RuntimeException та їх підкласами є неперевіреними винятками (ті омилки, для яких можна не описувати сценарії їхньої обробки якщо вони виникнуть). Усі інші класи винятків є перевіреними винятками.

Перевірені винятки – це, як правило, ті винятки, з яких програма може відновитися, і може бути гарною ідеєю відновити такі винятки програмним шляхом. Приклади включають FileNotFoundException, ParseException тощо. Очікується, що розробник буде оброблювати такі винятки за допомогою блоку try-catch або передасть їх тому, хто викликав цей метод.

З іншого боку, є неперевірені винятки. Це ті винятки, які можуть не статися, якщо все в порядку, але вони трапляються. Приклади включають ArrayIndexOutOfBoundException, ClassCastException тощо. Багато програм використовуватимуть речення try-catch або throws для RuntimeExceptions та їхніх підкласів, але з точки зору мови це робити не потрібно.

Помилки також є неперевіреним винятком, і програміст не зобов’язаний нічого робити з ними. Насправді, використовувати речення try-catch для

помилок – погана ідея. Найчастіше відновлення після помилки неможливе, і програмі слід дозволити завершити роботу. Приклади включають OutOfMemoryError, StackOverflowError тощо.

Хоча помилки є неперевіреними винятками, ми не повинні намагатися з ними боротися, але можна мати справу з RuntimeExceptions (також неперевіреними винятками) у коді. Перевірені винятки повинні оброблятися кодом.

Exception: виняток вказує на умови, які програма може захотіти перехопити, при цьому програма може відновити своє виконання.

Error: помилка вказує на серйозні проблеми, які програма не повинна намагатися виявити або відновити – для прикладу OutOfMemoryError.

Ті самі правила застосовуються і до світу реактивного програмування, але оскільки все є сигналом (незалежно від того, чи то успішна подія, чи помилка), розробники використовують Error, коли посилаються на сигнали помилки в реактивних потоках. Через це початківцю який опановує реактивний підхід до програмування з використанням Spring WebFlux варто це враховувати.

Сигнали про помилку – це такі сигнали, які перевіряються в першу чергу. Це означає, що це зупинить сформовану послідовність для виконання. Навіть якщо використовується оператор обробки помилок, він не дозволяє продовжити виконуватись для вхідної послідовності. Це дійсно важливий факт, який розробники повинні розуміти, використовуючи реактивні потоки.

Елементи Flux та Mono мають реалізації методів які дозволяють визначити остаточну поведінку сформованого потоку подій для виконання якщо в процесі виникає помилка. Це може бути логування або повернення стандартного значення, яке буде по визначеному стандарту відображати те, що при виконанні сталася помилка, або взагалі заміна поведінки виконання.

Загалом в об’єктах Mono/Flux реалізовано п’ять методів обробки помилок:

- onErrorReturn: повертає резервне значення для всього потоку (Mono/Flux). напр. якщо є потік з 10 елементів, і помилка сталася на елементі 3, то залишок з елементів 4,5,6… не буде виконаний, замість цього буде розглянуто резервне значення (заміняє значення на інше)
- onErrorResume: повертає резервне значення в термінах Mono/Flux для всього потоку (Mono/Flux). напр. якщо є потік з 10 елементів, і помилка сталася на елементі 3 то залишок з елементів 4,5,6… не буде виконаний, замість цього виконання перейде до переданого в цей метод параметра резервного значення (буквально – заміняє виконання на інше)
- onErrorContinue: перехоплює помилку лише того елемента, в якому вона сталася, після результаті чого виконання переходить до настпуного елемента з потоку. Наприклад, якщо є потік з 10 елементів, і помилка відбувається на елементі 3, то всі елементи (1-10), крім 3, завершаться нормально, а результат виконання для елемента 3 буде залежати від споживача, який був переданий в onErrorContinue
- doOnError: перехоплює помилку та зупиняє виконання для подальших елементів у потоці
- onErrorMap: перетворює одну помилку в іншу та зупиняє виконання для подальших елементів у потоці

```
public Mono<ServerResponse> handleRequest(ServerRequest request) {
    return handleRequestInternal(request)
      .flatMap(s -> ServerResponse.ok()
          .contentType(MediaType.TEXT_PLAIN)
          .bodyValue(s)
      )
      .onErrorResume(e -> Mono.just("Error " + e.getMessage())
          .flatMap(s -> ServerResponse.ok().contentType(APPLICATION_JSON)
            .bodyValue(s))
      );
}
```

Такий імперативний підхід до визначення поведінки при помилці в сформованій послідовності дій спонукає до написання великої кількості повторюваного коду, який в перспективі виявиться важко підтримуваним через велику кількість подібних появ в коді – для кожного потоку даних, щоб

програма не завершила своє виконання коли цього не треба, необхідно буде описувати реакцію на помилки, які сталися.

Саме для того, щоб вирішити проблеми такого підходу, існує механізм, який дозволяє в загальній області для всього сервісу описати поведінку при будь-яких помилках, які можуть статися (за виключенням тих, які відносяться до операційної системи або зникнення живлення електроенергії). Зазвичай це відноситься до веб-застосунків, які надають API і обробляють запити згідно визначеного контракту для запитів та відповідей на них, бо в більшості випадків якщо в процесі стається помилка, то виконання має негайно завершитися і клієнт має отримати повідомлення про помилку.

Маючи необхідність продовжувати виконання при такому підході у випадку виникнення помилки, легко буде додати в потрібні місця додати сценарії, які будуть викликатися в критичних ситуаціях. Тим самим проблема повторюваності і важкості в підтриманні коду зникає, а місця де треба визначена поведінка все одно залишаються.

```
@Component
@RequiredArgsConstructor
public class GlobalExceptionHandler implements ErrorWebExceptionHandler {

  private final DataBufferWriter bufferWriter;

  @Override
  public Mono<Void> handle(ServerWebExchange exchange, Throwable ex) {
    HttpStatus httpStatus;
    ErrorResponse errorResponse;

    if (ex instanceof MethodArgumentNotvalidException) {
      httpStatus = BAD_REQUEST;
      errorResponse = buildErrorResponse(INVALID_PARAMS, ex.getMessage());
    } else if () {
        // ...
    } else {
       httpStatus = INTERNAL_SERVER_ERROR;
       errorResponse = buildErrorResponse(GENERAL, ex.getMessage());

    }


    exchange.getResponse().setStatusCode(status);
    return bufferWriter.write(exchange.getResponse(), errorResponse);
  }
}
```

```
@Component
@RequiredArgsConstructor
public class DataBufferWriter {

    private final ObjectMapper objectMapper;

    public <T> Mono<Void> write(ServerHttpResponse httpResponse, T object){
        return httpResponse
            .writeWith(Mono.fromSupplier(() -> {
                DataBufferFactory bufferFactory = 
httpResponse.bufferFactory();
                try {
                    return 
bufferFactory.wrap(objectMapper.writeValueAsBytes(object));
                } catch (Exception ex) {
                    log.warn("Error writing response", ex);
                    return bufferFactory.wrap(new byte[0]);
                }
            }));
    }
}
```

Більш того, починаючи зі Spring Framework 5.2, сервіс, побудований на бібліотеці Spring WebFlux, має змогу використовувати такий самий підхід до обробки помилок, як реалізація у Spring MVC, де декларативно вказується для помилок яких контролерів можуть бути викликані методи даного обробника. Якщо не вказувати цього, даний обробник помилок вважатиметься глобальним для всіх точок в описаному сервісі (за винятком самого себе – якщо помилка станеться під час обробки помилки, виконання передається до глобального стандартного обробника помилок, який міститься в реалізації бібліотеки).

```
@RestControllerAdvice
public class WebErrorHandler {

  @ExceptionHandler(Exception.class)
  @ResponseStatus(INTERNAL_SERVER_ERROR)
  public ErrorResponse handleGeneralException(Exception ex) {
    // log information about exception
    // return appropriate response body
  }

}
```

Даний опис обробника помилок визначає поведінку сервісу при будь-якій помилці, яка сталась в процесі виконання. Цей підхід підтримує ієрархію помилок при обробці: визначення яким методом необхідно обробити помилку

буде відбуватися в такому ж порядку, як і опис методів класу-оброблювача. Це дає змогу диверсифікувати поведінку при обробці різних помилок, тим самим додаючи більшу кількість інформації, яка може знадобитися при визначенні причини виникнення цієї помилки:

```
@RestControllerAdvice
public class WebErrorHandler {

  @ExceptionHandler(MethodArgumentNotValidException.class)
  @ResponseStatus(BAD_REQUEST)
  public ErrorResponse
handleArgumentException(MethodArgumentNotValidException ex) {
    String errorMessage =
getValidationErrorMessage(ex.getBindingResult());

    log.error("Handling validation exception: [{}]", errorMessage);
    return buildErrorResponse(INVALID_PARAMS, errorMessage);
  }

  @ExceptionHandler(Exception.class)
  @ResponseStatus(INTERNAL_SERVER_ERROR)
  public ErrorResponse handleGeneralException(Exception ex) {
    log.error("Handling general exception: [{}]", ex.getMessage());
    return buildErrorResponse(GENERAL, ex.getMessage());
  }

}
```

В даному прикладі формується повідомлення про те, які дані з запиту, який прийшов, мали некоректне значення, після чого воно відправляється клієнтові, який має змогу прочитати повідомлення про помилку та виправити свій запит.

3.2.2.3 Взаємодія з базами даних

Разом зі Spring Boot 2 був представлений Spring WebFlux, який дозволяв створювати реактивні веб-сервіси. Це було чудово, і це добре працювало з базами даних NOSQL, але коли справа дійшла до реляційних баз даних, це стало проблемою. Під капотом для звернень до реляційної бази даних Java використовує стандарт API JDBC [17].

Підключення до бази даних Java (JDBC) — це специфікація стандартного інтерфейсу прикладного програмування (API), що дозволяє програмам Java

отримувати доступ до систем керування базами даних. JDBC API складається з набору інтерфейсів і класів, написаних мовою програмування Java.

Використовуючи ці стандартні інтерфейси та класи, програмісти можуть писати програми, які підключаються до баз даних, надсилати запити, написані мовою структурованих запитів (SQL), та обробляти результати.

Оскільки JDBC є стандартною специфікацією, одна програма Java, яка використовує JDBC API, може підключитися до будь-якої системи управління базами даних (СУБД), якщо для цієї конкретної СУБД існує драйвер.

Такий підхід застосовується до всіх популярних реляційних баз даних: Postgres, MySQL, SQL Server, OracleDB, H2.

Операції з базою даних JDBC блокуються за своєю природою, і це обмежує розробника у створенні програми, яка повністю має підтримувати неблокуючі виклики та реактивність. Але для того, щоб мати асинхронну та неблокуючу програму, необхідно застосувати реактивний підхід до кожного шару програми. Рішенням, яке допомагає при написанні реактивних систем, став драйвер R2DBC – Reactive Relational Database Connectivity [18], що дає можливість здійснювати неблокуючі виклики до реляційних баз даних.

Через високу популярність реактивного підходу до програмування систем, для Project Reactor реалізовані імплементації часто вживаних сховищ даних та брокерів повідомлень: MongoDB, Redis, Neo4J, Cassandra, Couchbase, Kafka, RabbitMQ, інтерфейси репозиторіїв яких мають стандартне іменування ReactiveMongoRepository, ReactiveRedisRepository, ReactiveNeo4jRepository, ReactiveCassandraRepository і т. д. відповідно.

3.3 Порівняння бібліотек

Бібліотека	Найкраще застосування	Переваги	Недоілки
RxJava	Android-додатки	Немає залежностей від інших бібліотек	Частково підтримує Reactive Streams підхід
Project Reactor	Spring, малий час відклику	Повністю підтримує Reactive Streams підхід	Сильно залежить від інших бібліотек
Akka	Серверні застосунки	Має сценарії відновлення після невдачі виконання	Використовує власний формат обміну повідомлень
Vert.x	Мікросервіси	Перший по перформансу	Немає змоги відновлюватись після краху

Табл. 3.1 – Порівняння реактивних бібліотек

Як можна побачити, кожна з реалізацій має як свої переваги, так і недоліки. Саме тому вибір конкретного фреймворку покладається на архітектора інформаційної системи і залежить від доменної області розроблюваного додатку, бо він є одним з тих, хто може оцінити і визначити ступінь корисності використання того чи іншого фреймворку при розробці.

3.4 Висновки до розділу

У даному розділі були розглянуті наявні рішення реактивних систем на мові Java, які підтримують стандартну специфікацію в своїй реалізації, описані їхні особливості, проведено порівняння, визначені переваги та недоліки кожної з бібліотек. Було досліджено застосування бібліотеки Project Reactor у фреймворку Spring, були розглянуті особливості реалізації Spring WebFlux та його компонентів, наведені приклади реалізацій та підходів, були розглянуті особливості взаємодії з базами даних при побудові реактивного сервісу.

4 ЗАСТОСУВАННЯ РЕАКТИВНИХ СИСТЕМ В МІКРОСЕРВІСНІЙ АРХІТЕКТУРІ

Мікросервісна архітектура, або просто мікросервіси, — це особливий метод розробки систем, який намагається сфокусуватись на створенні модулів, які мають єдине призначення з чітко визначеними інтерфейсами та операціями. Ця тенденція стала популярною в останньому десятилітті, оскільки компанії прагнуть стати більш Agile [19] і рухатися до DevOps [20] і безперервного тестування.

Мікросервіси вирішують проблеми монолітних систем, будучи максимально розподіленими на модулі. У найпростішій формі вони допомагають розділити програму на набір невеликих служб, кожна з яких виконується окремо та розгортається незалежно від інших. Ці служби можуть використовувати різні мови програмування, методи зберігання даних та різний стандарт контрактів API. Це призводить до імплементації систем, які є масштабованими та гнучкими, але все одно воно потребує постійного динамічного оновлення. Мікросервіси часто підключаються через API і можуть використовувати багато інструментів і рішень, які були розроблені в екосистемі RESTful і веб-сервісів.

Там, де присутній великий обсяг, велика різноманітність, велика швидкість надходження даних, де необхідна одночасна обробка запитів та де обчислення потрібно розбивати на кроки, мікросервісів недостатньо. Ці мікросервіси також повинні бути реактивними. Реактивна архітектура мікросервісів є специфічним типом архітектури. Властивість реактивності дає змогу сервісу бути гнучким (обчислювальні ресурси можуть збільшуватися і зменшуватися в хмарному сервісі), стійким (якщо вузол виходить з ладу, він може самовідновлюватися) і мати швидкий відклик (висока доступність / низька затримка). Основною особливістю реактивних мікросервісів є асинхронна передача повідомлень між сервісами.

4.1 Особливості реактивних мікросервісів

Мікросервіси - це стиль архітектури, який структурує додаток як набір окремих сервісів, які

- зручно підтримуються і тестуються
- слабо зв’язані між собою
- мають змогу самостійно розгортатися
- побудовані навколо можливостей та вимог бізнесу

Мікросервісна архітектура гарантує швидке, часте, постійне та надійне постачання великих складних сервісів, які «під капотом» розподілені на невеликі окремі служби. [21]

4.1.1 Основні характеристики

Так само, як немає формального визначення терміна мікросервіси, не існує стандартної моделі, яку можна побачити в кожній системі, заснованій на цьому архітектурному стилі. Але можна очікувати, що більшість мікросервісних систем будуть мати кілька помітних характеристик. Серед них можна виділити шість основних [22]:

- Розподіленість на компоненти

  Додатки, створені як мікросервіси, за визначенням можуть бути розділені на кілька компонентів для того, щоб кожен з них можна було розгорнути, налаштувати, а потім повторно розгорнути не впливаючи на інші і без шкоди для цілісності програми. Для модифікування знадобиться змінити лише одну або кілька окремих служб замість того, щоб повторно розгортати всю єдину систему.

- Відповідність вимогам бізнесу

  Мікросервіси зазвичай розгортаються навколо бізнес-можливостей і цілей компанії. Відрізняючись від традиційного монолітного підходу до розробки, коли різні команди фокусуються на інтерфейсах користувача, базах даних, технологічних рівнях або логіці на стороні сервера, в архітектурі мікросервісів використовуються міжфункціональні команди. Кожна команда відповідає за створення конкретних продуктів на основі однієї або кількох окремих служб, що спілкуються між собою за визначеними контрактами.

- Проста маршрутизація

  Мікросервіси працюють подібно до класичної системи UNIX: вони можуть отримувати запити, обробляти їх і відповідно формувати відповідь. Це відрізняється від того, як працюють інші продукти, де використовуються високотехнологічні системи для маршрутизації повідомлень, оркестрації та застосування бізнес-правил. Можна сказати, що мікросервіси мають ендпоінти, які обробляють дані та містять в собі бізнес-логіку, і прості канали, через які протікає інформація.

- Децентралізована система

  Так як мікросервіси побудовані з використанням різноманітних технологій, старі підходи централізованого керування не є оптимальними. При такому підході перевага надається децентралізованому управлінню, оскільки його розробники прагнуть створити корисні інструменти, які потім можуть бути використані іншими. Подібно до децентралізованого керування, архітектура мікросервісів також сприяє децентралізованому керуванню даними. При стандартній побудові монолітної системи використовується єдина логічна база даних для різних додатків. На відміну від цього, мікросервіс зазвичай керується своїм власним станом і має окреме підключення до власної бази даних.

- Стійкість до збоїв

  Так як в одному середовищі декілька унікальних і різноманітних служб взаємодіють разом, цілком можливо, що щось може вийти з ладу. В таких ситуаціях сервіси, які не відносяться до даного все одно мають продовжувати працювати. Додатково для запобігання ризиків збою може допомогти моніторинг мікросервісів. Зі зрозумілих причин ця можливість додає більше складності мікросервісній архітектурі порівняно з монолітними системами, де моніторинг застосовується тільки для одного «сервісу».

- Еволюційність

  Архітектура мікросервісів є еволюційним дизайном і вона ідеально підходить для тих систем які мають властивість часто змінюватись з часом. Існує багато прикладів коли додатки починали розроблятись на основі монолітної архітектури, але коли з’являлась якась кількість непередбачених вимог, вони повільно застосовували мікросервіси, які спілкувались із перехідним монолітом через надане API.

4.1.2 Використання брокерів повідомлень

Брокер повідомлень – це програма-посередник, яку сервіси використовують для зв’язку один з одним для обміну інформацією або поширення подій. Брокери повідомлень використовуються для перевірки, зберігання, маршрутизації та доставки повідомлень до вказаних місць призначення. Передавати інформацію можуть не тільки програми, навіть якщо вони реалізовані різними мовами програмування. Оскільки брокери повідомлень діють як посередницький сервіс, відправник не має уявлення про кількість одержувачів, якщо вони присутні в мережі.

Найважливіше те, що брокери гарантують, що одержувачі отримають повідомлення, навіть якщо вони не в мережі або не активні. Це схоже на

електронну пошту. Не потрібно бути онлайн щоб отримати повідомлення. Брокери повідомлень використовують для цього чергу повідомлень, і вона зберігається в пам’яті або на жорсткому диску запущеної машини. Вони використовуються для зберігання та доставки повідомлень.

Черга повідомлень зберігає повідомлення в точному порядку їх отримання та надсилає одержувачу в такому ж порядку. Якщо якось не вдалося доставити повідомлення (проблема з мережею або збій сервісу), повідомлення буде перенесено в кінець черги (в залежності від реалізації брокера).

Завдяки чергам повідомлень програми можуть працювати асинхронно, це запобігає втраті даних і дає можливість системам продовжувати функціонувати, якщо процеси або з’єднання виходять з ладу. Це дозволяє розробникам розділяти процеси та програми.

Основні компоненти брокера повідомлень:

- Виробник – цей компонент відповідає за надсилання повідомлень. Він підключений до брокера повідомлень. У шаблоні публікації/підписки вони називаються видавцями.
- Споживач — цей компонент споживає повідомлення в брокері повідомлень. У шаблоні опублікувати/підписатись вони називаються споживачами.
- Черга/тема — посередник повідомлень де зберігаються власне повідомлення.

Одним із способів створення реактивних мікросервісів є використання посередника повідомлень, наприклад Kafka або RabbitMQ для надсилання повідомлень між мікросервісами. Це дозволяє запускати кожен мікросервіс повністю незалежно від інших, а також динамічно додавати та видаляти їх як споживачів черги/теми повідомлень.

Використовуючи широкомасштабований брокер повідомлень, такий як Kafka, можна створювати великі кластери мікросервісів з балансуванням навантаження через групи споживачів (або повідомлень). Ці групи дозволяють лише одному споживачу в групі споживачів читати дане повідомлення.

Основною перевагою використання брокера повідомлень є гарантії доставки повідомлень. Ніхто і ніщо не дає гарантій стосовно того, що будь-яке повідомлення буде успішно доставлено. Якщо, наприклад, обробляються конфіденційні дані, такі як фінансові транзакції, їхня втрата при обробці нікого не влаштує, і в цьому випадку перевага надається підходу мікросервісів, керованих повідомленнями.

Недоліком використання брокера повідомлень є відносно велика затримка для відправки повідомлення від виробника до споживача. З огляду на це, за умови належної мережі брокер повідомлень може бути налаштований на зменшення цих затримок, коли це необхідно, і можна очікувати розумних затримок (менше 10 мс).

4.2 Переваги та недоліки мікросервісної архітектури

Мікросервіси — це не те рішення, яке після своєї реалізації підійде до будь-яких вимог. Навпаки, побудувавши мікросервісну архітектуру, додатково можна знайти проблеми в роботі команди та інші, які раніше могли бути непоміченими. API в мікросервісах можуть значно скоротити час і зусилля на збірку та контроль якості.

Однією з поширених проблем є спільне використання схеми/логіки перевірки між службами. Те, що потрібно мікросервісу A, щоб вважати деякі дані дійсними, не завжди стосується мікросервісу B, мікросервіс B може мати інші вимоги. Найкращою рекомендацією є застосування версій і розповсюдження схеми в спільних бібліотеках. Зміни в бібліотеках можуть обговорюватись між командами. Крім того, із сильним керуванням версіями виникають залежності,

що може призвести до додаткових витрат. Найкращою практикою для подолання цього є планування архітектури щодо зворотної сумісності та прийняття регресійних тестів від зовнішніх служб/команд.

Як і в будь-якому іншому, чи підходить архітектура мікросервісів для вирішення задачі, залежить від вимог, тому що всі вони мають свої плюси і мінуси.

Переваги застосування мікросервісної архітектури:

- Мікросервісна архітектура дозволяє розробникам самостійно розробляти та розгортати служби
- Мікросервіс може бути імплементований досить невеликою кількістю розробників
- Кожен мікросервіс може бути написаний використовуючи різні мови
- Проста інтеграція та автоматичне розгортання (за допомогою інструментів безперервної інтеграції з відкритим кодом, таких як Jenkins, GitLab CI тощо)
- Розробникам легко зрозуміти та змінити реалізацію, таким чином це може допомогти новому члену команди швидко зрозуміти призначення сервісу
- Код організовано навколо можливостей бізнесу
- Відсутність необхідності змінювати та повторно розгортати всю систему – коли потрібні зміни в певній частині програми, необхідно змінити та повторно розгорнути лише пов’язану службу
- Краща ізоляція несправностей: якщо один мікросервіс виходить з ладу, інший продовжуватиме працювати (хоча одна проблемна область додатку може поставити під загрозу функціонування всієї системи)
- Легка масштабованість та інтеграція зі сторонніми службами
- Відсутність довгострокової прив’язаності до стека технологій

Недоліки застосування мікросервісної архітектури:

- Через розподілене розгортання тестування може стати складним і виснажливим
- Архітектура створює додаткову складність, оскільки розробникам доводиться зменшувати відмовостійкість, затримку мережі та працювати з різними форматами повідомлень, а також з балансуванням навантаження
- Будучи розподіленою системою, це може призвести до дублювання зусиль і задіювати зайві людські ресурси
- Інтеграція та управління цілими продуктами можуть ускладнитися коли збільшується кількість послуг, які сервіси надають
- Крім кількох складнощів монолітної архітектури, розробникам доводиться мати справу з додатковою складністю розподіленої системи.
- Розробникам доводиться докласти додаткових зусиль для впровадження механізму зв'язку між сервісами
- Обробляти випадки використання, які охоплюють більше однієї служби, не тільки складно, але й ще це вимагає спілкування та співпраці між різними командами

4.3 Приклад застосування Spring в реактивному сервісі

Розроблений сервіс містить в собі два з трьох рівнів архітектури – бізнес-прошарок та прошарок даних, де реалізований доступ до бази даних. 3-рівнева архітектура програми — це модульна архітектура клієнт-сервер, яка складається з рівня презентації, рівня програми та рівня даних. Рівень даних зберігає інформацію, рівень програми обробляє логіку, а рівень презентації — це графічний інтерфейс користувача (GUI), який взаємодіє з двома іншими

рівнями. Ці три рівні є логічними, а не фізичними, і можуть виконуватися на одному фізичному сервері, а можуть і не працювати.

В якості сховища даних була обрана база даних MongoDB. Найновіші релізи MongoDB підтримують специфікацію Reactive Streams.

4.3.1 Структура проекту

При реалізації сервісу був застосований стандартний підхід до побудови архітектури та взаємодії компонентів в ній.

Пліч-о-пліч в сервісі присутні стандартна реалізація Spring MVC та Spring WebFlux для того, щоб показати особливості і відмінності реалізацій та провести тестування і порівняти швидкодію розроблених підходів.

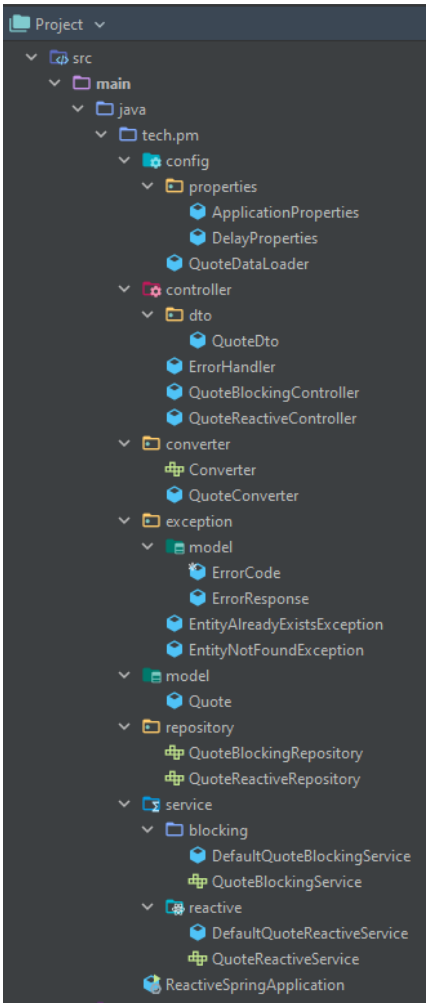

Рис. 4.1 – Структура проекту

Клас Quote являє собою модель, яка зберігається у базі даних. Для тестування будуть використовуватись запити на отримання списків Quotes за допомогою наявних ендпоінтів, які обробляють запити в залежності від викликаного методу використовуючи блокуючу або реактивну реалізацію сервісів.

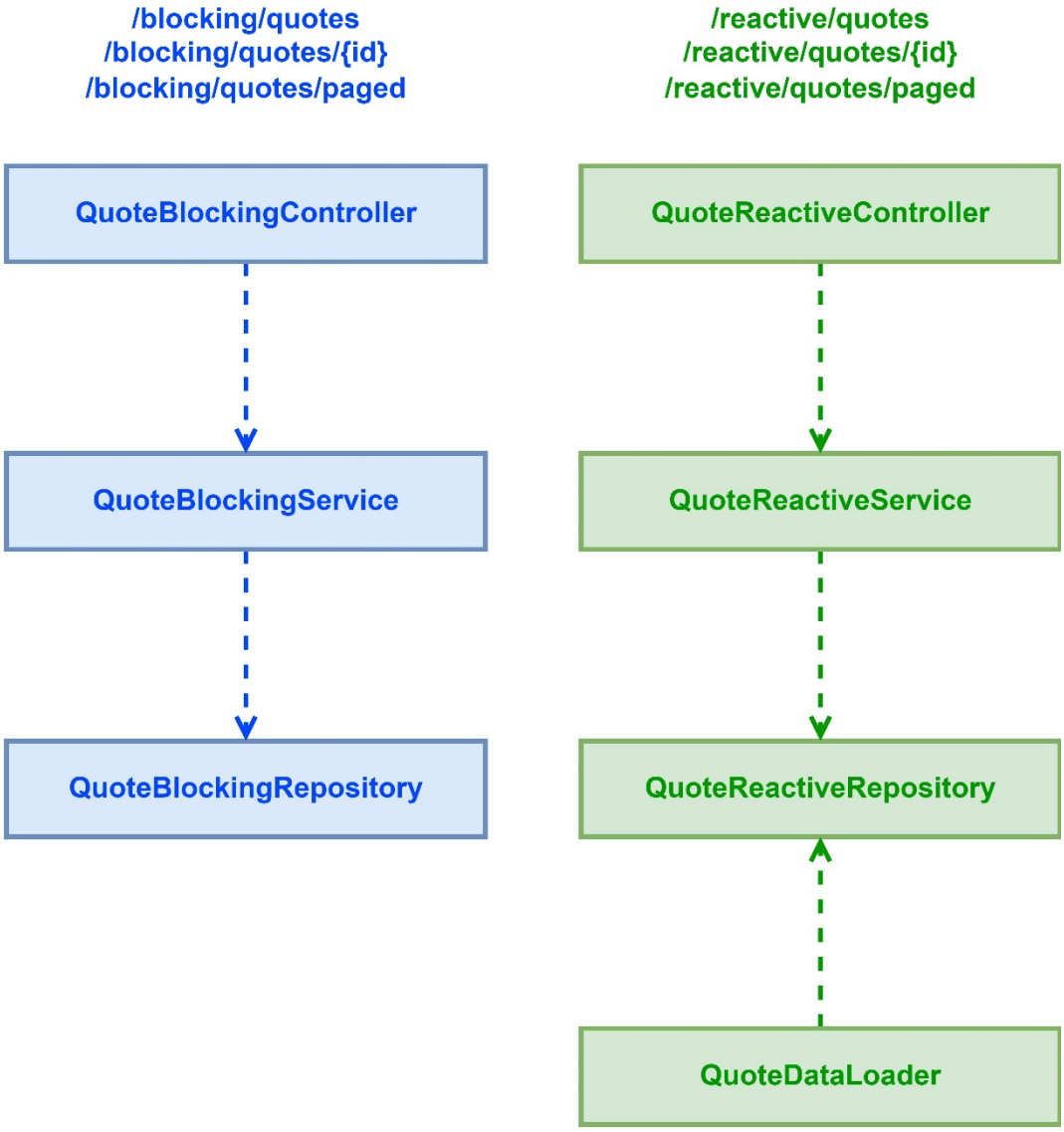

Рис. 4.2 – UML-діаграма описаних сервісів

Блокуючі та реактивні виклики повністю розділені одне від одного та ніяк не використовують інший функціонал окрім описаного спеціально для них.

4.3.2 Розробка проекту

Джерело даних реактивного потоку: MongoDB [23]. Описаний клас Quote представляє доменний об’єкт, який зберігається в базі даних та при виконанні запитів співставляється з об’єктами Java.

Опис класу Quote:

```
@Data
@Document
@NoArgsConstructor
@AllArgsConstructor
public class Quote {

  private String id;
  private String book;
  private String content;

}
```

Описані репозиторії для обох підходів:

Блокуючий:

```
@Repository
public interface QuoteBlockingRepository extends MongoRepository<Quote,
String> {

  List<Quote> findAllByIdNotNullOrderByIdAsc(Pageable page);

}
```

Реактивний:

```
@Repository
public interface QuoteReactiveRepository extends
ReactiveMongoRepository<Quote, String> {

  Flux<Quote> findAllByIdNotNullOrderByIdAsc(Pageable page);

}
```

Інтерфейси, які ми розширюємо, MongoRepository<T, ID> та ReactiveMongoRepository<T, ID>, мають можливості сортування окрім базового ReactiveCrudRepository<T, ID>, який містить основні операції CRUD. Ми отримуємо все, що нам потрібно, крім однієї вимоги: отримання сторінок. Для цього ми використовуємо методи запиту Spring Data і передаємо аргумент Pageable, щоб визначити зміщення та результати на сторінку. Зауважте, що

запит відповідатиме всім лапкам, оскільки наш фільтр шукає ненульові ідентифікатори, тому він там лише тому, що метод запиту findAllBy... завжди очікує фільтр. Ми також хочемо відсортувати результати за ідентифікатором цитати, тому ми додаємо суфікс OrderByIdAsc, а Spring Data подбає про переведення цього у правильне положення сортування MongoDB.

Наявні методи findAll, і findAllByIdNotNullOrderByIdAsc, які використовуються в імплементації сервісу, повертають Flux. Це означає, що він може контролювати, як швидко витягувати дані з бази даних.

Одне, що привертає увагу людей, не знайомих з реактивними шаблонами, це те, як працює метод збереження (і його варіанти). Можна порівняти сигнатури CrudRepository і ReactiveCrudRepository:

```
public interface CrudRepository<T, ID> extends Repository<T, ID> {

   <S extends T> S save(S entity);

   ...
}

public interface ReactiveCrudRepository<T, ID> extends Repository<T, ID> {

   <S extends T> Mono<S> save(S entity);

   ...
}
```

Для того, щоб зберегти новий запис в базі даних, при блокуючому підході слід викликати метод:

```
quoteRepository.save(quote);
```

Це буде працювати з тими інтерфейсами репозиторіїв, які розширюють CrudRepository. Однак, якщо цей метод буде викликаний для ReactiveCrudRepository, сутність не буде збережена. Реактивний репозиторій повертає Mono, який є Publisher, тому він не почне працювати, поки хтось не підпишеться на нього. Якщо є потреба імітувати поведінку блокування, яку пропонує CrudRepository, замість цього потрібно викликати:

```
quoteRepository.save(quote).block();
```

Такий підхід до взаємодії з реактивними об’єктами буде вести себе як класичне визначення репозиторію, при цьому втрачається вся суть використання реактивності.

Блокуючий та реактивний контролери дуже схожі між собою та мають набір методів, які підтримують CRUD операції.

Блокуючий контролер:

```
@RestController
@RequestMapping("/blocking/quotes")
public class QuoteBlockingController {

  private final QuoteBlockingService quoteService;
  private final QuoteConverter quoteConverter;

  @GetMapping
  public List<QuoteDto> getAll() {
    log.info("Retrieving all quotes");
    List<QuoteDto> quoteDtos = quoteService.getAll()
        .stream()
        .map(quoteConverter::toDto)
        .toList();
    log.info("Successfully retrieved all quotes");
    return quoteDtos;
  }

  @GetMapping("/paged")
  public List<QuoteDto> getAll(@RequestParam("page") int page,
                               @RequestParam("size") int size) {
    log.info("Retrieving all quotes by page [{}] and size [{}]", page,
size);
    List<QuoteDto> quoteDtos = quoteService.getAllByPage(page, size)
        .stream()
        .map(quoteConverter::toDto)
        .toList();
    log.info("Successfully retrieved by page [{}] and size [{}]", page,
size);
    return quoteDtos;
  }

  @GetMapping("/{id}")
  public QuoteDto getById(@PathVariable String id) {
    log.info("Retrieving quote by id [{}]", id);
    QuoteDto retrieved = quoteConverter.toDto(quoteService.get(id));
    log.info("Successfully retrieved quote [{}]", retrieved);
    return retrieved;
  }

  @PostMapping
  @ResponseStatus(CREATED)
  public   QuoteDto   create(@Validated({Default.class,   OnCreate.class})
@RequestBody QuoteDto quoteDto) {
    log.info("Creating quote [{}]", quoteDto);
    QuoteDto created = quoteConverter.toDto(
        quoteService.create(quoteConverter.fromDto(quoteDto)));
```

```
    log.info("Successfully created quote [{}]", created);
    return created;
  }

  @PutMapping("/{id}")
  public QuoteDto update(@PathVariable String id,
                         @Validated @RequestBody QuoteDto quoteDto) {
    log.info("Updating quote by id [{}], [{}]", id, quoteDto);
    QuoteDto updated = quoteConverter.toDto(
        quoteService.update(id, quoteConverter.fromDto(quoteDto)));
    log.info("Successfully updated quote [{}]", updated);
    return updated;
  }

  @DeleteMapping("/{id}")
  public void deleteById(@PathVariable String id) {
    log.info("Deleting quote by id [{}]", id);
    quoteService.delete(id);
    log.info("Successfully deleted quote by id [{}]", id);
  }

}
```

Реактивний контролер:

```
@RestController
@RequestMapping("/reactive/quotes")
public class QuoteReactiveController {

  private final QuoteReactiveService quoteService;
  private final QuoteConverter quoteConverter;

  @GetMapping
  public Flux<QuoteDto> getAll() {
    log.info("Retrieving all quotes");
    return quoteService.getAll()
        .map(quoteConverter::toDto)
        .doOnComplete(() -> log.info("Successfully retrieved all quotes"));
  }

  @GetMapping("/paged")
  public Flux<QuoteDto> getAll(@RequestParam("page") int page,
                               @RequestParam("size") int size) {
    log.info("Retrieving all quotes by page [{}] and size [{}]", page,
size);
    return quoteService.getAllByPage(page, size)
        .map(quoteConverter::toDto)
        .doOnComplete(() -> log.info("Successfully retrieved by page [{}]
and size [{}]", page, size));
  }

  @GetMapping("/{id}")
  public Mono<QuoteDto> getById(@PathVariable String id) {
    log.info("Retrieving quote by id [{}]", id);
    return quoteService.get(id)
        .map(quoteConverter::toDto)
        .doOnSuccess(retrieved -> log.info("Successfully retrieved quote
[{}]", retrieved));
  }
```

```
  @PostMapping
  @ResponseStatus(CREATED)
  public Mono<QuoteDto> create(@Validated({Default.class, OnCreate.class})
@RequestBody QuoteDto quoteDto) {
    log.info("Creating quote [{}]", quoteDto);
    return quoteService.create(quoteConverter.fromDto(quoteDto))
        .map(quoteConverter::toDto)
        .doOnSuccess(created -> log.info("Successfully created quote [{}]",
created));
  }

  @PutMapping("/{id}")
  public Mono<QuoteDto> update(@PathVariable String id,
                               @Validated @RequestBody QuoteDto quoteDto) {
    log.info("Updating quote by id [{}], [{}]", id, quoteDto);
    return quoteService.update(id, quoteConverter.fromDto(quoteDto))
        .map(quoteConverter::toDto)
        .doOnSuccess(updated -> log.info("Successfully updated quote [{}]",
updated));
  }

  @DeleteMapping("/{id}")
  public Mono<Void> deleteById(@PathVariable String id) {
    log.info("Deleting quote by id [{}]", id);
    return quoteService.delete(id)
        .doOnSuccess(deleted -> log.info("Successfully deleted quote by id
[{}]", id));
  }

}
```

Єдина частина коду, яка здається іншою, в реактивному контролері — це типи повертаємих даних – об’єкти Flux та Mono. У Spring MVC при стандартному підході мали б повертатися колекції або власне об’єкти, які являли в собою тіло відповіді на запит.

Також як альтернатива функціям маршрутизатора замість анотованих контролерів (анотації @RestController і @GetMapping/@PostMapping/@PutMapping/@DeleteMapping). Реалізація була б зовсім іншою, але функціональність в результаті була б точно такою ж.

Контролер викликає QuoteMongoRepository, щоб отримати всі лапки. Завдяки реактивному підходу нам не потрібно, щоб повний список результатів був доступний у серверній частині перед їх отриманням: ми можемо використовувати об’єкти один за одним, як тільки драйвер MongoDB публікуватиме результати.

4.3.3 Тестування розробленого сервісу

4.3.3.1 Завантаження тестових даних

Для тестових сценаріїв необхідно мати якийсь набір записів в базі даних. Цю проблему можна вирішити попередньо завантажуючи визначений набір даних, який буде зберігатися у файлі, під час першого запуску програми.

Під час першого запуску програми кожен абзац цього тексту (з файлу data.txt у ресурсах проекту) буде збережено як об’єкт Quote в MongoDB. Щоб досягти цього, ми вводимо реалізацію ApplicationRunner в контекст програми: клас QuoteDataLoader. Цей метод буде виконаний одразу після того, як програма успішно запуститься. Для завантаження даних і уникнення їхнього дублювання спочатку буде виконуватись перевірка на нульову кількість елементів в базі даних. У випадку істинності умови буде створюємо Flux з потоку BufferedReader і кожний рядок файлу з ресурсів буде завантажено в базу даних. Для ідентифікаторів унікальних ідентифікаторів об’єктів буде використаний функціональний інтерфейс постачальника для створення послідовності.

```
@Slf4j
@Component
@RequiredArgsConstructor
public class QuoteDataLoader implements
ApplicationListener<ApplicationStartedEvent> {

  private final ApplicationProperties applicationProperties;
  private final QuoteReactiveRepository quoteReactiveRepository;

  @Override
  public void onApplicationEvent(ApplicationStartedEvent event) {
    log.info("Found {} entities in db",
quoteReactiveRepository.count().block());

    if (quoteReactiveRepository.findAll().count().block().equals(0L)) {
      Supplier<String> idGenerator = getIdGenerator();

      BufferedReader dataReader = new BufferedReader(new InputStreamReader(

requireNonNull(getClass().getClassLoader().getResourceAsStream(application
Properties.getDataFile()))));

      List<Quote> quotes = dataReader.lines()
          .filter(line -> !line.trim().isEmpty())
          .map(line -> {
```

```
            String id = idGenerator.get();
            String book = String.format("book-%s", id);
            return new Quote(id, book, line);
          })
          .toList();

      Flux.fromIterable(quotes)
          .flatMap(quoteReactiveRepository::save)
          .subscribe();

      log.info("Successfully loaded {} entities into db",
quoteReactiveRepository.count().block());
    }
  }

  private Supplier<String> getIdGenerator() {
    AtomicLong id = new AtomicLong();
    return () -> String.format("%05d", id.incrementAndGet());
  }

}
```

Завантажувач даних є хорошим прикладом використання реактивного стилю програмування з логікою блокування та самостійної підписки, оскільки інтерфейс ApplicationRunner не підготовлений до реактивного підходу: оскільки використовуваний репозиторій є реактивним, необхідно викликати у нього метод block(), щоб дочекатися результату одноелементного видавця (Mono), що містить кількість лапок у сховищі (метод підрахунку). Це також торкається процесу завантаження об’єктів, які заздалегідь формуються в колекцію, після чого до кожного з них застосовується метод save() репозиторію. При завантаженні для цього треба підписатися на Flux:

```
Flux.fromIterable(quotes)
          .flatMap(quoteReactiveRepository::save)
          .subscribe();
```

Якби інтерфейс ApplicationRunner підтримував реактивний підхід, тобто тип повернення Flux або Mono, замість цього можна було би підписатися на метод count() і об’єднати Mono та потоки. Через відсутність існування такої реалізації у Spring, необхідно робити раніше описані кроки, інакше завантажити дані до завершення роботи виконавця буде неможливо.

Для тестування використовується фреймворк JUnit який надає потужний функціонал для тестування додатків, описаних на мові програмування Java з використанням Spring Boot.

Тестування додатків, які розробляються на основі фреймворку Spring Boot відбувається завдяки опису тестових класів і методів. Тестовий клас має містити анотацію @SpringBootTest, яка завантажує як стандартні конфігурації, необхідні для виконання тестів, так і ті, які описані в тестовому модулі і помічені @Configuration або @TestConfiguration.

Тестові методи повинні мати анотацію @Test, що свідчить про те, що даним метод є тестовим. При цьому на нього накладаються такі умови: метод не повинен мати модифікатор доступу private, він не повинен бути статичним та не має повертати значення (має бути void). Виконання тестів без внесення необхідних змін в тестову конфігурацію не має чіткого порядку

За замовчуванням методи тестування будуть впорядковані за допомогою алгоритму, який є детермінованим, але навмисно неочевидним. Це було зроблено для того, щоб гарантувати, що наступні запуски набору тестів виконують тестові методи в тому ж порядку. Ця поведінка в залежності від вимог до тестів може бути змінена на виконання тестів в алфавітному порядку або в залежності від привласненого порядкового номеру.

Серед описаних тестових сценаріїв присутні як негативні (ті, які очікують, що при обробці запиту станеться помилка і тіло, яке повернеться буде мати очікуваний статус-код і повідомлення про помилку), так і позитивні кейси (ті, які очікують у тілі відповіді параметри згідно ситуації, яка моделюється в тесті).

Приклади негативних сценаріїв (блокуючий підхід):

- очікування відповідних коду і повідомлення при умові якщо об’єкта з переданим ідентифікатором не існує

```
  @Test
  @SneakyThrows
  void getQuote_notExisting_returnsErrorResponse() {
    String notExistingId = getUniqueId();

    String expectedErrorCode = ErrorCode.ENTITY_NOT_FOUND;
    String expectedErrorMessage = String.format("Quote with id [%s]
not found", notExistingId);

    mockMvc.perform(get(QUOTES_ENDPOINT + "/{id}", notExistingId))
        .andExpect(status().isNotFound())
        .andExpect(jsonPath("code").value(expectedErrorCode))
        .andExpect(jsonPath("message").value(expectedErrorMessage))
        .andExpect(jsonPath("at").exists());
  }
```

- спроба видалити об’єкт з неіснуючим ідентифікатором

```
  @Test
  @SneakyThrows
  void deleteQuote_notExisting_returnsErrorResponse() {
    String notExistingId = getUniqueId();

    String expectedErrorCode = ErrorCode.ENTITY_NOT_FOUND;
    String expectedErrorMessage = String.format("Quote with id [%s]
not found", notExistingId);

    mockMvc.perform(delete(QUOTES_ENDPOINT + "/{id}", notExistingId))
        .andExpect(status().isNotFound())
        .andExpect(jsonPath("code").value(expectedErrorCode))
        .andExpect(jsonPath("message").value(expectedErrorMessage))
        .andExpect(jsonPath("at").exists());
  }
```

Приклади позитивних сценаріїв (реактивний підхід):

- отримання об’єкта за його ідентифікатором

```
  @Test
  void getQuote_isOk() {
    webTestClient.get()
        .uri(QUOTES_ENDPOINT + "/{id}", quote.getId())
        .exchange().expectStatus().isOk()
        .expectBody()
        .jsonPath("id").isEqualTo(quote.getId())
        .jsonPath("book").isEqualTo(quote.getBook())
        .jsonPath("content").isEqualTo(quote.getContent());
  }
```

- перевірка чи присутній об’єкт в базі даних після його створення

```
@Test
void createQuote_isOk() {
  QuoteDto newQuote = prepareQuoteDto();

  webTestClient.post()
      .uri(QUOTES_ENDPOINT)
      .contentType(MediaType.APPLICATION_JSON)
      .bodyValue(serialize(newQuote))
      .exchange().expectStatus().isCreated()
      .expectBody()
      .jsonPath("id").isEqualTo(newQuote.getId())
      .jsonPath("book").isEqualTo(newQuote.getBook())
      .jsonPath("content").isEqualTo(newQuote.getContent());

  webTestClient.get()
      .uri(QUOTES_ENDPOINT + "/{id}", newQuote.getId())
      .exchange().expectStatus().isOk()
      .expectBody()
      .jsonPath("id").isEqualTo(newQuote.getId())
      .jsonPath("book").isEqualTo(newQuote.getBook())
      .jsonPath("content").isEqualTo(newQuote.getContent());
}
```

Щоб правильно оцінити Reactive Web, необхідно змоделювати такі проблеми, як нерегулярна затримка мережі або перевантаженість серверу. Для прикладу можна імітувати ситуацію, коли на обробку об’єкта витрачається фіксований час (затримка).

Наявність змодельованої затримки також допоможе нам уявити відмінності між стратегіями Reactive та MVC. Клієнт і сервер для цього можна запустити на одній машині, тому, якщо не використовувати затримку, час відповіді буде настільки малим, що буде важко помітити відмінності.

Опискаючись трохи більше в деталі необхідно розібратись як маючи в сигнатурі методу Flux<Quote> клієнт отримує список об’єктів Quote. Документація Spring WebFlux перелічує реактивні типи (включаючи Flux) як такі, які можуть бути повернуті як результат методу контролера. Отже, що відбувається, коли ми запитуємо вміст на сервер? Як WebFlux перетворює дані в дійсну відповідь? Насправді це залежить від того, як це відбувається:

- якщо надсилати запит, не використовуючи заголовок Accept, або його значення проставляти як application/json, в результаті відбувається процес блокування та клієнт отримує відповідь у форматі JSON.
- якщо необхідна повна реактивність і підтримка відправлених сервером подій, для реалізації повного реактивного стеку, клієнт повинен підтримувати відповідь Event-Stream. Для цього необхідно, щоб заголовок Accept в запиті мав значення text/event-stream, що активує реактивну функціональність у Spring, щоб публікувати події між сервером і клієнтом.

Для того, щоб сервіс запустився, необхідно мати активну базу даних. Для цього можна використовувати локально запущену MongoDB, яка буде запускатись завдяки Docker та docker-compose:

```
version: "3.9"

services:
  mongo:
    container_name: reactive-mongo
    image: mongo
    ports:
      - "27017:27017"
```

Для збирання проекту використовується утиліта Gradle, яка дозволяє додавати зовнішні залежності так компілювати проект. Для того, щоб скомпілювався проект, необхідно мати запущеним Docker. Щоб запустити його, слід спочатку скомпілювати проект:

```
>>> gradle build
>>> java -jar build/libs/reactive-spring-1.0-SNAPSHOT.jar
```

Версії мають бути: Gradle не нижче 7.3.3, Java не нижче 17.0.2, Docker не нижче 4.8.2.

4.3.3.2 Опис інтеграційних тестів

Тестування реактивного та блокуючого контролерів відбувається за однаковим принципом та сценаріями. Єдине, що відрізняється –

використовувані інструменти для тестування. Для тестування блокуючого контролера використовується MockMvc, а реактивного – WebTestClient.

Клас MockMvc призначений для тестування контролерів. Він дає змогу тестувати контролери без запуску http-сервера. Тобто, при виконанні тестів мережеве з'єднання не створюється. З MockMvc можна писати як інтеграційні тести, і unit-тести. Для цього необхідно над тестовим класом додати анотацію @AutoConfigureMockMvc щоб Spring Test автоматично налаштував всі необхідні залежні об’єкти та сервіси, які будуть використовуватись при тестуванні.

WebTestClient — це обгортка навколо WebClient, яка використовує його для виконання запитів і описує API для перевірки відповідей, статус-коду та наявності повідомлення. WebTestClient прив’язується до програми WebFlux за допомогою фіктивного запиту та відповіді, або може перевірити будь-який веб-сервер через з’єднання HTTP. Для цього необхідно над тестовим класом додати анотацію @AutoConfigureWebTestClient та вказати, що сервіс, який тестується, реактивний: @SpringBootTest(properties = "spring.main.web-application-type=reactive")

Для інтеграційних тестів використовується залежність testcontainers для MongoDB, яка при компіляції проекту використовується для запуску окремого інстансу бази даних, яка спеціально виділяється під тестування. Весь цикл життя її покладається на реалізацію testcontainers. В купі це дає змогу протестувати одночасно всі компоненти розробленого сервісу та виявити помилки реалізації.

Для того, щоб testcontainers запустили MongoDB, необхідно описати створення об’єкту контейнера в коді. Це може бути опис у стилі testcontainers з відповідними анотаціями @Testcontainers та @Container, який має бути присутнім в кожному тестовому класі, який використовує сервіси, які роблять запити до бази даних:

```
@Testcontainers
public class MongoDbTestContainer {

  private static final String MONGODB_IMAGE = "mongo:4.4.1";
  private static final Integer MONGODB_MAPPED_PORT = 27017;

  @Container
  static MongoDBContainer mongoDBContainer = new
MongoDBContainer(MONGODB_IMAGE);

  @DynamicPropertySource
  static void mongoDbProperties(DynamicPropertyRegistry registry) {
    registry.add("spring.data.mongodb.port", () ->
mongoDBContainer.getMappedPort(MONGODB_MAPPED_PORT));
    registry.add("spring.data.mongodb.host", mongoDBContainer::getHost);
  }

}
```

при цьому до конфігурації контексту після запуску контейнера з базою даних, будуть внесені зміни до констант, які використовуються при підключенні до бази даних клієнтами MongoDB.

Або створити об’єкт контейнеру, який мануально запустити, після чого сконфігурувати клієнтів (в даному випадку блокуючого та реактивного) бази даних:

```
@TestConfiguration
public class MongoDbConfig {

  private static final String MONGODB_IMAGE = "mongo:4.4.1";
  private static final Integer MONGODB_MAPPED_PORT = 27017;

  @Bean(initMethod = "start", destroyMethod = "stop")
  public MongoDBContainer mongoDbContainer() {
    return new MongoDBContainer(MONGODB_IMAGE)
        .withExposedPorts(MONGODB_MAPPED_PORT);
  }

  @Bean
  public MongoClientFactoryBean mongoClient(MongoDBContainer
mongoDbContainer) {
    MongoClientFactoryBean mongo = new MongoClientFactoryBean();
    mongo.setHost(mongoDbContainer.getHost());
    mongo.setPort(mongoDbContainer.getMappedPort(MONGODB_MAPPED_PORT));
    return mongo;
  }

  @Bean
  public ReactiveMongoClientFactoryBean
mongoClientReactive(MongoDBContainer mongoDbContainer) {
    ReactiveMongoClientFactoryBean mongo = new
ReactiveMongoClientFactoryBean();
    mongo.setHost(mongoDbContainer.getHost());
```

```
        mongo.setPort(mongoDbContainer.getMappedPort(MONGODB_MAPPED_PORT));
        return mongo;
    }

}
```

Другий підхід дозволяє не змінювати конфігурацію сервісу та не звертатися до конкретних констант, а зосередитись лише на використовуваних інструментах, при цьому гнучко використовуючи функціонал, який надає об’єкт контейнеру – хост та порт підключення до бази даних, які можуть бути використані при створенні клієнтів після запуску контейнеру.

4.3.3.3 Результати тестування

Описані тести містять в собі як позитивні, так і негативні сценарії, які перевіряють коректність поведінки сервісу на різні комбінації запитів та критичні ситуації. Всього було описано 22 тестових сценарії.

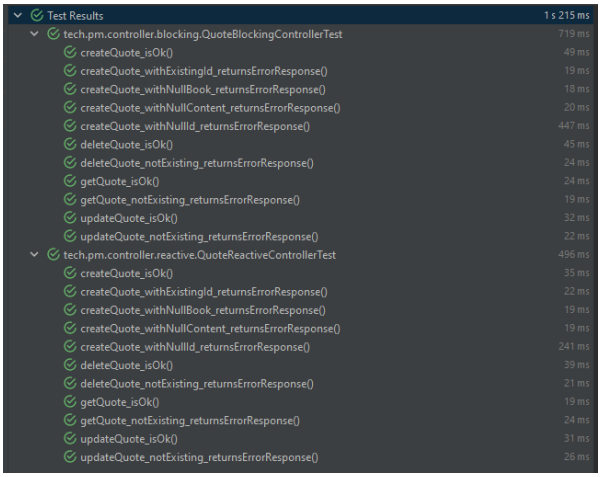

Рис. 4.3 – Результат виконання тестових сценаріїв

Для тестування перформансу розроблених контролерів, які використовують різні підходи, був описаний тестовий клас, який необхідно запускати лише при активному сервері, так як запити мають робитися в базу даних з великим об’ємом інформації (в нашому випадку це 7336 записів):

```
Found 7336 entities in db
```

Рис. 4.4 – Інформація про кількість записів в базі даних

```
@Slf4j
//@Disabled // comment to run benchmark
public class BenchmarkTest {

  private static final int THREADS_AMOUNT = 32;

  private static final String BLOCKING_QUOTES_ENDPOINT =
"/blocking/quotes";
  private static final String REACTIVE_QUOTES_ENDPOINT =
"/reactive/quotes";

  private static final String PAGED_REQUEST = "/paged?page=5&size=100";

  private static final String BASE_URL = "http://localhost:8444";

  @ParameterizedTest
  @MethodSource("requests")
  void benchmark_blocking(int threadsAmount) {
    run(threadsAmount, THREADS_AMOUNT, BLOCKING_QUOTES_ENDPOINT +
PAGED_REQUEST);
  }

  @ParameterizedTest
  @MethodSource("requests")
  void benchmark_reactive(int threadsAmount) {
    run(threadsAmount, THREADS_AMOUNT, REACTIVE_QUOTES_ENDPOINT +
PAGED_REQUEST);
  }

  @SneakyThrows
  private void run(int requestsAmount, int threadsAmount, String endpoint)
{
    long start = System.nanoTime();

    WebClient webClient = WebClient.create(BASE_URL);
    Map<Integer, BenchmarkRequestResult<List<QuoteDto>>> results = new
HashMap<>();

    List<Callable<BenchmarkRequestResult<List<QuoteDto>>>>
requestCallableList = IntStream.range(0, requestsAmount)
        .mapToObj(i -> createMonoRequest(i, webClient, endpoint))
        .collect(Collectors.toList());

    log.info("========== NEW BENCHMARK --> RequestsAmount: {},
ThreadsAmount: {}, Endpoint: {}", requestsAmount, threadsAmount, endpoint);
    log.info("========== Requests created");

    ExecutorService executorService =
Executors.newFixedThreadPool(threadsAmount);
    ExecutorCompletionService<BenchmarkRequestResult<List<QuoteDto>>>
completionService = new ExecutorCompletionService<>(executorService);
    requestCallableList.forEach(completionService::submit);

    log.info("========== Requests submitted @ {}",
Duration.ofNanos(System.nanoTime() - start));

    for (int n = 0; n < requestCallableList.size(); n++) {
      BenchmarkRequestResult<List<QuoteDto>> benchmarkRequestResult =
completionService.take().get();
```

```
      results.put(benchmarkRequestResult.getRequestId(),
benchmarkRequestResult);
    }

    log.info("==========       Requests       completed       @       {}",
Duration.ofNanos(System.nanoTime() - start));

    log.info("========== RESULTS ==========");

    double avg = results.values()
        .stream()
        .mapToLong(BenchmarkRequestResult::getTookTimeNs)
        .average()
        .getAsDouble();

    log.info("Average         time         per         request:         {}",
Duration.ofNanos(Math.round(avg)));

    double successRate = results.values().stream().
        filter(r                                                          ->
r.getResponseEntity().getStatusCode().equals(HttpStatus.OK))
        .count() * 100.0 / results.size();

    double errorRate = 100.0 - successRate;

    log.info("Success Rate: {}", successRate);
    log.info("Error Rate:   {}", errorRate);

    long objectsNumber = results.values()
        .stream()
        .map(r -> r.getResponseEntity().getBody().size())
        .reduce(Integer::sum)
        .get();

    log.info("Total Number of objects: {}", objectsNumber);

    long end = System.nanoTime();
    log.info("==========  Benchmark  took  {}  ",  Duration.ofNanos(end  -
start));
  }

  private                  Callable<BenchmarkRequestResult<List<QuoteDto>>>
createMonoRequest(int requestId, WebClient webClient, final String uri) {
    return () -> {
      long start = System.nanoTime();

      ResponseEntity<List<QuoteDto>> responseEntity = webClient.get()
          .uri(uri)
          .retrieve()
          .toEntity(new ParameterizedTypeReference<List<QuoteDto>>() {})
          .block();

      long end = System.nanoTime();

      return   new   BenchmarkRequestResult<>(requestId,   end   -   start,
responseEntity);
    };
  }

  private static Stream<Arguments> requests() {
```

```
    return Stream.of(
        Arguments.of(1),
        Arguments.of(16),
        Arguments.of(64),
        Arguments.of(128)
    );
  }

}
```

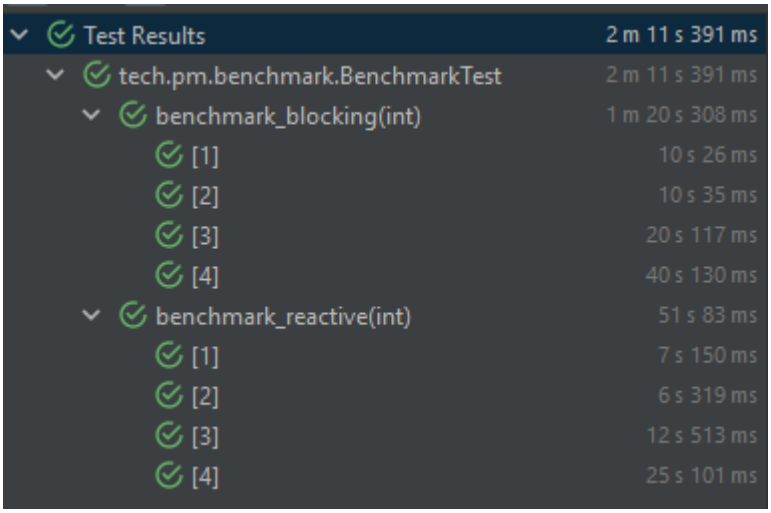

Рис. 4.5 – Результати виконання тестів

Описані методи для тестування блокуючого та реактивного контролерів виконуються відповідно для кожного елементу з потоку даних по визначеному порядку:

```
  private static Stream<Arguments> requests() {
    return Stream.of(
        Arguments.of(1),
        Arguments.of(16),
        Arguments.of(64),
        Arguments.of(128),
        ...
    );
  }
```

Ці числа характеризують кількість запитів, які будуть надіслані одночасно. По результатам цих тестів можна побачити явну перевагу при обробці запитів – середній час обробки запиту при блокуючому підході за результатами складає 10.024 секунди, в той час як реактивний підхід показує кращу швидкодію і в середньому оброблює запит за 6.255 секунд:

Тест блокуючого контролера:

```
========== NEW BENCHMARK --> RequestsAmount: 128, ThreadsAmount: 32,
Endpoint: /blocking/quotes/paged?page=5&size=100
========== Requests created
========== Requests submitted @ PT0.0056262S
========== Requests completed @ PT40.1277638S
========== RESULTS ==========
Average time per request: PT10.024860575S
Success Rate: 100.0
Error Rate:   0.0
Total Number of objects: 12800
========== Benchmark took PT40.1297692S
```

Тест реактивного контролера:

```
========== NEW BENCHMARK --> RequestsAmount: 128, ThreadsAmount: 32,
Endpoint: /reactive/quotes/paged?page=5&size=100
========== Requests created
========== Requests submitted @ PT0.0039934S
========== Requests completed @ PT25.0982182S
========== RESULTS ==========
Average time per request: PT6.255917027S
Success Rate: 100.0
Error Rate:   0.0
Total Number of objects: 12800
========== Benchmark took PT25.0996523S
```

За результатами тестування можна впевнитись в явній перевазі реактивного підходу до програмування.

4.4 Висновки до розділу

У даному розділі були розглянуті особливості мікросервісів, які побудовані з використанням реактивного підходу, були визначені можливості застосування брокерів повідомлень, були описані їхні переваги та недоліки використання, був наданий приклад застосування Spring WebFlux, для розробленого проекту проведено інтеграційне тестування використовуючи різні підходи – блокуючий та реактивний. Завдяки функціональному тестуванню було виявлено, що реактивна реалізація сервісу при великому навантаженні в середньому обробляє запити швидше майже у 2 рази.

5 ВИКОРИСТАННЯ ХМАРНОЇ ІНФРАСТРУКТУРИ ДЛЯ РОЗГОРТАННЯ СЕРВІСУ

Для розгортання сервісу у хмарній інфраструктурі скористаємося інструментами, які надає хмарна платформа Heroku [23]. Безкоштовний функціонал даної платформи повністю підходить для розгортання невеликих проектів і надає багато інструментаріїв (такі сторонні надбудови над сервісом як логування, моніторинг метрик, формування звітів, кешування тощо). Для розгортання бази даних скористаємося стандартним планом MongoDB Cloud [24] – безкоштовного функціоналу цілком достатньо для розробленого сервісу.

5.1 Розгортання бази даних MongoDB

Так як Heroku має функціонал лише для розгортання сервісів, а розроблений додаток вимагає підключення до сторонньої бази даних, необхідно налаштувати MongoDB, як було зазначено раніше, в іншому додатку – MongoDB Cloud.

Для розгортання бази даних необхідно створити кластер:

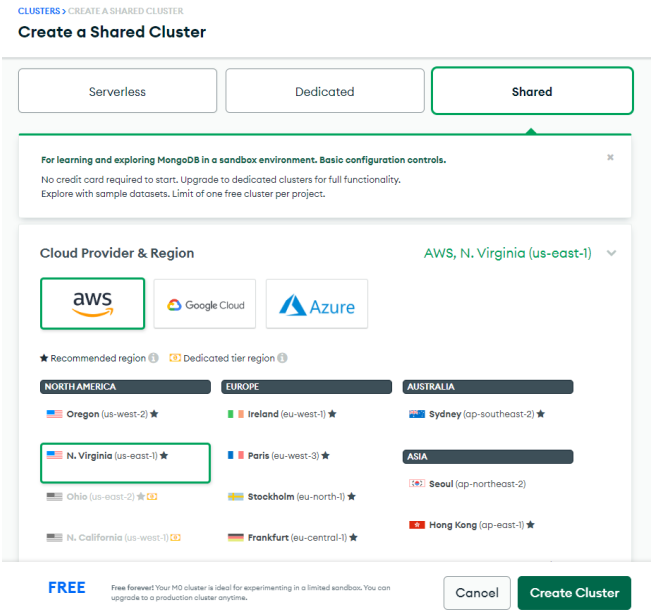

Рис. 5.1 – Створення кластеру MongoDB

при цьому впевнитись, що обраний безкоштовний план з мінімальним об’ємом пам’яті, яка виділена під кластер бази даних:

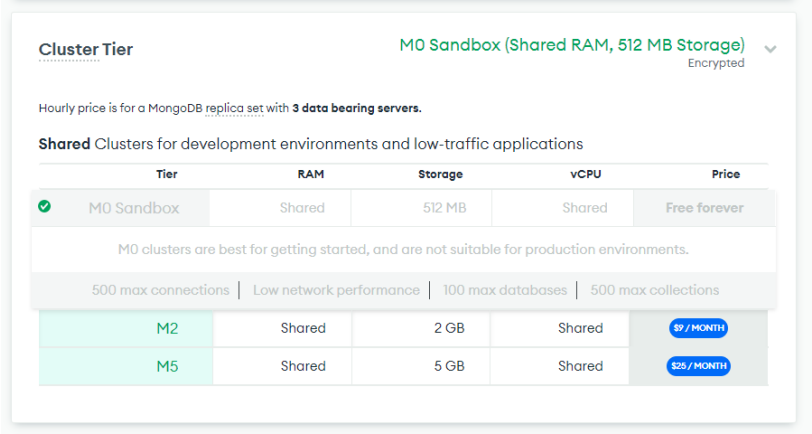

Рис. 5.2 – Вибір об’єму пам’яті кластера

Після створення кластеру необхідно створити користувача, авторизаційні дані якого будуть використовуватись при підключенні, вказати йому роль, а також додати IP сервісу у whitelist для того, щоб мати змогу взаємодіяти з базою.

Створення користувача з адміністраторськими правами:

Рис. 5.3 – Список користувачів, які мають доступ до бази даних

Після створення користувача необхідно дозволити доступ до бази даних з будь-якого IP (0.0.0.0):

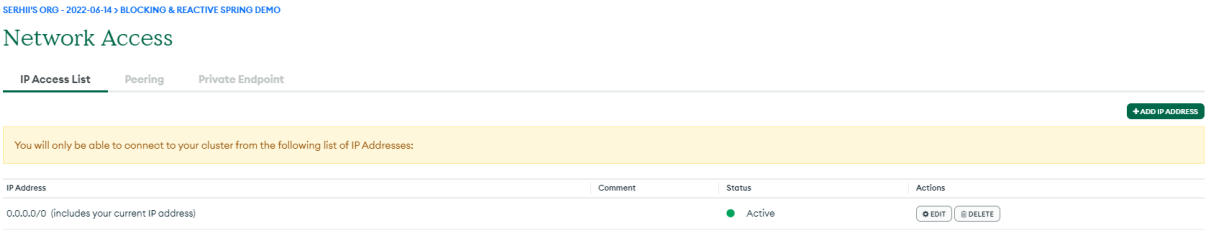

Рис. 5.4 – Список IP, з яких можуть підключатися сервіси для доступу до бази даних

Створений кластер відображається на головній сторінці:

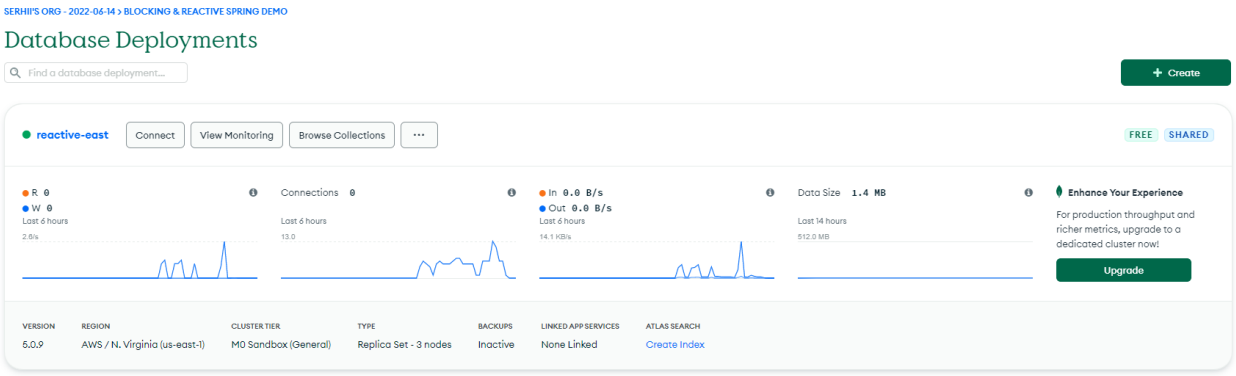

Рис. 5.5 – Відображення кластеру MongoDB

Після цього необхідно запам’ятати посилання URI (кнопка Connect на Рис. 5.5), яке буде використовуватись для підключення до бази даних клієнтами блокуючого та реактивного репозиторіїв з побудованого сервісу:

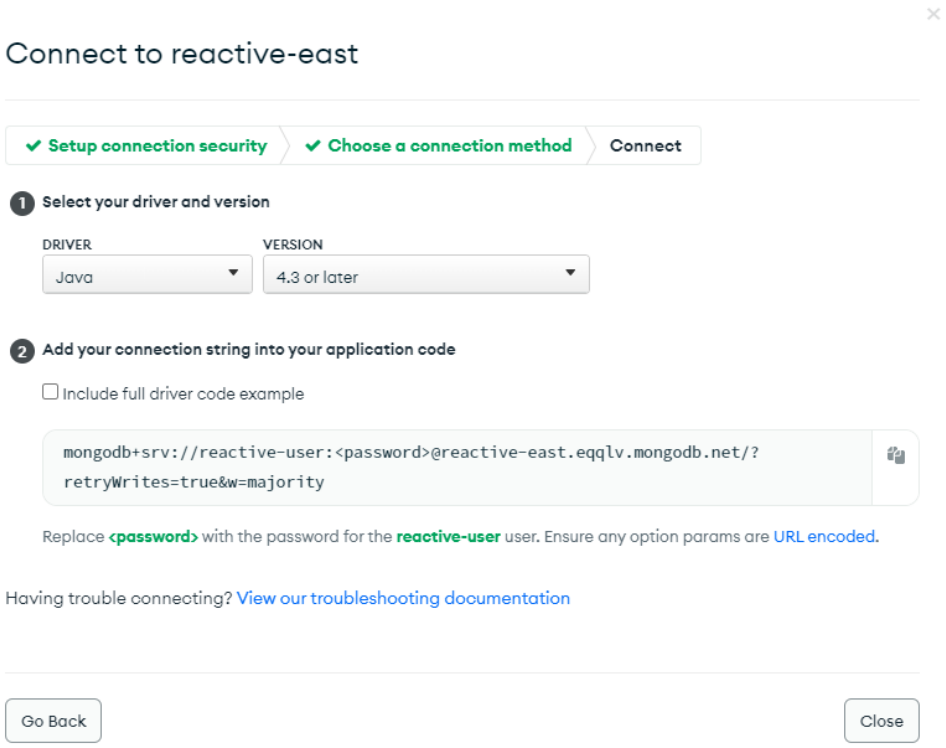

Рис. 5.6 – Посилання для підключення до бази даних

В загальному випадку посиланням виглядатиме як:

```
mongodb+srv://<login>:<password>@<url>/?retryWrites=true&w=majority
```

де login – логін створеного користувача, який має доступ до бази даних, password – пароль користувача, url – унікальний вказівник на ресурс, який автоматично генерується платформою.

5.2 Розгортання сервісу на платформі Heroku

Для реєстрації сервісу необхідно вказати ім’я сервісу:

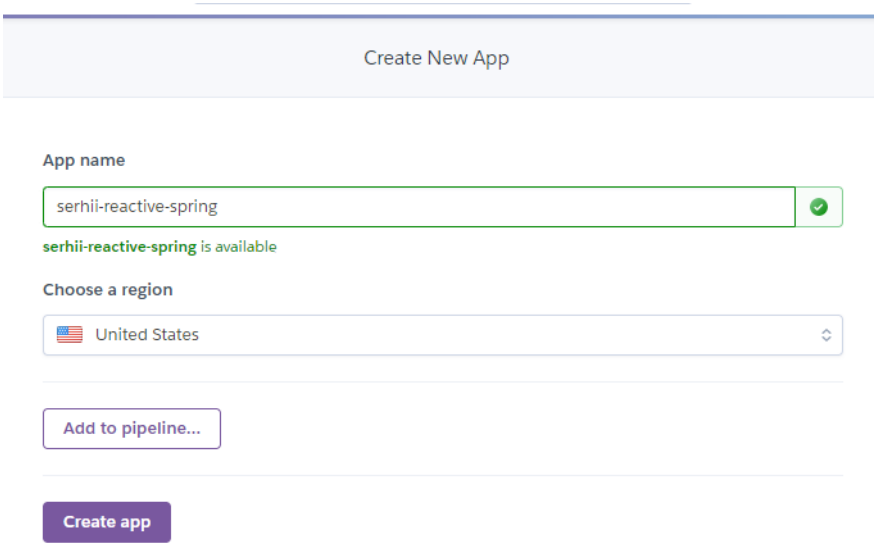

Рис. 5.7 – Створення сервісу

після чого він з’являється на головній сторінці в списку серед інших сервісів:

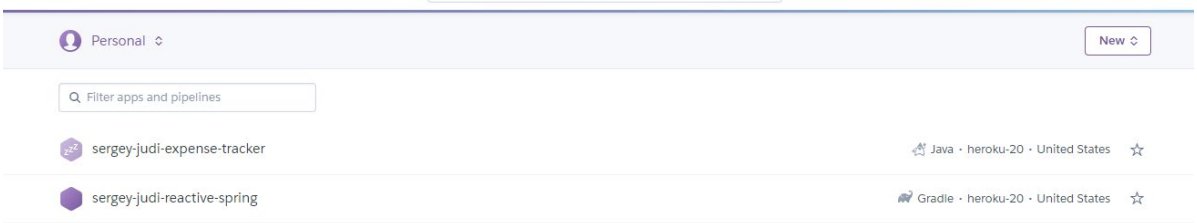

Рис. 5.8 – Список сервісів

Для розгортання сервісу необхідно вказати спосіб розгортання:

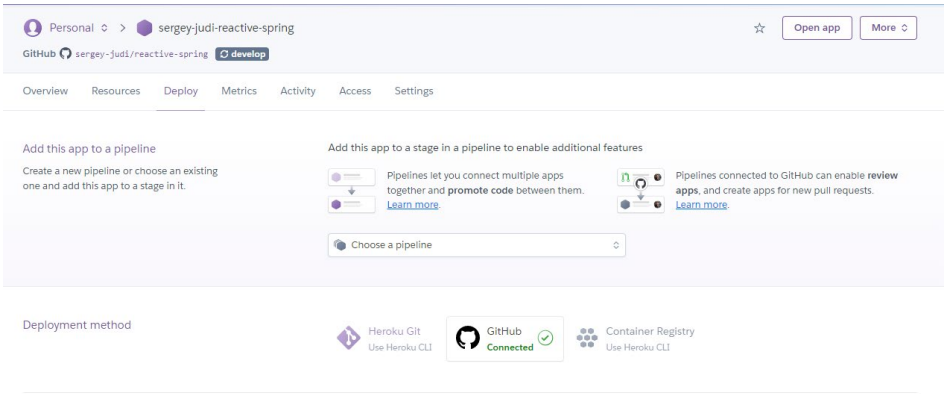

Рис. 5.9 – Вибір способу розгортання сервісу

це може бути Heroku Git – інструмент для завантаження сервісів через Heroku CLI, налаштування їхніх конфігурацій та взаємодії з ними, GitHub-репозиторій – Heroku автоматично буде клонувати репозиторій в свою локальну систему, після чого буде відбуватися розгортання сервісу, Container Registry – завантаження Docker-контейнеру через Heroku CLI.

Так як при розробці додатку використовувалась система контролю версій git та GitHub, доцільно буде обрати саме цю опцію, оскільки все відбувається в автоматичному режимі, через що відсутня необхідність мануального втручання у процес розгортання сервісу.

Для сервісу, який буде розгортатися у хмарній системі, був створений окремий файл конфігурації, який відповідає профілю stage:

```
spring:
  main:
    banner-mode: off
  data:
    mongodb:
      uri: ${MONGODB_URI}
  jackson:
    default-property-inclusion: non_null
    deserialization:
      fail-on-unknown-properties: false

service:
  data-file: data.txt
  delay:
    enabled: false
    duration: 50ms
```

де посилання для підключення до MongoDB вичитується зі змінної середовища MONGODB_URI (необхідно винести в окрему змінну так як зберігати посилання з логіном, паролем та ресурсом в репозиторії є поганою практикою, оскільки будь-яка людина зможе отримати доступ до бази даних, доступ до якої може бути наданий для будь-якого IP).

Додавання змінних середовища в конфігурацію Heroku:

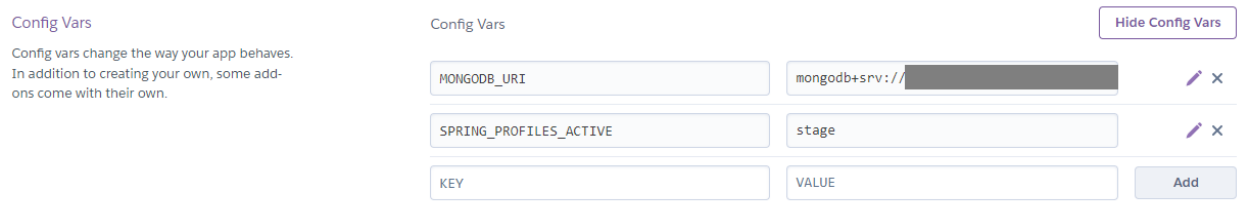

Рис. 5.10 – Змінні середовища Heroku

При цьому як значення MONGODB_URI встановлюється посилання, яке було отримане в MongoDB Cloud, а SPRING_PROFILES_ACTIVE – значення stage.

Для розгортання сервісу необхідно обрати гілку репозиторію, яка буде використовуватись в якості джерела кодової бази для сервісу:

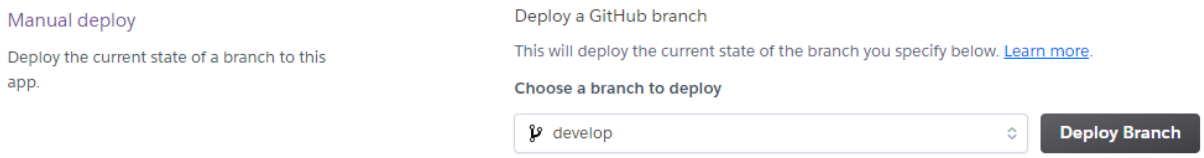

Рис. 5.11 – Меню вибору гілки репозиторію для розгортання сервісу

Якщо в репозиторії з розробленим сервісом присутній CI крок, який виконує тести, наприклад для GitHub:

```
name: Java CI

on: [push]

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Setup JDK 17
```

```
      uses: actions/setup-java@v2
      with:
        java-version: 17
        distribution: adopt
        cache: gradle
    - name: Build with Gradle
      uses: gradle/gradle-build-action@v2
      with:
        gradle-version: 7.3.3
        arguments: build
```

можна обрати опцію автоматичного розгортання сервісу після успішного проходження тестів в гілці develop:

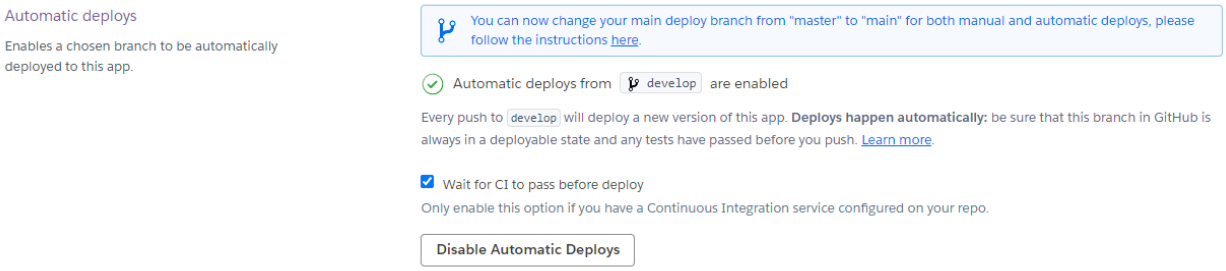

Рис. 5.12 – Опція автоматичного розгортання сервісу

Для мануального розгортання сервісу необхідно натиснути кнопку “Deploy Branch”. Heroku скопіює гілку develop, проведе компіляцію проекту, його збірку та розгорне сервіс.

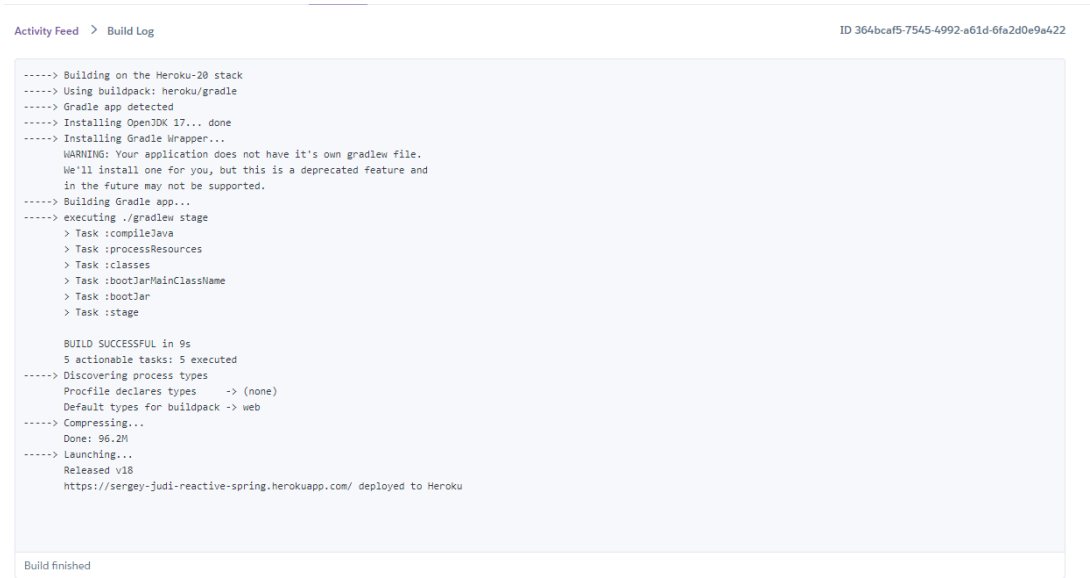

Рис. 5.13 – Компіляція проекту

Рис. 5.14 – Початок розгортання сервісу

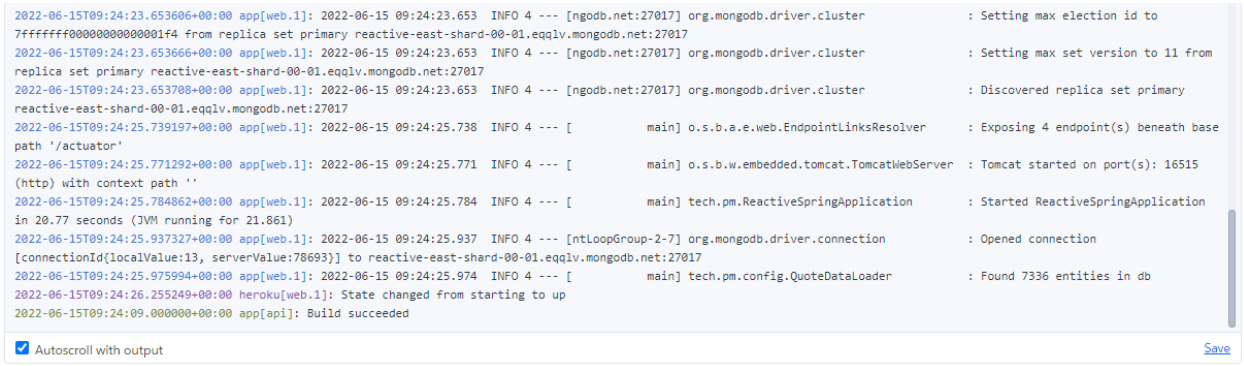

Рис. 5.15 – Успішне розгортання сервісу

В результаті сервіс розгорнутий за посиланням https://sergey-judi-reactive-spring.herokuapp.com/ і має такі ендпоінти:

- https://sergey-judi-reactive-spring.herokuapp.com/blocking/quotes
- https://sergey-judi-reactive-spring.herokuapp.com/reactive/quotes

які повністю підтримують CRUD-операції.

5.3 Висновки до розділу

У даному розділі було розгорнуто хмарну інфраструктуру для описаного сервісу та його залежностей. База даних MongoDB була розгорнута у кластері MongoDB Cloud використовуючи безкоштовний функціонал, а сам сервіс був розгорнути у хмарній платформі Heroku. Побудована система дозволяє автоматично розгортати сервіс у хмарі завдяки інструментам GitHub та Heroku.

6 ФУНКЦІОНАЛЬНО-ВАРТІСНИЙ АНАЛІЗ ПРОГРАМНОГО ПРОДУКТУ

У даному розділі надається оцінка основних характеристик продукту, розробленого для дослідження застосування асинхронного програмування у мікросервісній архітектурі. Даний продукт в якості методичних матеріалів призначено для використання на персональних комп’ютерах під управлінням будь-якої операційної системи та апаратного забезпечення.

Нижче наводиться аналіз різних варіантів реалізації модулю з метою вибору оптимальної стратегії створення програмного продукту, враховуючи при цьому економічні фактори та основні характеристики продукту, що впливають на продуктивність роботи і на його сумісність з апаратним забезпеченням. Для цього було використано апарат функціонально-вартісного аналізу.

Функціонально-вартісний аналіз (ФВА) – це технологія, яка оцінює реальну вартість продукту або послуги незалежно від організаційної структури компанії. Обидва види витрат – прямі та побічні – розподіляються в залежності від потрібних обсягів ресурсів на кожному етапі.

Метою функціонально-вартісного аналізу є забезпечення найбільш оптимального розподілення ресурсів, які були виділені на виробництво продукції або надання послуг. У даному випадку – аналізу функцій програмного продукту й виявлення усіх витрат на реалізацію цих функцій.

Коротко кажучи, даний метод аналізу розробленого продукту починається з визначення послідовності функцій, необхідних для виробництва продукту. Перш за все спочатку йдуть всі можливі функції, які розподіляються по двом групам: ті, що впливають на вартість продукту і ні. Додатково на цьому етапі оптимізується сама послідовність скороченням кроків, що не впливають на витрати.

Для кожної функції застосовується визначення повного обсягу річних витрат та кількість робочих часів. Базуючись на отриманих оцінках визначається кількісна характеристика джерел витрат, після чого проводиться кінцевий розрахунок витрат на виробництво продукту.

6.1 Постановка задачі техніко-економічного аналізу

Даний метод ФВА застосовується для проведення техніко-економічного аналізу розробки реактивного додатку та його тестування, який буде застосовуватись в мікросервісній системі. Так як основні сервісні рішення стосуються всієї системи, кожна окрема підсистема має їм задовольняти. Тому аналіз представляє собою аналіз фреймворку, на якому будується реактивний мікросервіс.

Технічні вимоги до фреймворку:

1. висока швидкість обробки даних та відповідь користувачеві у реальному часі
2. зручність та простота взаємодії
3. висока щвидкість розгортання та розробки
4. можливість зручного налаштування, масштабування та обслуговування
5. широка вживаність та поширення серед спільноти

6.1.1 Обґрунтування функцій програмного продукту

Головна функція F_0 – розробка програмного продукту, який вирішує задачу для виявлення та фільтрування агресивних висловлювань в коментарях в соціальних мережах. Виходячи з конкретної мети, можна виділити наступні основні функції програмного продукту:

F_1 – вибір мови програмування

F_2 – вибір фреймворку

F_3 – вибір середовища розробки

Кожна з основних функцій може мати декілька варіантів реалізації.

Функція F_1:

1) Python
2) Java

Функція F_2:

1) Spring WebFlux
2) Akka
3) Vert.x

Функція F_3:

1) PyCharm
2) IntelliJ IDEA

6.1.2 Варіанти реалізацій основних функцій

Варіанти реалізації наявних функцій наведені у морфологічній карті системи на рисунку 6.1.2.1. Морфологічна карта містить всі можливі комбінації варіантів реалізації функцій, які складають повну множину варіантів ПП.

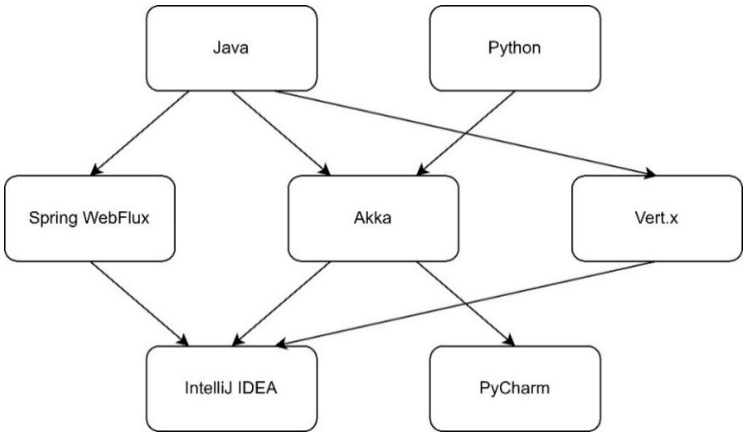

Рис. 6.1 – Морфологічна карта варіанів реалізації функцій

На основі цієї карти було побудовано позитивно-негативну матрицю варіантів основних функцій (таблиця 6.1.2.1).

Функція	Варіант реалізації	Переваги	Недоліки
F_1	А	Кросплатформна мова програмування, займає менше часу на написання коду	Займає більше часу для виконання коду
	Б	Кросплатформна мова програмування, широко поширена у веб-сервісах, висока швидкодія	Займає більше часу на написання коду
F_2	А	Безкоштовне використання, широка вживаність, обширна документація, велика спільнота	Прив’язаність до екосистеми, використовується лише в мові програмування Java
	Б	Безкоштовне використання, гарна документація, підтримка багатьох мов	Високий поріг входу
	В	Висока швидкодія в мобільних додатках	Невеликий розмір підтримки від спільноти, використовується лише в мові програмування Java
F_3	А	Висока підтримка від JetBrains, поширеність серед спільноти	Використовується тільки для мови програмування Java
	Б	Висока підтримка від JetBrains, поширеність серед спільноти	Використовується тільки для мови програмування Python

Табл. 6.1 – Позитивно-негативна матриця варіантів основних функцій

Проаналізувавши позитивно-негативну матрицю можна зробити висновок, що при розробці програмного продукту деякі варіанти реалізації функцій варто відкинути, тому, що вони не відповідають поставленим перед програмним продуктом задачам. Ці варіанти наявні у морфологічній карті.

Функція F_1:

Оскільки реалізація програмного коду для вирішення поставленої задачі потребує підтримку фреймворків, необхідно обрати мову, завдяки якій буде описана реалізація якнайвдобніше, тому варіант А має бути відкинутий.

Функція F_2:

Описані фреймворки повністю підтримуть мову програмування Java, тому вважаємо всі варіанти гідними розгляду.

Функція F_3:

Оскільки, програмний продукт реалізується мовою Java, варіант Б можна відкинути, оскільки він не підтримує цю мову програмування.

Таким чином, будемо розглядати такі варіанти реалізації програмного продукту:

- F_1(А) – F_2(А) – F_3(А)
- F_1(А) – F_2(Б) – F_3(А)
- F_1(А) – F_2(В) – F_3(А)

6.2 Обґрунтування системи параметрів програмного продукту

6.2.1 Опис параметрів

На підставі даних про основні функції, що повинен реалізувати програмний продукт, вимог до нього, визначаються основні параметри виробу, що будуть використані для розрахунку коефіцієнта технічного рівня.

Для того, щоб охарактеризувати програмний продукт, будемо використовувати наступні параметри:

- X1 – об’єм пам’яті для збереження даних персонального комп’ютера, необхідний для збереження та обробки даних під час виконання програми

– X2 – час, який витрачається на виконання коду
– X3 – потенційний об’єм програмного коду, який необхідно створити безпосередньо розробнику

6.2.2 Кількісна оцінка параметрів

Гірші, середні і кращі значення параметрів вибираються на основі вимог замовника й умов, що характеризують експлуатацію програмного продукту як показано у нище.

Опис параметру	Умовні позначення	Одиниці виміру	Значення параметра		
			гірші	середні	кращі
Об’єм пам’яті для збереження даних	X1	Мб	512	156	128
Середній час обробки запиту	X2	с	5	3	1
Потенційний об’єм програмного коду	X3	строк коду	190000	100000	80000

Табл. 6.2 – Основні параметри програмного продукту

За даними таблиці будуємо графічні характеристики параметрів:

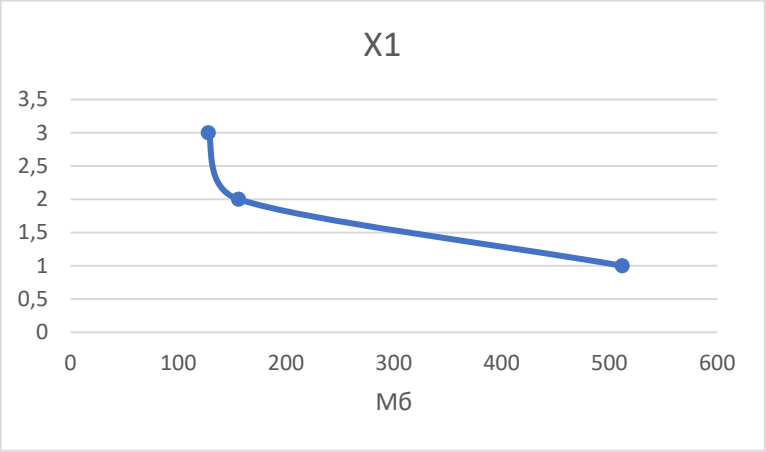

Рис. 6.2 – Об’єм пам’яті для збереження даних

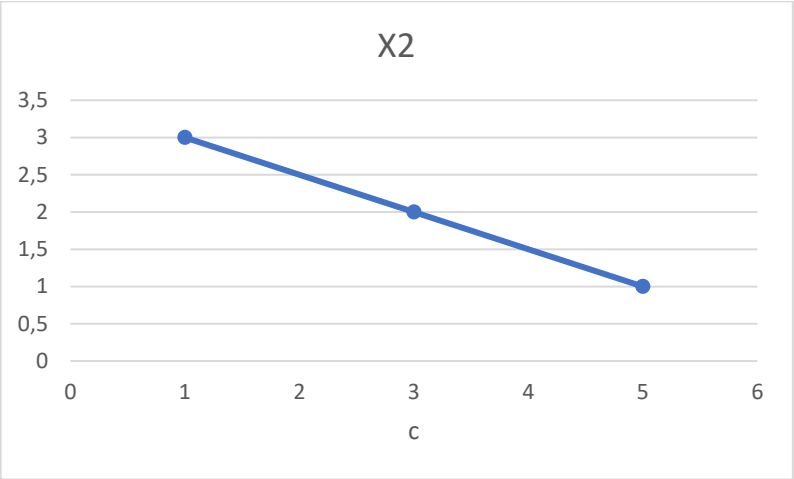

Рис. 6.3 – Середній час обробки запиту

Рис. 6.4 – Потенційний об’єм програмного коду

6.2.3 Аналіз експертного оцінювання параметрів

Після детального обговорення й аналізу кожний експерт оцінює ступінь важливості кожного параметру для конкретно поставленої цілі – розробка програмного продукту, який має найбільш зручний інтерфейс та зрозумілу взаємодію з користувачем. [25]

Значимість кожного параметра визначається методом попарного порівняння. Оцінку проводить експертна комісія із 4 людей. Визначення коефіцієнтів значимості передбачає:

1) визначення рівня значимості параметра шляхом присвоєння різних рангів

2) перевірку придатності експертних оцінок для подальшого використання
3) визначення оцінки попарного пріоритету параметрів
4) обробку результатів та визначення коефіцієнту значимості

Параметр	Ранг параметра за оцінкою експерта				Сума рангів	Відхилення, Δ_i	Δ_i^2
	1	2	3	4			
X1	1	1	1	1	4	4	16
X2	3	2	3	2	10	-2	4
X3	2	3	2	3	10	-2	4
Разом	6	6	6	6	24	0	24

Табл. 6.3 – Основні параметри програмного продукту

Для перевірки ступеню достовірності експертних оцінок, визначимо наступні параметри:

а) сума рангів кожного з параметрів і загальна сума рангів:

$$R_i = \sum_{j=1}^{N} * r_{ij} = 24,$$

де r_{ij} – ранг *i*-го параметра, визначений *j*-м експертом;

N – число експертів.

б) середня сума рангів *T*:

$$T = \frac{1}{n} R_i = 8.$$

в) відхилення суми рангів кожного параметра від середньої суми рангів:

$$\Delta_i = R_i - T.$$

г) загальна сума квадратів відхилення:

$$S = \sum_{i=1}^{n} * \Delta_i^2 = 24.$$

д) коефіцієнт узгодженості (конкордації):

$$W = \frac{12S}{N^2(n^3 - n)} = \frac{12 \cdot 24}{4^2(3^3 - 3)} = 0{,}75 > W_k = 0{,}67.$$

Ранжирування можна вважати достовірним, тому що знайдений коефіцієнт узгодженості перевищує нормативний, який рівний 0,75. Скориставшись результатами ранжирування, проведемо попарне порівняння всіх параметрів і результати занесемо у таблицю 6.4. Числове значення, що визначає ступінь переваги *i*-го параметра над *j*-тим, a_{ij} визначається за формулою:

$$a_{ij} = \{1{,}5 \;\; x_i > x_j;\; 1{,}0 \;\; x_i = x_j;\; 0{,}5 \;\; x_i < x_j\}.$$

Параметри	Експерти				Підсумкова оцінка	Числове значення коефіцієнтів переваги
	1	2	3	4		
X1,X2	<	<	<	<	<	0,5
X1,X3	<	<	<	<	<	0,5
X2,X3	>	<	>	=	>	1,0

Табл. 6.4 – Результати ранжування параметрів

З отриманих числових оцінок переваги складемо матрицю $A = \|a_{ij}\|$. Для кожного параметра розрахунок вагомості K_{B_i} проводиться за наступною формулою:

$$K_{B_i} = \frac{b_i}{\sum_{i=1}^{n} * b_i},$$

де $b_i = \sum_{j=1}^{N} a_{ij}$ – вагомість *i*-го параметра за результатами оцінок всіх експертів;

a_{ij} –коефіцієнт переваги *i*-го на *j*-тим параметром.

Відносні оцінки розраховуються декілька разів доти, поки наступні значення не будуть незначно відрізнятися від попередніх (менше 2%). На другому і наступних кроках відносні оцінки розраховуються за наступною формулою:

$$K_{B_i} = \frac{b_i'}{\sum_{i=1}^{n} * b_i'},$$

де $b_i' = \sum_{j=1}^{N} * a_{ij} b_j$.

Як видно з таблиці 6.5, різниця значень коефіцієнтів вагомості після другої ітерації не перевищує 2%, тому додаткові ітерації не потрібні.

i	j			Перша ітерація		Друга ітерація	
	X1	X2	X3	B_i	K_{B_i}	B_i^1	$K_{B_i}^1$
X1	1,0	0,5	0,5	2	0,222	18	0,222
X2	1,5	1,0	1,0	3,5	0,389	31,5	0,389
X3	1,5	1,0	1,0	3,5	0,389	31,5	0,389
Разом				9	1	81	1

Табл. 6.5 – Розрахунок вагомості параметрів

6.3 Аналіз рівня якості варіантів реалізації функцій

Рівень якості кожного варіанту виконання основних функцій визначається окремо.

Абсолютні значення параметрів X1 – об’єм пам’яті для збереження даних, та X2 – середній час обробки запиту, відповідають технічним вимогам умов функціонування даного ПП.

Абсолютне значення параметра X3 – кількість строк коду, обрано не максимальним і це значення відповідає варіанту б) 100000 або в) 80000 с.

Коефіцієнт технічного рівня якості для кожного варіанта реалізації ПП розраховується за формулою:

$$K_{TP} = \sum_{i=1}^{n} * K_{B_i} B_i,$$

де n – кількість параметрів;

K_{B_i} – коефіцієнт вагомості і–го параметра;

B_i – оцінка i–го параметра в балах.

Розрахунок показників рівня якості представлено відповідно в таблиці 6.6.

Основні функції	Варіант реалізації	Абсолютне значення параметра	Бальна оцінка параметра	Коефіцієнт вагомості параметра	Коефіцієнт рівня якості
F_1	А	100000	5	0,222	1,11
F_2	А	5	10	0,389	3,89
	Б	3	5	0,222	1,11
	В	1	1	0,145	0,145
F_3	А	80000	10	0,389	3,89

Табл. 6.6 – Розрахунок показників якості

За цими даними визначаємо рівень якості кожного з варіантів:

- F_1(А) – F_2(А) – F_3(А) = 1,11 + 3,89 + 3,89 = 8,89
- F_1(А) – F_2(Б) – F_3(А) = 1,11 + 1,11 + 3,89 = 6,11
- F_1(А) – F_2(В) – F_3(А) = 1,11 + 0,145 + 3,89 = 5,145

Отже, найкращим є перший варіант, для якого коефіцієнт технічного рівня має найбільше значення.

6.4 Економічний аналіз варіантів розробки програмного продукту

Для визначення вартості розробки програмного продукту спочатку проведемо розрахунок трудомісткості. [25]

Всі варіанти включають в себе два окремих завдання:

1. Розробка проекту програмного продукту

2. Розробка алгоритму збору даних

Завдання 1 за ступенем новизни відноситься до групи А, завдання 2 – до групи Б. За складністю алгоритми, які використовуються в завданні 1 належать до групи 1; а в завданні 2 – до групи 3.

Для реалізації завдання 1 використовується інформація з довідників, а завдання 2 використовує інформацію у вигляді даних. Проведемо розрахунок норм часу на розробку та програмування для кожного з завдань. Загальна трудомісткість обчислюється за формулою.

$$T_O = T_P \cdot K_П \cdot K_{СК} \cdot K_М \cdot K_{СТ} \cdot K_{СТ.М}$$

де T_P – трудомісткість розробки програмного продукту;

$K_П$ – поправочний коефіцієнт;

$K_{СК}$ – коефіцієнт на складність вхідної інформації;

$K_М$ – коефіцієнт рівня мови програмування;

$K_{СТ}$ – коефіцієнт використання стандартних модулів і прикладних програм;

$K_{СТ.М}$ – коефіцієнт стандартного математичного забезпечення.

Для першого завдання, виходячи із норм часу для завдань розрахункового характеру степеню новизни А та групи складності алгоритму 1, трудомісткість дорівнює: $T_P = 90$ людино-днів. Поправочний коефіцієнт вхідної інформації для першого завдання: $K_П = 1{,}4$. Поправочний коефіцієнт, який враховує складність контролю вхідної та вихідної інформації рівний $K_{СК} = 1$. Оскільки при розробці першого завдання використовуються стандартні модулі, врахуємо це за допомогою коефіцієнта $K_{СТ} = 0{,}8$. Тоді загальна трудомісткість програмування першого завдання дорівнює:

$$T_1 = 90 \cdot 1.4 \cdot 0.8 = 100{,}8 \text{ людино-дні.}$$

Проведемо аналогічні розрахунки для другого завдання, в якому використовується алгоритм третьої групи складності зі ступенем новизни Б, тобто $\mathrm{Т_Р} = 27$ людино-днів, $\mathrm{К_П} = 0{,}9$, $\mathrm{К_{СК}} = 1$, $\mathrm{К_{СТ}} = 0{,}8$:

$$\mathrm{Т_2} = 27 \cdot 0.9 \cdot 0.8 = 19{,}44 \text{ людино-дні.}$$

Оскільки загальна трудомісткість усіх варіантів реалізації збігаються, їх можна об’єднати в одну групу.

Загальна трудомісткість складає:

$$\mathrm{Т_0} = (100{,}8 + 19{,}44) \cdot 8 = 961{,}92 \text{ людино-годин.}$$

В розробці беруть участь програміст з окладом 25000 грн., один аналітик в області даних з окладом 25000 грн, та один інженер даних з окладом 20000 грн. Визначимо середню зарплату за годину за формулою:

$$\mathrm{С_Ч} = \frac{M}{T_m \cdot t} \text{ грн.,}$$

де M – місячний оклад працівників;

T_m – кількість робочих днів на місяць;

t – кількість робочих годин в день.

$$\mathrm{С_Ч} = \frac{25000 + 25000 + 20000}{3 \cdot 20 \cdot 8} = 145{,}83 \text{ грн.}$$

Тоді, розрахуємо заробітну плату за формулою:

$$\mathrm{С_{ЗП}} = \mathrm{С_Ч} \cdot T_i \cdot \mathrm{К_Д},$$

де $\mathrm{С_Ч}$ – величина погодинної оплати праці програміста;

T_i – трудомісткість відповідного завдання;

$\mathrm{К_Д}$ – норматив, який враховує додаткову заробітну плату.

Зарплата розробників за варіантами становить:

$$С_{ЗП} = 145,83 \cdot 351,36 \cdot 1,2 = 168332 \text{ грн.}$$

Відрахування на соціальний внесок становить 22%:

$$С_{СВ} = С_{ЗП} \cdot 0,22 = 168332 \cdot 0,22 = 37033 \text{ грн.}$$

Тепер визначимо витрати на оплату однієї машино-години. Оскільки одна ЕОМ обслуговується одним інженером апаратного забезпечення з окладом 15000 грн. та коефіцієнтом зайнятості $К_З = 0,2$ то для однієї машини отримаємо:

$$С_Г = 12 \cdot М \cdot К_З = 12 \cdot 15000 \cdot 0,2 = 36000 \text{ грн.}$$

З урахуванням додаткової заробітної плати:

$$С_{ЗП} = С_Г \cdot (1 + К_З) = 36000 \cdot (1 + 0,2) = 43200 \text{ грн.}$$

Відрахування на соціальний внесок становить 22%:

$$С_{СВ} = С_{ЗП} \cdot 0,22 = 43200 \cdot 0,22 = 9504 \text{ грн.}$$

Амортизаційні відрахування розраховуємо за формулою при амортизації 25% та вартості ЕОМ – 35 000 грн.:

$$С_А = К_{ТМ} \cdot К_А \cdot Ц_{ПР} = 1,15 \cdot 0,25 \cdot 35000 = 10062,5 \text{ грн.}$$

де $К_{ТМ}$ – коефіцієнт, який враховує витрати на транспортування та монтаж приладу у користувача;

$К_А$ – річна норма амортизації;

$Ц_{ПР}$ – договірна ціна приладу.

Витрати на ремонт та профілактику розраховуємо за формулою:

$$С_Р = К_{ТМ} \cdot К_Р \cdot Ц_{ПР} = 1,15 \cdot 0,05 \cdot 35000 = 2012,5 \text{ грн.}$$

де $К_Р$– відсоток витрат на поточні ремонти.

Ефективний годинний фонд часу ПК за рік розраховуємо за формулою:

$$Т_{ЕФ} = (Д_К - Д_В - Д_С - Д_Р) \cdot t \cdot К_В,$$

$$Т_{ЕФ} = (365 - 104 - 12 - 16) \cdot 8 \cdot 0{,}9 = 1\,667{,}6 \text{ год.}$$

де $Д_К$ – календарна кількість днів у році;

$Д_В, Д_С$ – відповідно кількість вихідних та святкових днів;

$Д_Р$ – кількість днів планових ремонтів устаткування;

t – кількість робочих годин в день;

$К_В$ – коефіцієнт використання приладу у часі протягом зміни.

Витрати на оплату електроенергії розраховуємо за формулою:

$$С_{ЕЛ} = Т_{ЕФ} \cdot N_C \cdot К_З \cdot Ц_{ЕЛ} = 1\,677{,}6 \cdot 0{,}65 \cdot 0{,}2 \cdot 3{,}6 = 758{,}12 \text{ грн.}$$

де N_C – середньо-споживча потужність приладу;

$К_З$ – коефіцієнтом зайнятості приладу;

$Ц_{ЕЛ}$ – тариф за 1 КВт-годин електроенергії.

Накладні витрати розраховуємо за формулою:

$$С_Н = Ц_{ПР} \cdot 0{,}67 = 35\,000 \cdot 0{,}67 = 23450 \text{ грн.}$$

Тоді, річні експлуатаційні витрати будуть складати:

$$С_{ЕК} = С_{ЗП} + С_{СВ} + С_А + С_Р + С_{ЕЛ} + С_Н,$$

$$С_{ЕК} = 43200 + 9504 + 10062{,}5 + 2012{,}5 + 758{,}12 + 23450 = 88987{,}12 \text{ грн.}$$

Собівартість однієї машино-години ЕОМ дорівнюватиме:

$$С_{МГ} = \frac{С_{ЕК}}{Т_{ЕФ}} = \frac{88987{,}12}{1\,677{,}6} = 53{,}04 \text{ грн/год.}$$

Оскільки в даному випадку всі роботи, які пов‘язані з розробкою програмного продукту ведуться на ЕОМ, витрати на оплату машинного часу складають:

$$С_М = С_{МГ} \cdot Т = 53{,}04 \cdot 961{,}92 = 51020 \text{ грн.}$$

Накладні витрати складають 67% від заробітної плати:

$$С_Н = С_{ЗП} \cdot 0{,}67 = 43200 \cdot 0{,}67 = 28944 \text{ грн.}$$

Отже, вартість розробки програмного продукту за варіантами становить:

$$С_{ПП} = С_{ЗП} + С_{СВ} + С_{М} + С_{Н},$$

$$С_{ПП} = 43200 + 9504 + 51020 + 28944 = 132668 \text{ грн.}$$

Розрахуємо коефіцієнт техніко-економічного рівня за формулою:

$$К_{ТЕР} = \frac{К_{К}}{С_{ПП}} = \frac{8{,}89}{132668} = 6{,}7 \cdot 10^{-5}$$

6.5 Висновки до розділу

У даному розділі в результаті виконання економічного розділу були систематизовані і закріплені теоретичні знання в галузі економіки та організації виробництва використанням їх для техніко-економічного обґрунтування розробки методом функціонально-вартісного аналізу. Для цього використовувався методичний матеріал [25]

На основі даних про зміст основних функцій, які повинен реалізувати програмний продукт, були визначені чотири найбільш перспективні варіанти реалізації продукту. Найбільш ефективним виявився третій варіант реалізації функцій ПП, який дає максимальну величину коефіцієнта техніко-економічного рівня, вартість витрат для нього становить $С_{ПП} = 132668$ грн.

Цей варіант передбачає:

- застосування мови програмування Java;
- використання фреймворку Spring WebFlux
- зберігання вихідних даних у базі даних MongoDB;
- використання ілюстративних матеріалів;
- найвищу точність результатів обробки даних.

ВИСНОВКИ

В ході проведеної роботи були розглянуті основні концепції реактивних інструментів на мові програмування Java, були розглянуті та проаналізовані популярні фреймворки які надають реактивні інструменти для побудови інформаційних систем та сервісів, було детально розглянуто функціонал, який надає Spring WebFlux, були описані його особливості реалізації, надані приклади використання та розгорнута хмарна інфраструктура, яка використовує побудований сервіс та базу даних MongoDB.

Реактивний підхід до програмування допомагає розробникам підвищити продуктивність програми, оскільки цей підхід дозволяє розроблюваній системі набагато швидше обробляти великі обсяги даних. Основні властивості реактивного підходу:

- більша гнучкість
- стійкість і масштабованість
- більша пристосованість до помилок при обробці
- ефективність використання ресурсів
- менша затримка виконання при обробці

Код, написаний при реактивному підході є компактним і лаконічним, що полегшує читання та масштабування. Таким чином, зміни, модифікації та оновлення можна зробити спрощеним способом. Ці особливості надають цьому підходу додаткову перевагу – менші витрати часу на розробку.

Хоча розуміння концепції та вивчення реактивного підходу може зайняти деякий час і вимагає багато сумлінної роботи, вивчення тощо, парадигма є надзвичайно корисною сьогодні. Вона має ряд переваг, які полегшують роботу розробників, покращує продуктивність додатків і, крім того, покращує досвід користувачів.

Незалежно від того, стане архітектура мікросервісів у майбутньому бажаним стилем розробників, це, безсумнівно, потужна ідея, яка пропонує серйозні переваги для проектування та впровадження корпоративних додатків. Багато розробників та організацій, ніколи не використовуючи назву і навіть не позначаючи свою практику як SOA, використовували підхід до використання API, які можна було б класифікувати як мікросервіси.

В результаті були визначені поняття реактивного програмування, особливості застосування, його становлення, був наданий перелік правил, які характеризують реактивні програми та системи, на базовому рівні були описані шаблони взаємодії в реактивних програмах між об’єктами, які несуть в собі потоки даних, і тими, які ці потоки даних споживають, були наведені приклади реалізацій і використання існуючого API в Java і продемонстровані можливості низькорівневих реалізацій інструментів мови програмування.

Були розглянути наявні рішення реактивних систем на мові Java, описані їхні особливості, проведено порівняння.

Було досліджено застосування фреймворку Spring в реактивних системах, був розглянути основний функціонал, на якому базується реалізація Spring Boot в реактивних системах, були розглянуті особливості реалізації WebFlux, та його компонентів, наведені приклади реалізацій та підходів, було розглянуто взаємодію з реляційними базами даних проблеми та переваги міграцій даних при користуванні сторонніми сховищами.

Кодову базу розробленого сервісу можна знайти за посиланням: https://github.com/sergey-judi/reactive-spring

Хмарна інфраструктура сервісу розгорнута за посиланням: https://sergey-judi-reactive-spring.herokuapp.com/

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Contributors to Wikimedia projects. Reactive programming - Wikipedia. *Wikipedia, the free encyclopedia.* URL: https://en.wikipedia.org/wiki/Reactive_programming (дата звернення: 07.05.2022).

2. The reactive manifesto. *The Reactive Manifesto.* URL: https://www.reactivemanifesto.org/ (дата звернення: 07.05.2022).

3. Race conditions and deadlocks - Visual Basic. *Developer tools, technical documentation and coding examples | Microsoft Docs.* URL: https://docs.microsoft.com/en-us/troubleshoot/developer/visualstudio/visual-basic/language-compilers/race-conditions-deadlocks (дата звернення: 07.05.2022).

4. Наблюдатель. *Refactoring and Design Patterns.* URL: https://refactoring.guru/ru/design-patterns/observer (дата звернення: 24.05.2022).

5. Reactive Streams URL: https://www.reactive-streams.org/ (дата звернення: 15.05.2022).

6. What Is an API (Application Programming Interface)? Definition and Examples. Talend - A Leader in Data Integration & Data Integrity. URL: https://www.talend.com/resources/what-is-an-api/ (дата звернення: 15.05.2022).

7. Akka: build concurrent, distributed, and resilient message-driven applications for Java and Scala | Akka. Akka: build concurrent, distributed, and resilient message-driven applications for Java and Scala | Akka. URL: https://akka.io/ (дата звернення: 15.05.2022).

8. ReactiveX. ReactiveX. URL: https://reactivex.io/ (дата звернення: 15.05.2022).

9. Eclipse Vert.x. Eclipse Vert.x. URL: https://vertx.io/ (дата звернення: 15.05.2022).

10. Project Reactor. Project Reactor. URL: https://projectreactor.io/ (дата звернення: 15.05.2022).

11. Flux (reactor-core 3.4.18). *Project Reactor.* URL: https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html (дата звернення: 30.05.2022).

12. Mono (reactor-core 3.4.18). *Project Reactor.* URL: https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Mono.html (дата звернення: 30.05.2022).

13. Web MVC framework. Spring | *Home*. URL: https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/mvc.html (дата звернення: 30.05.2022).

14. Web on Reactive Stack. Spring | *Home*. URL: https://docs.spring.io/spring-framework/docs/current/reference/html/web-reactive.html (дата звернення: 24.05.2022).

15. Project Reactor. *Project Reactor.* URL: https://projectreactor.io/ (дата звернення: 24.05.2022).

16. Contributors to Wikimedia projects. Pivotal Software - Wikipedia. *Wikipedia, the free encyclopedia.* URL: https://en.wikipedia.org/wiki/Pivotal_Software (дата звернення: 24.05.2022).

17. Behler M. JDBC - a short guide. *Learn more about Java, no matter your skill level, anytime and anywhere you want | Marco Behler GmbH.* URL: https://www.marcobehler.com/guides/jdbc (дата звернення: 24.05.2022).

18. Accessing data with R2DBC. *Spring | Home.* URL: https://spring.io/guides/gs/accessing-data-r2dbc/ (дата звернення: 24.05.2022).

19. Что такое agile? | Atlassian. *Atlassian.* URL: https://www.atlassian.com/ru/agile (дата звернення: 30.05.2022).

20. Was ist DevOps? Eine Erläuterung | Microsoft Azure. *Cloud Computing Services | Microsoft Azure.* URL: https://azure.microsoft.com/de-de/overview/what-is-devops/ (дата звернення: 30.05.2022).

21. Bonér J. Reactive Microservices Architecture. Design Principles for Distributed Systems : book. Boston : O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, 2016. 54 p.

22. MongoDB Java Reactive Streams MongoDB Drivers. *MongoDB: The Application Data Platform | MongoDB.* URL: https://www.mongodb.com/docs/drivers/reactive-streams/ (дата звернення: 29.05.2022).

23. Cloud Application Platform | Heroku. *Cloud Application Platform | Heroku.* URL: https://www.heroku.com/ (дата звернення: 29.05.2022).

24. MongoDB Cloud. *MongoDB.* URL: https://www.mongodb.com/cloud (дата звернення: 29.05.2022).

25. Пашін В. П. Методичні вказівки до виконання економіко організаційного розділу дипломних проектів (робіт) бакалаврів і спеціалістів для студентів інституту прикладного системного аналізу / В. П. Пашін, В. В. Романов, Н. В. Єгорова. // НТУУ “КПІ”. 2011. С. 118.

ДОДАТОК А

QuoteDataLoader.java

```
package com.iasa.kpi.config;

import com.iasa.kpi.config.properties.ApplicationProperties;
import com.iasa.kpi.repository.QuoteReactiveRepository;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.boot.context.event.ApplicationStartedEvent;
import org.springframework.context.ApplicationListener;
import org.springframework.stereotype.Component;
import reactor.core.publisher.Flux;
import com.iasa.kpi.model.Quote;

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.List;
import java.util.concurrent.atomic.AtomicLong;
import java.util.function.Supplier;

import static java.util.Objects.requireNonNull;

@Slf4j
@Component
@RequiredArgsConstructor
public class QuoteDataLoader implements
ApplicationListener<ApplicationStartedEvent> {

  private final ApplicationProperties applicationProperties;
  private final QuoteReactiveRepository quoteReactiveRepository;

  @Override
  public void onApplicationEvent(ApplicationStartedEvent event) {
    log.info("Found {} entities in db",
quoteReactiveRepository.count().block());

    if (quoteReactiveRepository.findAll().count().block().equals(0L)) {
      Supplier<String> idGenerator = getIdGenerator();

      BufferedReader dataReader = new BufferedReader(new InputStreamReader(

requireNonNull(getClass().getClassLoader().getResourceAsStream(application
Properties.getDataFile()))));

      List<Quote> quotes = dataReader.lines()
          .filter(line -> !line.trim().isEmpty())
          .map(line -> {
            String id = idGenerator.get();
            String book = String.format("book-%s", id);
            return new Quote(id, book, line);
          })
          .toList();

      Flux.fromIterable(quotes)
          .flatMap(quoteReactiveRepository::save)
```

```
          .subscribe();

      log.info("Successfully    loaded    {}    entities    into    db",
quoteReactiveRepository.count().block());
    }
  }

  private Supplier<String> getIdGenerator() {
    AtomicLong id = new AtomicLong();
    return () -> String.format("%05d", id.incrementAndGet());
  }

}
```

QuoteDto.java

```
package com.iasa.kpi.controller.dto;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import javax.validation.constraints.NotBlank;

@Data
@NoArgsConstructor
@AllArgsConstructor
public class QuoteDto {

  @NotBlank(groups = OnCreate.class)
  String id;

  @NotBlank(groups = OnCreate.class)
  String book;

  @NotBlank(groups = OnCreate.class)
  String content;

  public interface OnCreate {}

}
```

ErrorHandler.java

```
package com.iasa.kpi.controller;

import com.iasa.kpi.exception.EntityAlreadyExistsException;
import com.iasa.kpi.exception.model.ErrorCode;
import com.iasa.kpi.exception.model.ErrorResponse;
import lombok.extern.slf4j.Slf4j;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.MethodArgumentNotValidException;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestControllerAdvice;
import org.springframework.web.bind.support.WebExchangeBindException;
import com.iasa.kpi.exception.EntityNotFoundException;
```

```
import java.time.Instant;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

import static org.springframework.http.HttpStatus.BAD_REQUEST;
import static org.springframework.http.HttpStatus.INTERNAL_SERVER_ERROR;
import static org.springframework.http.HttpStatus.NOT_FOUND;

@Slf4j
@RestControllerAdvice(assignableTypes = {QuoteBlockingController.class,
QuoteReactiveController.class})
public class ErrorHandler {

  @ExceptionHandler(EntityNotFoundException.class)
  @ResponseStatus(NOT_FOUND)
  public                                                    ErrorResponse
handleEntityNotFoundException(EntityNotFoundException ex) {
    log.error("Handling    entity    not    found    exception:    [{}]",
ex.getMessage());
    return                  buildErrorResponse(ErrorCode.ENTITY_NOT_FOUND,
ex.getMessage());
  }

  @ExceptionHandler(EntityAlreadyExistsException.class)
  @ResponseStatus(BAD_REQUEST)
  public                                                    ErrorResponse
handleEntityNotFoundException(EntityAlreadyExistsException ex) {
    log.error("Handling   entity   already   exists   exception:   [{}]",
ex.getMessage());
    return             buildErrorResponse(ErrorCode.ENTITY_ALREADY_EXISTS,
ex.getMessage());
  }

  @ExceptionHandler(WebExchangeBindException.class)
  @ResponseStatus(BAD_REQUEST)
  public                                                    ErrorResponse
handleMethodArgumentNotValidException(WebExchangeBindException ex) {
    String                          errorMessage                          =
getValidationErrorMessage(ex.getBindingResult());

    log.error("Handling validation exception: [{}]", errorMessage);
    return buildErrorResponse(ErrorCode.INVALID_PARAMS, errorMessage);
  }

  @ExceptionHandler(MethodArgumentNotValidException.class)
  @ResponseStatus(BAD_REQUEST)
  public                                                    ErrorResponse
handleMethodArgumentNotValidException(MethodArgumentNotValidException  ex)
{
    String                          errorMessage                          =
getValidationErrorMessage(ex.getBindingResult());

    log.error("Handling validation exception: [{}]", errorMessage);
    return buildErrorResponse(ErrorCode.INVALID_PARAMS, errorMessage);
  }

  @ExceptionHandler(Exception.class)
  @ResponseStatus(INTERNAL_SERVER_ERROR)
  public ErrorResponse handleGeneralException(Exception ex) {
```

```
    log.error("Handling general exception: [{}]", ex.getMessage());
    System.out.println(ex.getClass());
    return buildErrorResponse(ErrorCode.GENERAL, ex.getMessage());
  }

  private String getValidationErrorMessage(BindingResult bindingResult) {
    Map<String, List<String>> validation =
bindingResult.getFieldErrors().stream()
        .map(error -> Map.entry(error.getField(),
error.getDefaultMessage()))
        .collect(Collectors.groupingBy(
            Map.Entry::getKey,
            Collectors.mapping(
                Map.Entry::getValue,
                Collectors.toList()
            )));

    List<String> errors = validation.entrySet()
        .stream()
        .map(entry -> entry.getKey() + ": " + String.join(", ",
entry.getValue()))
        .sorted()
        .toList();

    return String.join(", ", errors);
  }

  private ErrorResponse buildErrorResponse(String code, String description)
{
    return ErrorResponse.builder()
        .code(code)
        .message(description)
        .at(Instant.now())
        .build();
  }

}
```

Quote.java

```
package com.iasa.kpi.model;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.springframework.data.mongodb.core.mapping.Document;

@Data
@Document
@NoArgsConstructor
@AllArgsConstructor
public class Quote {

  private String id;
  private String book;
  private String content;

}
```

QuoteBlockingRepository.java

```
package com.iasa.kpi.repository;

import org.springframework.data.domain.Pageable;
import org.springframework.data.mongodb.repository.MongoRepository;
import org.springframework.stereotype.Repository;
import com.iasa.kpi.model.Quote;

import java.util.List;

@Repository
public interface QuoteBlockingRepository extends MongoRepository<Quote,
String> {

  List<Quote> findAllByIdNotNullOrderByIdAsc(Pageable page);

}
```

QuoteReactiveRepository.java

```
package com.iasa.kpi.repository;

import org.springframework.data.domain.Pageable;
import
org.springframework.data.mongodb.repository.ReactiveMongoRepository;
import org.springframework.stereotype.Repository;
import reactor.core.publisher.Flux;
import com.iasa.kpi.model.Quote;

@Repository
public interface QuoteReactiveRepository extends
ReactiveMongoRepository<Quote, String> {

  Flux<Quote> findAllByIdNotNullOrderByIdAsc(Pageable page);

}
```

QuoteConverter.java

```
package com.iasa.kpi.converter;

import org.springframework.stereotype.Component;
import com.iasa.kpi.controller.dto.QuoteDto;
import com.iasa.kpi.model.Quote;

@Component
public class QuoteConverter implements Converter<QuoteDto, Quote> {

  @Override
  public QuoteDto toDto(Quote model) {
    return new QuoteDto(model.getId(), model.getBook(),
model.getContent());
  }

  @Override
  public Quote fromDto(QuoteDto dto) {
    return new Quote(dto.getId(), dto.getBook(), dto.getContent());
```

```
  }

}
```

QuoteBlockingController.java

```
package com.iasa.kpi.controller;

import com.iasa.kpi.controller.dto.QuoteDto;
import com.iasa.kpi.converter.QuoteConverter;
import com.iasa.kpi.service.blocking.QuoteBlockingService;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.validation.annotation.Validated;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestController;

import javax.validation.groups.Default;
import java.util.List;

import static org.springframework.http.HttpStatus.CREATED;

@Slf4j
@RestController
@RequiredArgsConstructor
@RequestMapping("/blocking/quotes")
public class QuoteBlockingController {

  private final QuoteBlockingService quoteService;
  private final QuoteConverter quoteConverter;

  @GetMapping
  public List<QuoteDto> getAll() {
    log.info("Retrieving all quotes");
    List<QuoteDto> quoteDtos = quoteService.getAll()
        .stream()
        .map(quoteConverter::toDto)
        .toList();
    log.info("Successfully retrieved all quotes");
    return quoteDtos;
  }

  @GetMapping("/paged")
  public List<QuoteDto> getAll(@RequestParam("page") int page,
                               @RequestParam("size") int size) {
    log.info("Retrieving all quotes by page [{}] and size [{}]", page,
size);
    List<QuoteDto> quoteDtos = quoteService.getAllByPage(page, size)
        .stream()
```

```
        .map(quoteConverter::toDto)
        .toList();
    log.info("Successfully retrieved by page [{}] and size [{}]", page,
size);
    return quoteDtos;
  }

  @GetMapping("/{id}")
  public QuoteDto getById(@PathVariable String id) {
    log.info("Retrieving quote by id [{}]", id);
    QuoteDto retrieved = quoteConverter.toDto(quoteService.get(id));
    log.info("Successfully retrieved quote [{}]", retrieved);
    return retrieved;
  }

  @PostMapping
  @ResponseStatus(CREATED)
  public            QuoteDto            create(@Validated({Default.class,
QuoteDto.OnCreate.class}) @RequestBody QuoteDto quoteDto) {
    log.info("Creating quote [{}]", quoteDto);
    QuoteDto created = quoteConverter.toDto(
        quoteService.create(quoteConverter.fromDto(quoteDto)));
    log.info("Successfully created quote [{}]", created);
    return created;
  }

  @PutMapping("/{id}")
  public QuoteDto update(@PathVariable String id,
                         @Validated @RequestBody QuoteDto quoteDto) {
    log.info("Updating quote by id [{}], [{}]", id, quoteDto);
    QuoteDto updated = quoteConverter.toDto(
        quoteService.update(id, quoteConverter.fromDto(quoteDto)));
    log.info("Successfully updated quote [{}]", updated);
    return updated;
  }

  @DeleteMapping("/{id}")
  public void deleteById(@PathVariable String id) {
    log.info("Deleting quote by id [{}]", id);
    quoteService.delete(id);
    log.info("Successfully deleted quote by id [{}]", id);
  }

}
```

QuoteReactiveController.java

```
package com.iasa.kpi.controller;

import com.iasa.kpi.controller.dto.QuoteDto;
import com.iasa.kpi.converter.QuoteConverter;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.validation.annotation.Validated;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
```

```
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestController;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;
import com.iasa.kpi.service.reactive.QuoteReactiveService;

import javax.validation.groups.Default;

import static org.springframework.http.HttpStatus.CREATED;

@Slf4j
@RestController
@RequiredArgsConstructor
@RequestMapping("/reactive/quotes")
public class QuoteReactiveController {

  private final QuoteReactiveService quoteService;
  private final QuoteConverter quoteConverter;

  @GetMapping
  public Flux<QuoteDto> getAll() {
    log.info("Retrieving all quotes");
    return quoteService.getAll()
        .map(quoteConverter::toDto)
        .doOnComplete(() -> log.info("Successfully retrieved all quotes"));
  }

  @GetMapping("/paged")
  public Flux<QuoteDto> getAll(@RequestParam("page") int page,
                               @RequestParam("size") int size) {
    log.info("Retrieving all quotes by page [{}] and size [{}]", page,
size);
    return quoteService.getAllByPage(page, size)
        .map(quoteConverter::toDto)
        .doOnComplete(() -> log.info("Successfully retrieved by page [{}]
and size [{}]", page, size));
  }

  @GetMapping("/{id}")
  public Mono<QuoteDto> getById(@PathVariable String id) {
    log.info("Retrieving quote by id [{}]", id);
    return quoteService.get(id)
        .map(quoteConverter::toDto)
        .doOnSuccess(retrieved -> log.info("Successfully retrieved quote
[{}]", retrieved));
  }

  @PostMapping
  @ResponseStatus(CREATED)
  public          Mono<QuoteDto>          create(@Validated({Default.class,
QuoteDto.OnCreate.class}) @RequestBody QuoteDto quoteDto) {
    log.info("Creating quote [{}]", quoteDto);
    return quoteService.create(quoteConverter.fromDto(quoteDto))
        .map(quoteConverter::toDto)
        .doOnSuccess(created -> log.info("Successfully created quote [{}]",
created));
  }
```

```
  @PutMapping("/{id}")
  public Mono<QuoteDto> update(@PathVariable String id,
                               @Validated @RequestBody QuoteDto quoteDto) {
    log.info("Updating quote by id [{}], [{}]", id, quoteDto);
    return quoteService.update(id, quoteConverter.fromDto(quoteDto))
        .map(quoteConverter::toDto)
        .doOnSuccess(updated -> log.info("Successfully updated quote [{}]",
updated));
  }

  @DeleteMapping("/{id}")
  public Mono<Void> deleteById(@PathVariable String id) {
    log.info("Deleting quote by id [{}]", id);
    return quoteService.delete(id)
        .doOnSuccess(deleted -> log.info("Successfully deleted quote by id
[{}]", id));
  }

}
```