МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ

КАФЕДРА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ АВТОМАТИЗОВАНИХ СИСТЕМ

Кваліфікаційна робота

другий (магістерський)
(рівень вищої освіти)

на тему **Використання реактивного програмування для оптимізації ресурсів комп'ютера при пікових навантаженнях**

Виконав: студент 2 курсу, групи 8.1219-пзс
спеціальності 121 Інженерія програмного забезпечення
(код і назва спеціальності)
освітньої програми Інженерія програмного забезпечення
(код і назва освітньої програми)
А. І. Вуколова
(ініціали та прізвище)

Керівник доцент, к. т. н. Н. П. Полякова
(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)
Рецензент директор ТОВ «Дісітел»
П.О. Лютий
(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)

Запоріжжя
2020

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Інженерний навчально-науковий інститут
Кафедра _____ програмного забезпечення автоматизованих систем
Рівень вищої освіти _____другий (магістерський)_____
Спеціальність ___121_Інженерія програмного забезпечення_____
(код та назва)
Освітня програма ___Інженерія програмного забезпечення
(код та назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри___ _____В.Г. Вербицький
"_01_" __вересня_____2020_року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Вуколова Анастасія Ігорівна
(прізвище, ім'я, по батькові)

1.Тема роботи ___Використання реактивного програмування для оптимізації ресурсів комп'ютера при пікових навантаженнях

керівник роботи __Полякова Н. П., канд. технічних наук, доцент
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом ЗНУ від _"25"_травня_2020_року № 600-с_____

2. Строк подання студентом кваліфікаційної роботи___30.11.2020____
3. Вихідні дані магістерської роботи
- комплект нормативних документів ;
- технічне завдання до роботи.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
- огляд та збір літератури стосовно теми кваліфікаційної роботи;
- огляд та аналіз існуючих рішень та аналогів;
- дослідження проблеми ефективного використання реактивного підходу при створенні програмних систем;
- створення програмного продукту та його опис;
- перелік вимог для роботи програми;
- дослідження поставленої проблеми та розробка висновків та пропозицій.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
__15__ слайдів презентації

6. Консультанти розділів магістерської роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата Завдання прийняв

7. Дата видачі завдання 01.09.2020

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів магістерської роботи	Строк виконання етапів магістерської роботи	Примітка
1	Аналіз предметної області	02.09-10.09.20	виконано
2	Формулювання основної задачі дипломної роботи та узгодження її з науковим керівником	11.09-12.09.2020	виконано
3	Аналіз існуючих методів рішення	13.09-17.09.20	виконано
4	Дослідження області проектування і розробки реактивних систем	18.09-24.09.20	виконано
5	Узгодження подальших дій з науковим керівником	25.09-26.09.20	виконано
6	Аналіз теоретичних відомостей	27.09-15.10.20	виконано
7	Проектування інтерфейсу налаштувань параметрів тестування	15.10-23.10.20	виконано
8	Узгодження інтерфейсу з науковим керівником	23.10-24.10.20	виконано
9	Реалізація реактивної системи	25.10-10.11.20	виконано
10	Представлення отриманих результатів науковому керівнику і узгодження плану подальшого дослідження	11.11-12.11.20	виконано
11	Реалізація функціоналу імперативної системи	13.11-19.11.20	виконано
12	Проведення аналізу можливостей розроблених програмних за стосунків	20.11-22.11.20	виконано
13	Оформлення звіту	23.11-27.11.20	виконано

Студент _____ Вуколова А. І
(прізвище та ініціали)

Керівник роботи _____ Полякова Н.П.
(підпис) (прізвище та ініціали)

Нормоконтроль пройдено

Нормоконтролер _____ Скрипник І.А.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Вуколова А. І. Використання реактивного програмування для оптимізації ресурсів комп'ютера при пікових навантаженнях.

Кваліфікаційна робота магістра складається із введення, 4 розділів і висновків, списку використаних джерел з 43 найменувань. Робота містить 102 сторінки тексту, 33 рисунки, 15 лістингів коду.

Кваліфікаційна робота для здобуття ступеня вищої освіти магістра за спеціальністю 121 – Інженерія програмного забезпечення, науковий керівник Н.П. Полякова. Інженерний навчально-науковий інститут ЗНУ, 2020.

Мета роботи полягає у дослідженні, вивченні методів розробки та проектування реактивних систем та реалізації методів завдяки парадигмі реактивного програмування, та порівнянні реактивного підходу з імперативним за певними параметрами, які впливають на швидкість роботи застосунку та отримання користувачем результатів, а також на раціональне навантаження машини.

Досліджено найсучасніші конкуруючі фреймворки та бібліотеки для створеня реактивних систем, порівняно їх переваги та недоліки, детально вивчено їх можливості. Спроектовано та реалізовано два застосунки на мові програмування Java, один з яких розроблений у реактивному стилі, а інший – в імперативному. Одержано результати, які є відображенням переваг використання парадигми рекативного програмування у порівнянні з імперативним підходом.

Ключові слова: *РЕАКТИВНІ СИСТЕМИ, РЕАКТИВНЕ ПРОГРАМУВАННЯ, ІМПЕРАТИВНЕ ПРОГРАМУВАННЯ, ФРЕЙМВОРК, JAVA, PROJECT REACTOR, SPRING FRAMEWORK, КОМП'ЮТЕРНИЙ ЗАСТОСУНОК*

SUMMARY

Vukolova A. I. Using reactive programming to optimize computer resources at peak loads.

Qualification work for the degree of Master's degree in specialty 121 - Software Engineering, supervisor N.P. Polyakova. Engineering Educational and Scientific Institute of ZNU, 2020.

The aim of the research is to investigate and study methods of development and design of reactive systems and methods through the paradigm of reactive programming, and compare the reactive approach with the imperative of certain parameters that affect the speed of the application and the user's results, as well as rational machine load.

The most modern competing frameworks and libraries for creation of reactive systems are investigated, their advantages and disadvantages are compared, their possibilities are studied in detail. Two applications in the Java programming language have been designed and implemented, one of which is developed in a reactive style and the other in an imperative style. The results are obtained, which reflect the advantages of using the paradigm of recitative programming in comparison with the imperative approach.

Keywords: *REACTIVE SYSTEMS, REACTIVE PROGRAMMING, IMPERATIVE PROGRAMMING, FRAMEWORK, JAVA, PROJECT REACTOR, SPRING FRAMEWORK, COMPUTER APPLICATION*

ЗМІСТ

ВСТУП

Актуальність теми

Початок 21 століття ознаменувався появою великих систем, з якими працює величезна кількість користувачів. Щосекундна кількість запитів до системи нараховує не тисячі, а мільйони. Технологія реактивного програмування поступово набирає популярність серед розробників, тому що вона спрощує роботу з потоковими даними.

З ростом числа користувачів інтернету і технологій, які приводять в дію веб-сайти протягом багатьох років, реактивне програмування з'явилося як спосіб задоволення цих підвищених вимог до розробників. Звичайно, розробка додатків зараз так само важлива, і реактивне програмування є життєво важливим компонентом і в цій сфері.

Реактивні додатки представляють широкий клас програмного забезпечення, якому необхідно безперервно і в інтерактивному режимі реагувати на внутрішні або зовнішні подразники за допомогою відповідних дій. Приклади таких додатків включають в себе інтерактивне програмне забезпечення, таке як графічні інтерфейси і веб-додатки, графічні анімації, збір даних з датчиків і розподілені системи на основі подій. За останні кілька років реактивне програмування привернуло увагу дослідників і практиків через можливість висловити інакше складну поведінку застосунків.

Актуальність реактивного програмування обумовлена загальновідомою складністю реактивних додатків, які важко розробляти і розуміти через змішане сполучення даних і потоків управління. Реактивне програмування підтримує дизайн, заснований на потоках даних і значеннях, що змінюються в часі.

Мета і завдання дослідження

Мета дослідження полягає в аналізі методів розробки та проектування реактивних систем, а також реалізації їх методів завдяки парадигмі реактив-

ного програмування. Завданням є порівняння реактивного підходу з імперативним за певними параметрами, які впливають на швидкість роботи застосунку та отримання користувачем результатів, а також на раціональне навантаження машини.

Об’єкт дослідження

Реактивна система, побудована з використанням технологій Spring 5 та Project Reactor, імперативна система.

Предмет дослідження

Оптимальне розподілення ресурсів комп’ютера при пікових навантаженнях.

Методи дослідження

Для розв’язання представлених завдань використовуються такі методи дослідження:

1. Аналіз джерел з питань реактивного програмування, розподілу навантаження комп’ютера та проектування реактивних систем.
2. Порівняння існуючих рішень та застосунків.
3. Синтез отриманих результатів досліджень.
4. Порівняльний аналіз результатів тестування навантаження за використаних підходів реалізації програмного застосунку.

Наукова новизна одержаних результатів

В результаті проведених експериментів було отримано дані для порівняння декількох архітектурних підходів до створення реактивної системи за якісними та кількісними показниками, та запропоновано найефективніший з них. Отримані та проаналізовані власні статистичні дані для порівняння реактивного та імперативного підходу реалізації.

Практичне значення одержаних результатів

Одержані результати можуть бути використані для оцінки переваг та недоліків використання реактивного програмування з метою оптимального застосування ресурсів комп'ютера та вирішення проблем пікових навантажень.

Результати роботи дозволяють покращити розуміння розробниками того, які фреймворки та мови програмування є найбільш зручними для вирішення задачі оптимального застосування ресурсів та вирішення проблем пікових навантажень.

Отримані результати допоможуть у розробці застосунків, які будуть передавати потрібні дані користувачеві з мінімальною затримкою у часі.

Апробація результатів

Результати досліджень, викладені у кваліфікаційній роботі магістра були представлені на XXV науково-технічній конференції студентів, магістрантів, аспірантів, молодих вчених та викладачів [43] і опубліковані в збірнику наукових праць студентів, аспірантів і молодих вчених «Молода наука - 2020» [42].

Глосарій

Архітектура програмного забезпечення (англ. software architecture) — спосіб структурування програмної або обчислювальної системи, абстракція елементів системи на певній фазі її роботи. Система може складатись з кількох рівнів абстракції і мати багато фаз роботи, кожна з яких може мати окрему архітектуру.

Асинхронність — це процес обробки введення/виводу, що дозволяє продовжити обробку інших завдань, не чекаючи завершення попереднього завдання.

Багатоплатфо́рмність (кросплатформність, мультиплатформність) — властивість програмного забезпечення працювати більш ніж на

одній програмній або апаратній платформі; технології, що дозволяють досягти такої властивості. Кросплатформність дозволяє суттєво скоротити витрати на розробку нового або адаптацію існуючого програмного забезпечення.

Бібліотека (від англ. Library) в програмуванні — збірник підпрограм або об'єктів, які використовуються для розробки програмного забезпечення.

Бізнес-логіка — сукупність правил, принципів, залежностей поведінки об'єктів предметної області, реалізація правил і обмежень автоматизованих операцій.

Вебсервер — це сервер, що приймає HTTP-запити від клієнтів, зазвичай веббраузерів, видає їм HTTP-відповіді, зазвичай разом з HTML-сторінкою, зображенням, файлом, медіа-потоком або іншими даними. Веб-сервером називають як програмне забезпечення, що виконує функції веб-сервера, так і комп'ютер, на якому це програмне забезпечення працює. Клієнти дістаються веб-сервера за URL-адресою потрібної їм веб-сторінки або іншого ресурсу.

Віддалений виклик процедур (remote procedure call) — клас технологій, що дозволяють комп'ютерним програмам викликати функції або процедури в іншому адресному просторі

Відкрите програмне забезпечення (open source) — програмне забезпечення з відкритим вихідним кодом.

Імперативне програмуваання — парадигма програмування, згідно з якою описується процес отримання результатів як послідовність інструкцій зміни стану програми.

Контролер — компонент програми, який забезпечує взаємодію між іншими компонентами програми. Як приклад можна навести контролер, який забезпечує взаємодію між внутрішнім уявленням даних і поданням даних до інтерфейсу користувача.

Маніфест — це опублікована декларація про наміри, мотиви або погляди емітента, будь то окрема особа, група, політична партія або уряд.

Мікросервіс — сервіс, який працює у власному процесі та підтримує комунікацію з іншими сервісами використовуючи легковагкі механізми, як

правило HTTP, побудований навколо бізнес-потреб і розгортається незалежно з використанням повністю автоматизованого середовища.

Мова програмувння (англ. Programming language) — це штучна мова, створена для передачі команд машинам, зокрема комп'ютерам. Мови програмування використовуються для створення програм, котрі контролюють поведінку машин, та запису алгоритмів.

Модуль — функціонально закінчений фрагмент програми, оформлений у вигляді окремого файлу з вихідним кодом або поіменованої безперервної її частини.

Парадигма програмування — це система ідей і понять, які визначають стиль написання комп'ютерних програм, а також спосіб мислення програміста.

Подія (англ. event) — дія яка розпізнається програмним забезпеченням та оброблюється за допомогою певних інструкцій. Комп'ютерні події можуть бути згенеровані або ініціалізовані системно, користувачем в інший спосіб.

Потік даних — сукупність даних, що є результатом виконання команд, процедур чи програм, які передаються для подальшого оброблення або для пересилання віддаленим користувачам.

Прикладний програмний інтерфейс (інтерфейс програмування застосунків, інтерфейс прикладного програмування) (англ. Application Programming Interface, API) — набір визначень підпрограм, протоколів взаємодії та засобів для створення програмного забезпечення. Спрощено - це набір чітко визначених методів для взаємодії різних компонентів. API надає розробнику засоби для швидкої розробки програмного забезпечення. API може бути для веб-базованих систем, операційних систем, баз даних, апаратного забезпечення, програмних бібліотек.

Реактивне програмування — це парадигма програмування, побудована на потоках даних і розповсюдженні змін. Це означає, що у мовах програмування має бути можливість легко виразити статичні чи динамічні потоки да-

них, а реалізована модель виконання буде автоматично розсилати зміни через потік даних.

Фреймворк — програмна платформа, яка визначає структуру програмної системи; програмне забезпечення, що полегшує розробку і об'єднання різних компонентів великого програмного проекту; інфраструктура програмних рішень, що полегшує розробку складних систем. Спрощено дану інфраструктуру можна вважати своєрідною комплексною бібліотекою, але при цьому вона має ряд обмежень, що задають правила створення структури проекту та написання коду.

Шаблон проектування, чи патерн, у розробці програмного забезпечення — повторювана архітектурна конструкція, що є вирішенням проблеми проектування, у рамках деякого часто виникаючого контексту.

РОЗДІЛ 1 ДОСЛІДЖЕННЯ ПРОБЛЕМИ ВИКОРИСТАННЯ РЕАКТИВНИХ СИСТЕМ

1.1 Історія реактивних систем

В останні роки вимоги до програмного забезпечення значно змінилися. Десятки серверів, час відгуку в кілька секунд, оффлайнове обслуговування, яке могло тривати годинами, гігабайти даних - такими були великі програми буквально кілька років тому. Сьогодні ж програми працюють абсолютно на всьому, починаючи зі смартфонів і закінчуючи кластерами з тисячі процесорів. Користувачі очікують надшвидкого відгуку і стовідсоткової працездатності, в той час як дані виросли до дуже значних розмірів [1].

З часу створення Reactive Manifesto в 2013 році тема Reactive перетворилася з практично невизнаного методу конструювання додатків, який використовувався тільки незначними проектами в кількох обраних корпораціях, на частину загальної стратегічної платформи в багатьох великих гравцях в області проміжного програмного забезпечення [2].

З 2015 року спостерігається величезне зростання інтересу до Reactive з боку як комерційних постачальників проміжного програмного забезпечення, так і користувачів[2].

Реактивні додатки представляють широкий клас програмного забезпечення, яке дозволяє безперервно і в інтерактивному режимі реагувати на внутрішні або зовнішні подразники за допомогою відповідних дій. Приклади таких додатків включають в себе інтерактивне програмне забезпечення, таке як графічні інтерфейси і веб-додатки, графічні анімації, збір даних з датчиків і розподілені системи на основі подій. За останні кілька років реактивне програмування привернуло увагу дослідників і практиків через можливість висловити складну реактивну поведінку інтуїтивно і декларативно [1].

Архітектура реактивних додатків дозволяє програмістам створювати орієнтовані на події, масштабовані, відмовостійкі і відповідаючі на запити

програми, додатки, що працюють в реальному часі і забезпечують хороший час реакції, засновані на масштабованому і відмовостійкому стеку, які легко розгорнути на багатоядерних і хмарних архітектурах. Ці особливості критично важливі для реактивності [1].

1.2 Поняття реактивного програмування

Реактивне програмування – парадигма програмування, що побудована на потоках даних та взаємному розповсюдженні змін [1]. Шини подій або типові події натискань на кнопки — це все реальні приклади асинхронних подієвих потоків, які можна слухати і виконувати деякі побічні дії. По суті — реактивність експлуатує цю ідею. Розробникам надається можливість створювати потоки, а не тільки події натискань. Потоки використовуються всюди: змінні, користувальницькі поля для введення даних, властивості, кеш, структури даних і багато іншого. Наприклад, стрічка Твітера може бути потоком даних нарівні з низкою подій користувальницького інтерфейсу. Тобто можна слухати потік і реагувати на події в ньому.

Потік — це послідовність, що складається з постійних подій, відсортованих за часом. У ньому може бути три типи повідомлень: значення (дані деякого типу), помилки і сигнал про завершення роботи. Потоки — це центральна ідея реактивності [2].

Набір інструментів і функцій дозволяють створювати і фільтрувати кожен з потоків. Потоки можуть бути використані як вхідні параметри один одного. Навіть множинний потік може бути використаний як вхідний аргумент іншого потоку. Можна об'єднувати кілька потоків, фільтрувати один потік, щоб потім отримати інший, який містить тільки актуальні дані. Також можна об'єднувати дані з одного потоку з даними іншого, щоб отримати ще один [2].

Реактивні — значить готові реагувати на зовнішні події, це означає що компоненти весь час активні і завжди готові отримувати повідомлення. Це

визначення розкриває суть реактивних додатків, фокусуючись на системах, які реагують на події, реагують на підвищення навантаження, реагують на збої, реагують на користувачів.

Кожна з цих характеристик має велике значення для реактивних додатків. Всі вони залежать одна від одної, але не як яруси стандартної багаторівневої архітектури. Навпаки, вони описують властивості, застосовні на всьому стеку технологій.

Реактивні додатки є збалансованим підходом для вирішення сучасних проблем в розробці програмних систем. Вони побудовані на каркасі, орієнтованому на події та передачі повідомлень, і забезпечують інструменти для забезпечення масштабованості і відмовостійкості. Поверх цього вони підтримують складні інтерфейси для взаємодії з користувачем [3].

1.3 Теорія реактивних систем

Реактивне програмування — це окрема підмножина реактивних систем на рівні реалізації. Воно пропонує розробникам продуктивність та ефективність, за рахунок використання ресурсів на рівні компонентів для внутрішньої логіки і управління потоками даних [3].

Reactive Systems забезпечує продуктивність для архітекторів і DevOps на системному рівні завдяки стійкості і еластичності, для побудови Cloud Native або інших великомасштабних розподілених систем [4].

Звичайно, найбільш продуктивно буде використовувати реактивний підхід у програмуванні в архітектурніх компонентах реактивної системи.

Вельми вигідно використовувати реактивні системи для створення системи навколо компонентів, написаних з використанням реактивного програмування.

Реактивність – це набір принципів проектування.

Одним з останніх показників успіху є те, що Reactive став перевантаженим терміном і тепер асоціюється з кількома різними речами, наприклад, зі словами «потокова передача», «полегшений» і «в реальному часі» [3].

З точки зору експертів «Reactive» — це набір принципів проектування для створення зв’язних систем. Це спосіб створити архітектуру і дизайн систем в розподіленому середовищі, де методи реалізації, інструменти та шаблони проектування є компонентами цілої системи.

У реактивній системі велике значення має взаємодія між окремими частинами, а саме здатність працювати індивідуально, але діяти узгоджено для досягнення бажаного результату [4].

Реактивна система заснована на архітектурному стилі, який дозволяє кільком окремим службам об'єднуватися в єдине ціле і реагувати на оточення, залишаючись при цьому обізнаними один про одного. Це може проявлятися в можливості масштабування вгору / вниз, балансування навантаження [4].

Таким чином, ми бачимо, що можна написати один додаток в стилі Reactive (тобто використовуючи Reactive Programming); однак, це всього лише одна частина головоломки. Хоча кожен з перерахованих вище аспектів може здатися «реактивним», самі по собі вони не роблять систему реактивною.

Коли люди говорять про Reactive в контексті розробки і проектування програмного забезпечення, вони зазвичай мають на увазі або реактивні системи (архітектура і дизайн), або реактивне програмування (декларативне засноване на подіях).

Буде розглянуто більш детально, що означає кожна з цих практик і технік. Більш конкретного визначення потребують наступні питання: коли їх використовувати, як вони пов'язані одна з одною, і які вигоди очікувати від кожної з них, особливо в контексті побудови систем.

1.4 Проблеми, що вирішує реактивне програмування

Підвищення рівня абстракції коду

Реактивний підхід використовують для підвищення рівня абстракції коду. Замість постійної підтримки коду з високою деталізацією можна сконцентруватися на взаємозв'язку подій, які визначають бізнес-логіку [5]. Реактивне програмування може значно скоротити вихідний код, написаний в імперативному стилі [6].

Обробка великої кількості UI-подій

Перевага реактивного підходу більш помітна в сучасних веб- і мобільних додатках, які працюють з великою кількістю різноманітних UI-подій. 10 років тому вся взаємодія з веб-сторінкою зводилася до відправки великих форм на сервер і виконання простого рендерингу у клієнтській частині. Зараз додатки складніші: зміна одного поля може потягнути за собою автоматичне збереження даних на сервері, інформація про нову відмітку «сподобалося» повинна відправитися іншим підключеним користувачам і т.д. Реактивне програмування дуже добре підходить для обробки великої кількості різноманітних подій [7].

Асинхронне використання потоків

Асинхронність в програмуванні — виконання процесу в неблокуючому режимі, що дозволяє програмі продовжити обробку даних.

Асинхронний код прибирає блокуючу операцію з основного потоку програми, так що вона продовжує виконуватись, але десь в іншому місці, а обробник може обробляти далі. Простіше кажучи, головний «процес» ставить завдання і передає її іншому незалежному «процесу»[2].

Події — це теж асинхронні потоки даних, які можна прослуховувати, щоб реагувати на них будь-якими діями. Можна створювати потоки даних не тільки з подій наведення або клікання мишкою. Потоком може бути що за-

вгодно: змінні, користувальницькі поля введення, властивості, кеш, структури даних [2].

1.5 Маніфест реактивних систем

Реактивна система повинна задовольняти деякому набору правил (reactive manifesto). Даний маніфест був розроблений в 2013 році для усунення невизначеності. Справа в тому, що на той момент в Європі і США термін «reactive» не був чітко окреслений і охоплював надто багато понять і властивостей. Кожен розумів по-своєму, яку систему можна назвати реактивною. Це породжувало величезну плутанину, і в підсумку був створений маніфест, який встановив чіткі критерії реактивної системи (Рис. 1).

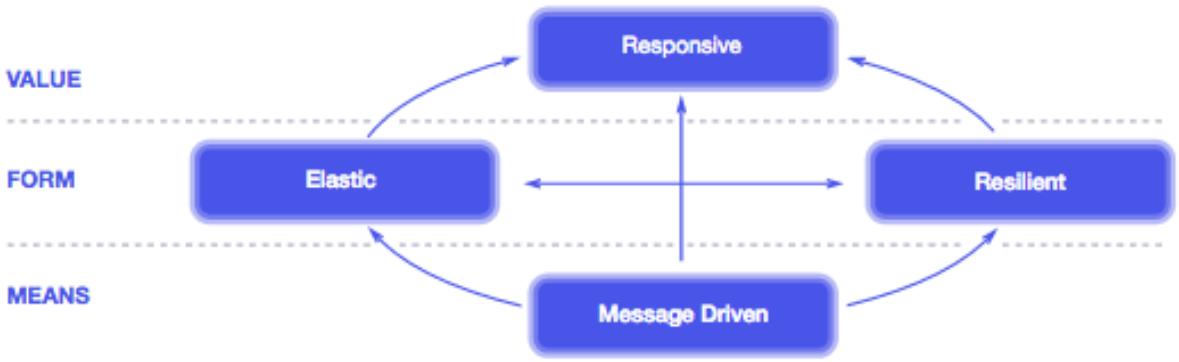

Рис. *1 Критерії реактивної системи*

Програми, які написані відповідно до визначених вище принципів, відрізняються підвищеною гнучкістю, масштабованістю і слабкою зв'язаністю. Завдяки цьому їх простіше розробляти і модифікувати. Вони більш стійкі до відмов і правильно обробляють виняткові ситуації, уникаючи катастрофічних наслідків. Реактивні системи характеризуються високою здатністю до взає-

модії, забезпечуючи користувачам ефективний та інтерактивний зворотний зв'язок.[1]

Далі більш детально розглянуто кожен з принципів маніфесту реактивних систем.

1.5.1 Орієнтованість на повідомлення

Програми, що використовують асинхронну модель, набагато краще забезпечують слабку зв'язаність, ніж додатки, що базуються на суто синхронних викликах. Відправник і одержувач можуть бути реалізовані, не спираючись на деталі того, як події поширюються в системі, що дозволяє інтерфейсам фокусуватися на вмісті передачі. Це призводить до реалізації, яку легше розширювати, змінювати і підтримувати, забезпечує більшу гнучкість і зменшує вартість підтримки [1].

Так як одержувачі асинхронної взаємодії не діють, поки не отримають повідомлення, то такий підхід може дозволити ефективно використовувати ресурси, роблячи можливим великій кількості споживачів працювати в одному і тому ж апаратному потоці. Таким чином, неблокуючий додаток може мати нижчу латентність і більшу пропускну здатність у порівнянні з традиційним додатком, заснованим на блокуючій синхронізації і примітивах комунікації. Це призводить до зменшення вартості операцій, збільшення утилізації ресурсів процесора і спрощує роботу кінцевих користувачів.

У подієво-орієнтованому додатку компоненти взаємодіють один з одним шляхом відправки та отримання повідомлень – дискретних частин інформації, що описують факти. Ці повідомлення відправляються і приймаються в асинхронному і неблокуючому режимі. Подієво-орієнтовані системи більш схильні до push-моделі, ніж до pull або poll. Тобто вони проштовхують дані до своїх споживачів, коли дані стають доступними, замість того щоб даремно витрачати ресурси, постійно запитуючи або чекаючи дані [1].

Асинхронна передача повідомлень означає, що додаток за своєю природою має високий ступінь конкуренції і може без змін працювати на багато-

ядерній архітектурі. Будь-яке ядро CPU може обробити будь-яке повідомлення, що дає більші можливість паралелізації.

Неблокування означає здатність продовжувати працювати, щоб додаток був здатним до взаємодії весь час, навіть в умовах збою або пікового навантаження. Для цього всі необхідні для забезпечення взаємодії ресурси, наприклад CPU, пам'ять і мережа, не повинні бути монополізовані. Це призведе до більш низької латентності, більшої пропускної спроможності і кращої масштабованості.

Традиційні серверні архітектури використовують загальнодоступний змінюваний стан і блокують операції на одному потоці. Це вносить труднощі при масштабуванні системи. Загальнодоступний змінюваний стан вимагає синхронізації, що привносить складність і недетерменірованість, роблячи код важким для розуміння і підтримки.

Поділяючи генерацію подій і їх обробку, ми дозволяємо платформі самій подбати про деталі синхронізації і диспетчеризації подій між потоками, в той час як самі концентруємося на більш високорівневих абстракціях і бізнес-логіці. З’являється більше можливостей подумати про те, звідки і куди пересилаються події, і про те, як компоненти взаємодіють між собою, замість того щоб розбиратися з низькорівневими примітивами, такими як потоки або блокування.

Подієво-орієнтовані системи забезпечують слабку зв'язаність між компонентами і підсистемами. Така зв'язаність є однією з необхідних умов масштабованості і відмовостійкості. Без складних і сильних залежностей між компонентами розширення системи вимагає мінімальних зусиль [1].

Коли від додатка потрібна висока продуктивність і масштабованість, важко передбачити, де можуть виникнути вузькі місця. Тому дуже важливо, щоб уся реалізація була асинхронною і неблокуючою. Для типового додатку це означає, що архітектура повинна бути повністю подієво-орієнтованою, починаючи з запитів користувачів через графічний інтерфейс і обробки запитів в веб-шарі і закінчуючи сервісами, кешем і базою даних. Якщо хоча б

один з цих шарів не відповідатиме цій вимозі, тобто буде робити блокуючі запити до БД, використовувати загальнодоступний змінюваний стан, викликати синхронні операції, то весь стек затихне і користувачі будуть страждати через велику кількість затримок і нестачу масштабованості.

Додаток має бути реактивним від верху до низу. Необхідність усувати слабку ланку в ланцюзі добре ілюструється законом Амдала (Рис. 2), який свідчить: прискорення програми за рахунок її розпаралелювання обмежується послідовною частиною програми [8]. Наприклад, якщо 95% обсягу обчислень може бути паралельним, то теоретичний максимум прискорення не може перевищити 20, в незалежності від кількість використаних процесорів.

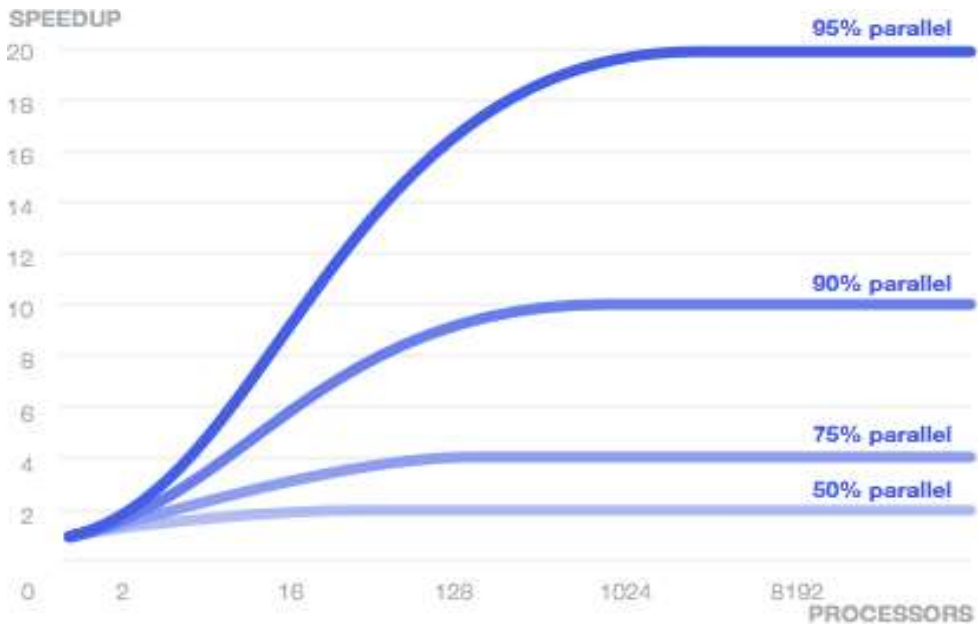

Рис.*2 Ілюстрація закону Амдала*

1.5.2 Масштабованість або гнучкість

Слово масштабований визначається словником Merriam Webster як «здатний легко розширитися або модернізуватися» [9]. Масштабований додаток може бути розширений до необхідних масштабів. Це досягається за рахунок надання додатку еластичності, властивості, яке дозволяє системі розтягуватися або стискатися (додавати або прибирати вузли) на вимогу. Крім того, така архітектура робить можливим розширюватися або скорочуватися

(розгортатися на більшій або меншій кількості процесорів) без необхідності перепроектування або переписування програми [1].

Масштабованість також допомагає управляти ризиками: занадто мала кількість обладнання може привести до незадоволення і втрати клієнтів, а надто велика буде бездіяльною і призведе до зайвих витрат. Масштабований додаток знижує ризик ситуації, коли є обладнання, але додаток не може його використовувати: потенціал майбутніх процесорів з сотнями, якщо не з тисячами апаратних потоків вимагає масштабованості на мікроскопічному рівні [1].

Подієво-орієнтована система, що базується на асинхронній передачі повідомлень, є основою масштабованості. Слабка зв'язаність і локаційна незалежність компонентів і підсистем дозволяють розгортати систему на безлічі вузлів, залишаючись в межах тієї ж самої програмної моделі з тією ж семантикою. При додаванні нових вузлів зростає пропускна здатність системи. У термінах реалізації не повинно бути ніякої різниці між розгортанням системи на більшій кількості ядер або більшій кількості вузлів кластера або центру обробки даних. Топологія додатку стає проблемою конфігурації і / або адаптивних алгоритмів часу виконання, що стежать за навантаженням на систему.

Потрібно охопити мережу, представивши її прямим чином в програмній моделі через механізм асинхронних повідомлень. Справжня масштабованість природним чином покладається на розподілені обчислення і їх міжвузлову взаємодію, що означає обхід мережі, який за своєю суттю є ненадійним. Тому важливо врахувати обмеження, компроміси і сценарії виняткових ситуацій явно в програмній моделі замість того, щоб ховати їх за абстракціями, які нібито намагаються «спростити» архітектуру.

1.5.3 Відмовостійкість

Відмова додатку — одна з найбільш руйнівних речей, які можуть статися. Дуже часто вимога відмовостійкості додатків повсюдно ігнорується або

вирішується ситуативно. Це часто означає, що проблему розглядають не на тому рівні деталізації з використанням занадто неточних і грубих інструментів. Поширеним рішенням є застосування кластеризації сервера додатків з відновленням у випадку відмови під час роботи. На жаль, подібні готові рішення вимагають великих грошових внесків і, крім того, небезпечні оскільки вони потенційно можуть каскадним чином припинити функціонування усього кластеру. Причина в тому, що проблема управління збоями повинна опрацьовуватися детально на рівні взаємодії більш дрібних компонентів.

У реактивному додатку робота над відмовостійкістю не залишається «на потім», а є частиною архітектури з самого початку. Ставлення до збоїв як до об'єктів першого класу в програмній моделі полегшить завдання реагування на них і управління ними, що зробить додаток стійким до збоїв і дозволить системі «лікувати» і «лагодити» саму себе в процесі роботи. Традиційні способи обробки виняткових ситуацій не можуть досягти цього, тому що проблеми вирішуються не на тих рівнях — виключення обробляються прямо там, де вони відбуваються, або ініціюється процедура відновлення всієї програми [1].

Щоб керувати збоями, потрібен спосіб щоб ізолювати їх, так щоб вони не поширювалися на інші працездатні компоненти, і вести за ними спостереження за межами контексту, в якому можуть відбуватися збої. Один спосіб, який приходить на розум, —це перебирання, що розділяють систему на відсіки таким чином, що якщо виходить з ладу один з відсіків, то це ніяк не впливає на інші відсіки. Це запобігає класичній проблемі каскадних збоїв і дозволяє вирішувати проблеми ізольовано [1].

Подієво-орієнтована модель, яка дає масштабованість, також надає необхідні примітиви для вирішення проблеми відмовостійкості. Слабка зв'язаність у подієво-орієнтованої моделі дає повністю ізольовані компоненти, в яких збої інкапсулюються в повідомлення разом з необхідними деталями і пересилаються до інших компонентів, які в свою чергу аналізують помилки і вирішують, як реагувати на них.

Такий підхід створює систему, в якій бізнес-логіка залишається чистою, відокремленої від обробки помилок, збої моделюються явно, щоб розбиття на відсіки, спостереження, управління і конфігурація задавалися декларативно, система може «лікувати» себе і відновлюватися автоматично.

Найкраще, якщо відсіки організовуються ієрархічним чином, подібно до великої корпорації, де проблеми піднімаються до рівня, що має достатньо влади, щоб вжити відповідних заходів.

Міць даної моделі в тому, що вона чисто подієво-орієнтована — вона заснована на реактивних компонентах і асинхронних подіях, і тому має локаційну прозорість. На практиці це означає, що її семантика не залежить від того, чи працює вона на локальному сервері або в розподіленому оточенні.

Адаптивний визначається словником Merriam Webster [9] як «відповідальний швидко або реагує належним чином». Відзначимо, що далі ми будемо використовувати це слово в його загальному сенсі і не будемо плутати з адаптивним веб-дизайном з його CSS медіа-запитами і прогресивними поліпшеннями.

1.5.4 Взаємодія, здатність до відгуків

Додатки, що здатні до взаємодії з користувачем — це додатки реального часу, вони насичені функціоналом і надають спільний доступ. З клієнтами підтримується відкритий і безперервний діалог через здатність до відгуків, взаємодію та інтерактивність. Це робить роботу клієнтів більш продуктивною, створює відчуття сталості і готовності в будь-який момент вирішити проблеми і виконати потрібні завдання. Одним з таких прикладів є Документи Google, які підтримують функцію спільного редагування в режимі реального часу, що дозволяє користувачам безпосередньо бачити правки один одного.

Додатки повинні своєчасно реагувати на події, навіть в умовах збою. Якщо програма не відповідає в межах розумного проміжку часу, то за фактом вона недоступна і тому її не можна вважати відмовостійкою [1].

Багато додатків швидко стають марними, якщо вони перестають відповідати сучасним вимогам, наприклад, додаток, який здійснює торгові операції, може втратити поточну операцію, якщо не встигне відповісти вчасно.

Більш загальнопоширені додатки, такі як онлайн-магазини роздрібних покупок, втрачають прибуток, якщо час відгуку збільшується. Користувачі інтенсивніше взаємодіють з додатками, які здатні до швидких відгуків, що призводить до великих обсягів покупок.

Реактивні додатки використовують спостережувані моделі і потоки подій.

Спостережувані моделі дозволяють іншим системам отримувати події, коли їх стан змінюється. Це забезпечує зв'язок в реальному часі між користувачами і системами. Наприклад, коли кілька користувачів працюють одночасно над однією і тією ж моделлю, зміни можуть реактивно синхронізуватися між ними, позбавляючи від необхідності блокування моделі [1].

Потоки подій утворюють базову абстракцію, на якій будуються такі зв'язки. Зберігаючи їх реактивними, уникаються блокування і перетворення та комунікацій можуть бути асинхронними і неблокуючими.

Реактивні додатки повинні знати про порядки алгоритмів, щоб час відгуку на події не перевищував O (1) або, як мінімум, O (log n) незалежно від навантаження. Може бути включений коефіцієнт масштабування, але він не повинен залежати від кількості клієнтів, сесій, продуктів [1].

Далі наведено кілька стратегій, які допоможуть зберегти швидкодію відгуку незалежно від профілю навантаження.

У разі вибухового трафіку реактивні додатки повинні амортизувати витрати на дорогі операції, такі як введення-виведення або конкурентний обмін даними, застосовуючи групування з розумінням і врахуванням специфіки ресурсів.

Черги повинні бути обмежені з урахуванням інтенсивності потоку, довжини черг при даних вимогах на час відгуку повинні визначатися відповідно до закону Літтла [10].

Системи повинні перебувати в стані постійного моніторингу і мати адекватний запас міцності.

У разі збоїв активуються автоматичні вимикачі і запускаються запасні стратегії обробки.

1.6 Огляд існуючих реактивних бібліотек та фреймворків

Реактивні бібліотеки та фреймворки – це проекти які містять імплементацію реактивного програмування для різних мов. Кожна окрема імплементація надає розробникам API для створення та роботи з потоками, а також даними всередині потоків.

Найвідоміші технології для імплементування реактивності описані у цьому підрозділі.

1.6.1 Reactive Extensions

У створенні програмного забезпечення Reactive Extensions (також відомий як ReactiveX) є набором інструментів, що дозволяють імперативним мовам програмування працювати з послідовностями даних, незалежно від того, чи є ці дані синхронними або асинхронними. Він пропонує набір операторів і алгоритмів, які працюють з кожним елементом у послідовності. Він представляє собою реалізацію реактивного програмування та пропонує схему інструментів, які будуть реалізовані декількома мовами програмування [11].

ReactiveX — це API для асинхронного програмування із спостереженнями за потоками [11].

Асинхронне програмування дозволяє програмістам викликати функції, а потім виконувати функції «зворотного виклику», коли вони будуть виконані. Програми, розроблені таким чином часто уникають зайвих витрат, пов'язаних з постійним запуском та зупинкою багатьох потоків.

Потоки, що спостерігаються у контексті Reactive Extensions подібні джерелам подій, які генерують 3 події (Рис. 3):

1. Next — чергова порція даних.

2. Error — сталася помилка.

3. Completed - потік завершений і даних більше не буде.

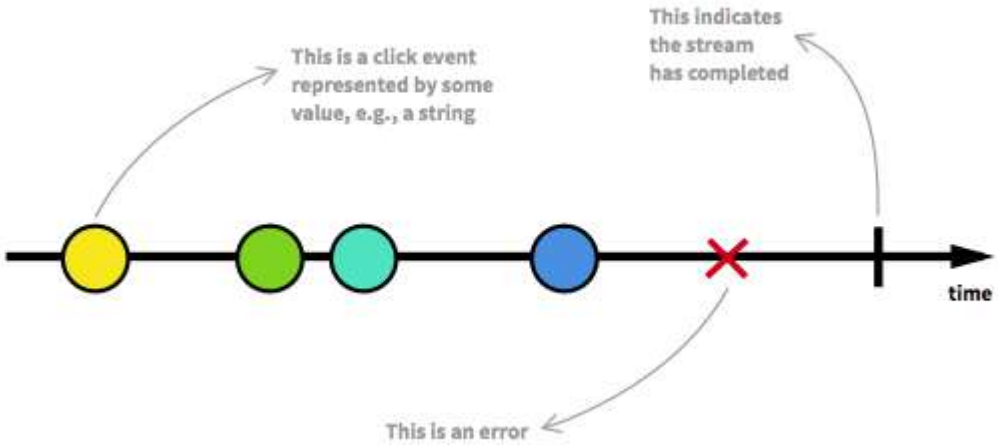

Рис. *3 Послідовність подій*

Reactive Extensions пропонує набір інструментів для різних мов програмування.

Rx.NET: Reactive Extensions (Rx) — це бібліотека для створення асинхронних програм, заснованих на подіях, з використанням спостережуваних послідовностей і операторів запитів в стилі LINQ.

RxCpp: Reactive Extensions for Native (RxCpp) — це бібліотека для створення асинхронних програм і програм, заснованих на подіях, з використанням спостережуваних послідовностей і операторів запитів в стилі LINQ як в C, так і в C ++.

Rx.rb: прототип реалізації реактивних розширень для Ruby (Rx.rb).

RxPy: Reactive Extensions для Python 3 (Rx.Py) — це набір бібліотек для створення асинхронних і заснованих на подіях програм з використанням спостережуваних колекцій і операторів запитів в стилі LINQ в Python 3.

Бібліотеки RxJava — реалізація реактивного програмування для Java та RxJS — бібліотека для мови Java Script, є найбільш популярними та вико-

ристовуються значно частіше бібліотек для інших мов, тому описані більш детально далі.

1.6.2 RxJava

Java – одна з найпопулярніших та найзатребуваніших мов програмування. Однак, Java не є "реактивною мовою" в тому сенсі, що вона не підтримує впорядкованих процедур. Тому останнім часом було докладено багато зусиль аби забезпечити реактивні шари поверх JDK.

RxJava — це специфічна реалізація реактивного програмування для Java та Android. Вона надає перевагу композиції функцій, уникненню глобальних станів та побічних ефектів та використанню потоків для складання асинхронних та подієво-орієнтованих програм. За основу було взято патерн проектування Observer, який абстрагує зворотні виклики виконавця та споживача, він поширюється на десятки операторів, які дозволяють складати, трансформувати, планувати, обробляти помилки та керувати життєвим циклом [12].

RxJava — це зріла бібліотека з відкритим кодом, яка знайшла широке застосування як на серверах, так і на мобільних пристроях на базі Android.

У Java є клас Observable і інтерфейс Observer. Реалізація інтерфейсу Observer — це об'єкт, який чекає на подію. Observable — це клас, який всім переданим йому Observer-об'єктам повідомляє про те, що подія настала. Ці ж назви використовуються і в RxJava. І сенс їх залишився тим же: Observable генерує подію, а Observer отримує її. Але було значно розширено саме поняття "подія". У RxJava події, які Observable передає в Observer, можна розглядати як потік даних [13].

1.6.3 RxJS

Реактивні розширення для JavaScript (RxJS) — це бібліотека для складання асинхронних програм і програм на основі подій з використанням спостережуваних послідовностей і операторів запитів в стилі LINQ в JavaScript, які можуть бути націлені як на браузер, так і на Node.js.

RxJS спрощує створення асинхронного коду або коду на основі зворотного виклику [14].

RxJS надає реалізацію типу Observable, який необхідний доти, поки тип не стане частиною мови і поки браузери не підтримають його. Бібліотека також надає службові функції для створення спостережуваних об'єктів і роботи з ними [14]. Ці службові функції можуть використовуватися для:

— перетворення існуючого коду для асинхронних операцій в спостережувані;

— ітерування значень в потоці;

— зіставлення значень з різними типами;

— фільтрації потоків; складання кількох потоків.

RxJS пропонує ряд функцій, які можна використовувати для створення нових Observables. Ці функції можуть спростити процес створення Observables з таких речей, як події, таймери, обіцянки і т. д.

Оператори — це функції, які побудовані на основі спостережуваних об'єктів і дозволяють виконувати складні операції з колекціями. Наприклад, RxJS визначає такі оператори, як map (), filter (), concat () і flatMap ().

Оператори беруть параметри конфігурації і повертають функцію, яка приймає спостережуване джерело. При виконанні цієї повертаємої функції оператор спостерігає за вихідними значеннями вихідного спостережуваного об’єкту, перетворює їх і повертає новий спостережуваний об’єкт з цих перетворених значень.

Можна використовувати канали(pipes) для зв'язування операторів разом. Канали дозволяють об'єднати кілька функцій в одну. Функція pipe () приймає в якості аргументів функції, які потрібно об'єднати, і повертає нову функцію, яка при виконанні послідовно запускає складені функції [14].

Набір операторів, що застосовуються до спостережуваного об’єкту — це рецепт, тобто набір інструкцій для отримання потрібних значень. Сама по

собі інструкція нічого не робить. Потрібно викликати subscribe (), щоб отримати результат через інструкцію.

На додаток до обробника помилок, який надається при підписці, RxJS надає оператор catchError, який дозволяє обробляти відомі помилки в спостережуваній інструкції.

Наприклад, припустимо, що є спостережуваний об'єкт, який робить запит API і зіставляється з відповіддю сервера. Якщо сервер повертає помилку або значення не існує, видається помилка. Якщо знайти цю помилку і надати значення за замовчуванням, потік продовжить обробляти значення, а не виводити помилки [14].

1.6.4 Reactive streams

Reactive streams — це ініціатива з надання стандарту для асинхронної обробки потоків з неблокуючим зворотним тиском [15], це контракт на дуже низькому рівні, виражений у кількох інтерфейсах Java, але також може бути застосовний до інших мов. Reactive Streams (Реактивні Потоки) складаються з 4-х простих Java - інтерфейсів (Publisher, Subscriber, Subscription і Processor). Реактивні потоки були включені в JDK як java.util.concurrent.Flow у версії 9. Проект Reactive streams — це співпраця інженерів з Kaazing, Netflix, Pivotal, Red Hat, Twitter, Typesafe та багатьох інших.

Основним завданням Reactive Streams є обробка backpressure. Backpressure (зворотній натиск) — це механізм, який дозволяє одержувачеві питати, скільки даних він хоче отримати. Тобто одержувач починає отримувати дані тільки тоді, коли він готовий їх обробити [15].

Реактивні потоки — це ініціатива створення стандарту для асинхронної обробки потоку з неблокуючим зворотнім натиском. Це охоплює зусилля, спрямовані на середовище виконання (JVM), а також мережеві протоколи.

Основна мета реактивних потоків — керувати обміном потоковими даними через асинхронну мережу, продумати передачу елементів на інший потік або пул потоків, при цьому гарантуючи, що приймаюча сторона не зму-

шена буферизувати довільні обсяги даних. Іншими словами, зворотній тиск є невід’ємною частиною цієї моделі для того, щоб дозволити обмеження черг, які опосередковуються між потоками.

Наміром цієї специфікації є дозволити створювати безліч відповідних реалізацій, які завдяки дотриманню правил зможуть плавно взаємодіяти, зберігаючи вищезазначені характеристики у всьому графіку обробки потокової програми.

Задача реактивних потоків полягає у пошуку мінімального набору інтерфейсів, методів та протоколів, які описують необхідні операції та сутності для досягнення мети - асинхронні потоки даних з неблокуючим зворотнім тиском. DSL-адреси кінцевого користувача або API-файли, що зв'язують протоколи, цілеспрямовано були вилучені із сфери застосування, щоб заохочувати та давати можливість різним реалізаціям, які потенційно використовують різні мови програмування, залишатися максимально правдивими до ідіом своєї платформи [15].

1.6.5 Akka

Akka - це інструментарій, а не фреймворк, він інтегрується в збірку, як і будь-яка інша бібліотека, не наслідуючи певну схему вихідного коду. Якщо розробляти свої системи як взаємодіючі суб'єкти, можна досягти правильної інкапсуляції внутрішнього стану, виявити природний поділ між бізнес-логікою і міжкомпонентною взаємодією. Akka спрощує створення паралельних і розподілених додатків на JVM [16]. Akka підтримує кілька моделей програмування для паралелізму. Мовні прив'язки існують як для Java, так і для Scala. Akka написаний на Scala .

1.6.6.Фреймворк Netty

Netty — це клієнт-серверна інфраструктура із неблокуючим вводом/виводом, яка дозволяє швидко і легко розробляти мережеві додатки, такі як сервери протоколів і клієнти. Це значно спрощує і оптимізує мережеве програмування, таке як сокети TCP і UDP [17].

«Швидко і просто» не означає, що додаток буде страждати від ремонтопридатності або буде мати проблеми з продуктивністю. Netty був ретельно спроектований з урахуванням досвіду, накопиченого при реалізації багатьох протоколів, таких як FTP, SMTP, HTTP, а також різних двійкових і текстових застарілих протоколів. В результаті Netty вдалося знайти спосіб досягнення простоти розробки, продуктивності, стабільності та гнучкості без компромісів [17].

Основне призначення Netty — створення високопродуктивних протокольних серверів на основі NIO (або, можливо, NIO.2) з поділом і слабким зв'язком компонентів мережі та бізнес-логіки. Він може реалізовувати широко відомий протокол, такий як HTTP, або власний конкретний протокол.

Netty використовує парадигму додатку, керованого подіями, тому конвеєр обробки даних являє собою ланцюжок подій, що проходять через обробники. Події та обробники можуть бути пов'язані з вхідним і вихідним потоком даних.

Вхідні події можуть бути наступними:

1. Активація та деактивація каналу.
2. Читання подій операції.
3. Виняткові події.
4. Події, призначені для користувача.

Вихідні події простіше і, як правило, пов'язані з відкриттям / закриттям з'єднання і записом / скиданням даних.

Netty-додатки складаються з пари мережевих і прикладних логічних подій і їх обробників. Базовими інтерфейсами для обробників подій каналу є

ChannelHandler і його предки ChannelOutboundHandler і ChannelInboundHandler [18].

Netty надає величезну ієрархію реалізацій ChannelHandler. Варто відзначити адаптери, які є просто пустими реалізаціями, наприклад, ChannelInboundHandlerAdapter і ChannelOutboundHandlerAdapter. Ці адаптери можна розширити, коли потрібно обробити тільки підмножину всіх подій.

Також існує багато реалізацій конкретних протоколів, таких як HTTP, наприклад. HttpRequestDecoder, HttpResponseEncoder, HttpObjectAggregator [18].

1.7 Переваги використання реактивного програмування

Серед переваг реактивного програмування можна виділити наступні [19]:

1. Набагато простіше організувати асинхронну / потокову роботу.
2. Багато операторів, які спрощують роботу.
3. Простіше складати композицію з потоків даних.
4. Складна реалізація роботи потоків спрощується.
5. Більш чистій, читаний, впорядкований та зрозумілий код.
6. Легкість реалізації зворотного тиску.

1.8 Приклади успішного використання реактивного програмування у створенні програмного забезпечення

Netflix

Декілька років назад команда Netflix API почала переробляти API, щоб підвищити продуктивність і дати можливість командам розробників UI в Netflix оптимізувати клієнтські програми для конкретних пристроїв.

Netflix API — це Java-додаток, який працює на сотнях серверів, що обробляють понад 2 мільярди вхідних запитів в день для мільйонів клієнтів по всьому світу. Система повинна знижувати ризики, властиві можливості швидкого і частого розгортання декількома групами з мінімальною координацією [20].

Для досягнення поставлених цілей архітектура була розділена на кілька ключових моментів: динамічний час виконання, повністю асинхронний рівень обслуговування, реактивна модель програмування.

Наступна діаграма (Рис. 4) і анотації пояснюють архітектуру:

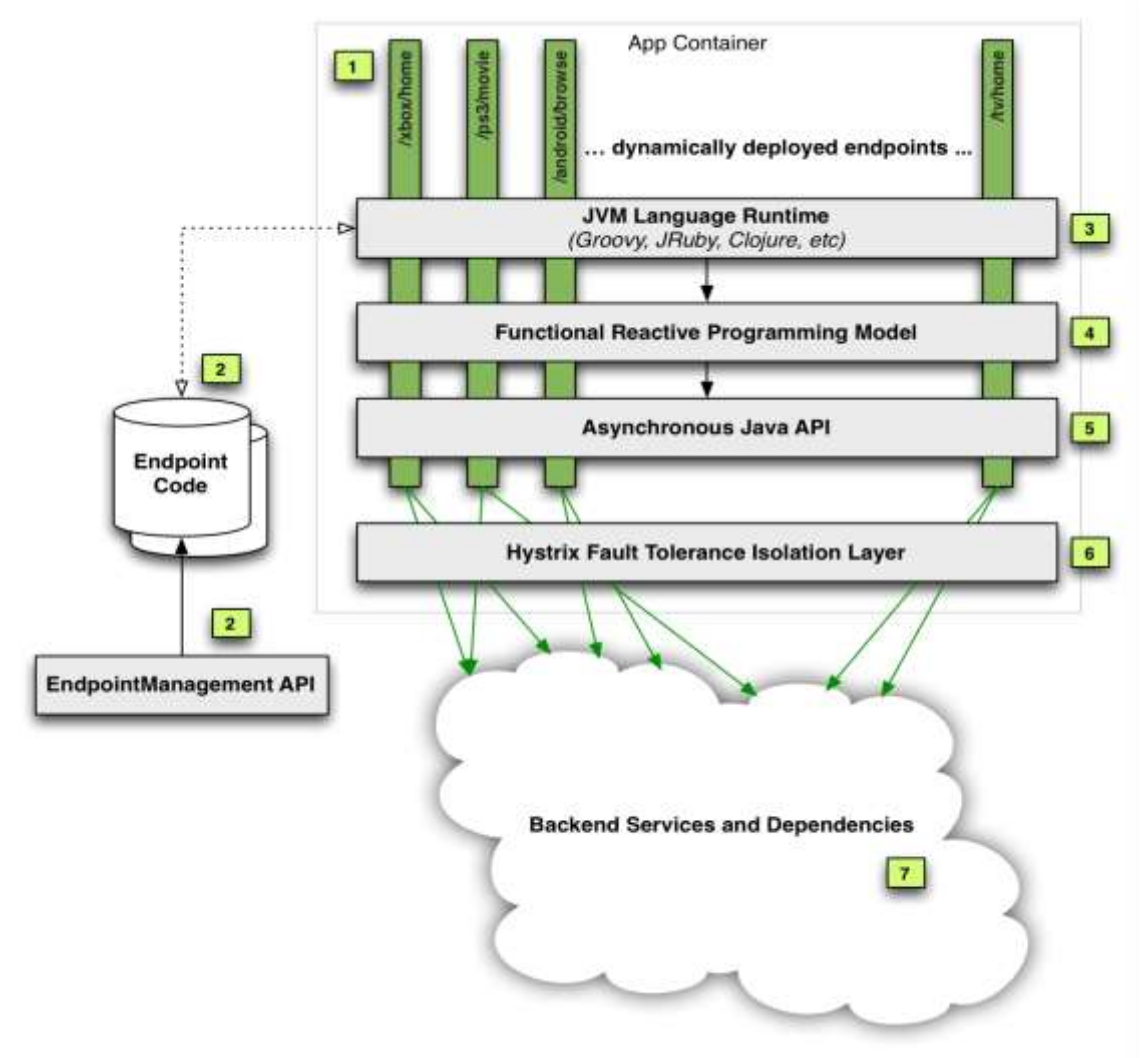

Рис. *4 Діаграма архітектури*

Забезпечення паралелізму було ключовою вимогою для досягнення приросту продуктивності, але абстрагування деталей реалізації потокової безпеки і паралельного виконання від розробників клієнтів було в рівній мірі важливим для зниження складності і прискорення темпів інновацій. Повністю асинхронний API Java був першим кроком, оскільки він дозволяє реалізації базового методу контролювати, чи виконується щось одночасно чи ні без зміни клієнтського коду. Була обрана модель реактивного програмування з функціональним стилем програмування для обробки композиції і умовних потоків асинхронних зворотних викликів [21].

Cortana

Персональна помічниця Кортана покликана передбачати потреби користувача. При бажанні їй можна дати доступ до особистих даних, таким як електронна пошта, адресна книга, історія пошуків в мережі, тощо, всі ці дані вона буде використовувати для попередження потреб користувача. Потрібний запит можна як надрукувати вручну, так і задати голосом. Необхідну інформацію вона буде знаходити, спираючись на результати пошуку в системі Bing, Foursquare і серед особистих файлів користувача. Також віртуальний асистент не позбавлена почуття гумору: вона може підтримувати бесіду, співати і розповідати анекдоти. Вона заздалегідь нагадає про заплановану зустріч, день народження друга і інші важливі події. Інтерфейс має гнучкі налаштування конфіденційності, що дозволяють користувачеві самому визначати, якого роду інформацію надавати віртуальному асистентові. За словами розробників, таким рівнем контролю не може похвалитися ні Siri, ні Google Now [22].

Кортана використовує браузер Edge і пошукову систему Bing і не підтримує інші браузери. Вона інтегрується з деякими додатками з Windows Store.

У Кортани є вікове обмеження — користуватися послугами помічниці не зможуть користувачі, в чиїх Microsoft-акаунтах вказаний вік нижче 13 ро-

ків. При спробі активувати асистента і задати їй будь-яке питання власник почує: «Шкодую, ви повинні бути трохи старше, перш ніж я зможу допомогти вам».

Для розробки такого помічника використовувались алгоритми штучного інтелекту, а також парадигма реактивного програмування, адже головними критеріями електронного помічника були швидка обробка даних, високий рівень взаємодії з користувачем, а отже, здатність до відгуків, також орієнтованість на повідомлення, тобто передачу дискретних частин інформації, та відмовостійкість, що грає дуже велику роль для додатків подібного роду.

SoundCloud

SoundCloud — онлайн-платформа та веб-сайт для розповсюдження оцифрованої звукової інформації (наприклад, музичних творів) володіє функціями соціальної мережі та однойменної організації.

Ключовими особливостями сервісу є можливість розповсюджувати кожен запис окремо, за допомогою унікального URL, що дозволяє їх вмонтовувати в популярні мережі (Twitter, Facebook) на відміну від MySpace, котрий дозволяє слухати плейлист тільки на його особистому веб-сайті. SoundCloud пропонують віджети, котрі дозволяють розміщувати їх на веб-сторінках та блогах.

Окрім Twitter, Facebook, SoundCloud інтегрований з іншими соціальними мережами: Songkick, FourSquare.

Однією за найцікавіших функцій є можливість залишати коментарі в будь-якому місці звукового треку.

SoundCloud використовує парадигму реактивного програмування у своєму мобільному додатку. Обидві команди розробників Android і iOS використовують парадигму реактивного програмування, щоб спростити асинхронний, паралельний код у мобільних додатках [23].

Couchbase

Couchbase Server, раніше відомий як Membase — система керування базами даних, що належить до класу NoSQL-систем і надає схожі на Apache CouchDB засоби для створення документо-орієнтованих баз даних у поєднанні з Membase-подібними сховищами в форматі ключ-значення. При цьому, завдяки підтримці стандартного протоколу memcached, система залишається сумісною з великим числом вже наявних програм і може виступати у ролі прозорої заміни інших NoSQL-систем. Сирцевий код системи поширюється під ліцензією Apache.

Couchbase Server відрізняється високою масштабованістю і дозволяє організувати зберігання даних як на одному сервері, так і у формі розподіленої системи, що розміщає дані поверх групи серверів. У тому числі є вбудовані засоби для забезпечення високої доступності, самовідновлення в разі збою обслуговуючих сховище вузлів (дані можуть дублюватися на різних вузлах) і побудови сегментованих сховищ, копії яких рознесені по різних датацентрах і наближені до кінцевих користувачів. Підтримуються як односпрямовані («master-slave»), так і двонаправлені («master-master») режими реплікації. Підтримується створення первинних і вторинних індексів, а також індексів по декількох ключах. Для додаткової оптимізації продуктивності застосовуються вбудовані механізми кешування в оперативній пам'яті і засоби автоматичної генерації індексів.

Команда розробників Couchbase вважає, що асинхронні API-інтерфейси — це єдиний розумний спосіб отримати продуктивність і масштабованість, які часто потрібні, а також набагато простіше перейти від асинхронної сисеми до синхронної, ніж навпаки. Одна зі зрілих концепцій відома як Reactive Extensions, що походить з Microsoft і .NET. заснована на ідеї, що додатки повинні бути орієнтовані на події і реагувати на ці події асинхронно. Ця концепція визначає дуже багатий набір операторів для роботи з даними (змінювати, комбінувати, фільтрувати їх і т. д.) [24].

Огляд існуючих рішень показав, що дуже багато компаній використовує реактивне програмування для своїх продуктів, тож ця тема актуальна. Ба-

гато з компаній використовують саме Reactive Extentions для розробки. Але окрім Reactive Extentions для реактивних додатків, що розробляються на мові програмуваня Java актуальним буде використання Project Reactor. Тому було прийнято рішення сфокусуватися на вивченні переваг розробки реактивних додатків за допомогою Project Reactor.

РОЗДІЛ 2 ПРОЕКТУВАННЯ РЕАКТИВНИХ СИСТЕМ

2.1 Основні підходи у створенні реактивних систем

Реактивне програмування, яке не слід плутати з функціональним реактивним програмуванням, є підмножиною асинхронного програмування і парадигми, в якій доступність нової інформації просуває логіку вперед, а не керує потоком управління потоком виконання.

Воно підтримує розкладання проблеми на кілька окремих етапів, кожен з яких може бути виконаний асинхронним і неблокуючим чином, а потім скомпонований для створення робочого процесу — можливо, необмеженого за своїми входом [2].

Оксфордський словник визначає асинхронність як «неіснуючу або не виникаючу одночасно», що в даному контексті означає, що обробка повідомлення або події відбувається в довільний час, можливо, в майбутньому. Це дуже важливий метод в реактивному програмуванні, так як він дозволяє виконувати неблокуючі дії, де потоки виконання, конкуруючі за загальний ресурс, не повинні чекати блокування. Потік може виконувати іншу корисну роботу, поки ресурс зайнятий.

Синхронна система, що блокує зв'язок (Рис. 5 ліворуч) неефективна з точки зору ресурсів і легко обмежується. Реактивний підхід (Рис. 5 праворуч) знижує ризик, зберігає цінні ресурси і вимагає менше обладнання / інфраструктури.

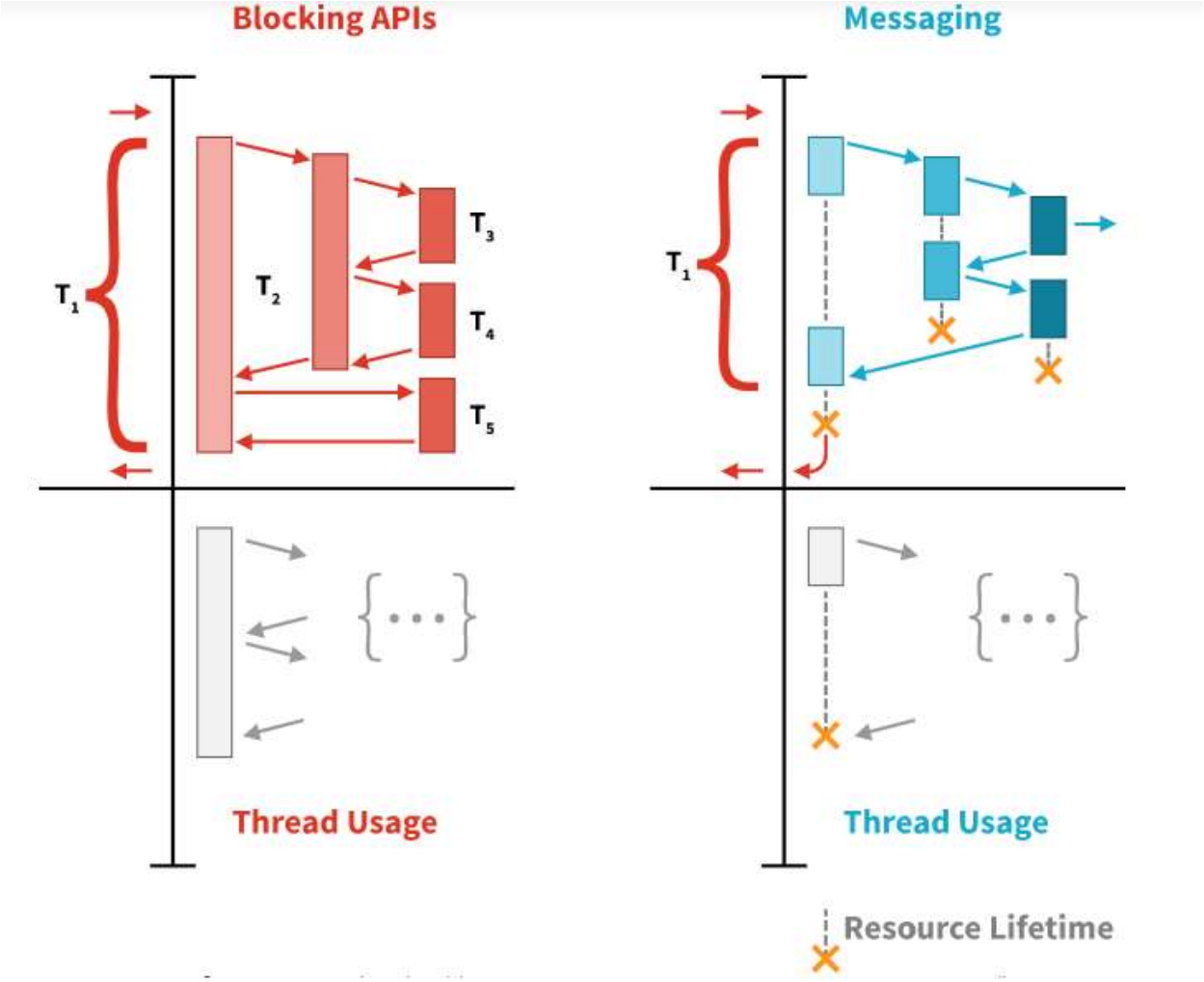

Рис. *5 Синхроннна та асинхронна системи*

Реактивне програмування зазвичай управляється подіями, на відміну від реактивних систем, які управляються повідомленнями .

Інтерфейс прикладного програмування (API) для бібліотек реактивного програмування зазвичай заснований на зворотньому виклику, тобто коли анонімні зворотні виклики з побічними ефектами приєднуються до джерел подій і викликаються, коли події відбуваються через ланцюжок потоку даних.

Було б розумно стверджувати, що реактивне програмування пов'язане з програмуванням потоку даних, оскільки акцент робиться на потік даних, а не на потік управління.

Приклади абстракцій програмування, які підтримують цю техніку програмування наведені далі.

Ф'ючерси / Обіцянки - контейнери з одним значенням, семантикою багаторазового читання / однократного запису, в які можна додавати асинхронні перетворення значення.

Потоки — як в Reactive Streams: необмежені потоки обробки даних, що забезпечують асинхронні, неблокуючі, зворотні канали перетворення зі зворотним тиском між безліччю джерел і місць призначення [15].

Змінні потоку даних — окремі змінні присвоювання (осередки пам'яті), які можуть залежати від введення, процедур та інших осередків так, що зміни автоматично оновлюються. Практичним прикладом є електронні таблиці, в яких зміна значення в одній клітинці, яка входить до діапазону застосування певної формули поширюється по всім залежним функціям, створюючи нові значення «вниз за течією».

Популярні бібліотеки, що підтримують методи реактивного програмування на JVM реалізують специфікацію Reactive Streams, яка є стандартом взаємодії між бібліотеками Reactive Programming в JVM, і відповідно до свого власного опису це «... ініціатива з надання стандарту для асинхронної обробки потоку з неблокуючим зворотним тиском» [15].

Як згадувалося раніше, реактивне програмування, що фокусується на обчисленнях через ефемерні ланцюжки потоків даних, зазвичай орієнтоване на події, в той час як реактивні системи, орієнтуються на стійкість і еластичність за допомогою зв'язку і координації розподілених систем, орієнтованих на повідомлення.

Основна відмінність між системою, керованою повідомленнями, з довгоживучими адресованими компонентами, і моделлю, керованої подіями, заснованою на потоці даних, полягає в тому, що повідомлення по своїй суті спрямовані, а події – ні [2]. Повідомлення мають чітке єдине призначення, в той час як події є фактами для інших подій. Крім того, обмін повідомленнями переважно є асинхронним.

У той час як кероване подіями спілкування використовує підхід «мильниці», передаючи факти (події), які інші можуть спостерігати (якщо вони слухають), кероване повідомленнями спілкування має адресного одержувача і одну мету.

Словник в Реактивному Маніфесті визначає концептуальне розходження [1]. Повідомлення — це елемент даних, який відправляється в певне місце призначення. Подія — це сигнал, що випромінюється компонентом при досягненні заданого стану. В керованій повідомленнями системі адресовані одержувачі очікують прибуття повідомлень і реагують на них, в іншому випадку вони не діють. В керованії подіями системі слухачі повідомлень прикріплюються до джерел подій так, що вони викликаються при відправці події. Це означає, що керована подіями система фокусується на адресованих джерелам подіях, тоді як керована повідомленнями система концентрується на адресованих одержувачах.

Повідомлення необхідні для зв'язку по мережі і утворюють основу для зв'язку в розподілених системах, в той час як події, з іншого боку, випромінюються локально. Зазвичай використовується обмін повідомленнями всередині системи, щоб зв'язати керовану подіями систему через мережу, відправляючи події всередині повідомлень. Це дозволяє підтримувати відносну простоту моделі програмування, керованої подіями, в розподіленому контексті і може дуже добре працювати для спеціалізованих і добре обмежених варіантів використання (наприклад, AWS Lambda, продукти обробки розподіленого потоку, такі як Spark Streaming, Flink, Kafka і Akka Streams через Gearpump і розподілену публікацію підписки продуктів, таких як Kafka і Kinesis).

Однак є компроміс, адже те, що людина отримує в абстракції і простоті моделі програмування, людина втрачає в плані контролю.

Обмін повідомленнями змушує нас приймати до уваги реальність і обмеження розподілених систем, такі як часткові збої, виявлення збоїв, відкинуті / дубльовані / упорядкувані повідомлення, можливу узгодженість, управління декількома паралельними реальностями, тощо і вирішувати їх в

лоб, замість того, щоб ховати їх за неглибокими абстракціями, вдаючи, що мережі немає.

Ці відмінності в семантиці і застосовності мають глибокі наслідки в дизайні додатку, включаючи такі речі, як пружність, еластичність, мобільність, прозорість розташування і управління складністю розподілених систем.

У реактивної системі, особливо тієї, яка використовує реактивне програмування, будуть присутні як події, так і повідомлення, оскільки одне є відмінним інструментом для комунікації (повідомлення), а інше — відмінним способом представлення фактів (подій) [3].

Відмовостійкість пов'язана зі швидкими відгуками при збої і є невід'ємною функціональною властивістю системи, те, що потрібно для розробки, а не те, що може бути додано згодом [1].

Стійкість виходить за рамки відмовостійкості, мова йде про можливість повного відновлення після збою — самозцілення.

Це вимагає ізоляції компонентів і стримування збоїв, щоб уникнути поширення збоїв на сусідні компоненти, що часто призводить до катастрофічних каскадних сценаріїв збоїв.

Таким чином, ключем до створення відмовостійких самовідновлюючихся систем є забезпечення того, щоб збої містилися, пересилалися як повідомлення, відправлялися іншим компонентам (які діють як супервізори) і управлялися з безпечного контексту поза відмовившого компоненту. У даному випадку керованість повідомленнями - це спосіб: відійти від сильно пов'язаних, тендітних, глибоко вкладених ланцюжків синхронних викликів, через які кожен буде страждати або ігноруватися. Ідея полягає в тому, щоб відокремити управління збоями від ланцюжка викликів, звільняючи клієнта від відповідальності за обробку збоїв сервера.

Еластичність - це здатність до відгуків навіть під навантаженням, це означає, що пропускна здатність системи збільшується або зменшується, а також включаються або відключаються вузли / машини в центрі обробки даних, оскільки ресурси пропорційно додаються або видаляються. Це важливий

елемент, необхідний для того, щоб дозволити системам бути ресурсоефективними, економічно ефективними, екологічними і платними [1].

Системи повинні бути адаптивними - дозволяти здійснювати автоматичне масштабування без втручання користувача, реплікації стану і поведінки, розподіл навантаження при обміні даними, відпрацювання відмови і поновлення, і все це без переписування або навіть переналаштування системи. Це забезпечує прозорість розташування, а саме можливість масштабувати систему однаковим чином, використовуючи одні й ті ж програмні абстракції, з однаковою семантикою, у всіх вимірах масштабу - від ядер ЦП до центрів обробки даних [3].

Як говорить Реактивний Маніфест, одним з ключових моментів, який дуже сильно спрощує цю проблему, є усвідомлення того, що ми всі займаємося розподіленими обчисленнями [1]. Це вірно незалежно від того, чи виконуємо ми наші системи на одному вузлі (з декількома незалежними ЦП, що обмінюються даними по каналу QPI) або на кластері вузлів (з незалежними машинами, що обмінюються даними по мережі). Ухвалення цього факту означає, що не існує концептуальної різниці між масштабуванням по вертикалі на багатоядерних чи горизонтально на кластері.

Це поділ в просторі, дозволений за допомогою асинхронної передачі повідомлень, і поділ примірників часу виконання від їх посилань - це те, що називається прозорістю розташування.

Тому незалежно від того, де знаходиться одержувач, ми спілкуємося з ним однаково. Єдиний спосіб, який можна зробити семантично еквівалентним, - це обмін повідомленнями.

2.2 Побудова реактивних архітектур

Реактивне програмування — це відмінний метод для управління внутрішньою логікою і перетворенням потоків даних, локально всередині компонентів, як спосіб підвищення рівня абстракції коду, продуктивності і ефективно-

сті використання ресурсів. Реактивні системи, будучи набором архітектурних принципів, роблять упор на розподілений зв'язок і дають інструменти для забезпечення стійкості і еластичності в розподілених системах.

Одна з поширених проблем, пов'язаних тільки з використанням реактивного програмування, полягає в тому, що його тісний зв'язок між етапами обчислень в керованій подіями заснованій на зворотньому виклику або декларативною програмою робить досягнення стійкості більш важким, оскільки їх ланцюги перетворення часто ефемерні, а їх етапи складаться зі зворотніх викликів або комбінаторів і є анонімними.

Оскільки ускладнюється досягнення відновлення окремих етапів, оскільки зазвичай неясно, де слід поширювати виключення або, чи потрібно їх поширювати, то в результаті збої пов'язані з ефемерними клієнтськими запитами, а не із загальним станом компонента, якщо один з етапів ланцюжка потоку даних завершується невдало, то необхідно перезапустити весь ланцюжок і повідомити клієнта. Це відрізняється від керованої повідомленнями реактивної системи, яка здатна до самовідновлення без необхідності повідомляти клієнта.

Інший контраст з підходом Reactive Systems полягає в тому, що чисте Reactive Programming дозволяє роз'єднувати в часі, але не в просторі (якщо не використовувати передачу повідомлень для розподілу графа потоку даних всередині мережі, як обговорювалося раніше) [4].

Роз'єднання у часі допускає паралелізм, але це розв'язання в просторі, яке забезпечує розподіл і мобільність, допускаючи не тільки статичні, але й динамічні топології, що є істотним для еластичності.

Відсутність прозорості розташування ускладнює адаптивне гнучке масштабування програми, заснованої виключно на методах реактивного програмування, і, отже, вимагає накладення зверху додаткових інструментів, таких як шина повідомлень, сітка даних або спеціальні мережеві протоколи. Саме тут простежується заснований на повідомленнях підхід Reactive Systems, оскільки це комунікаційна абстракція, яка підтримує свою модель

програмування і семантику у всіх вимірах масштабу і, отже, зменшує складність системи і когнітивні витрати [4].

Зазвичай згадувана проблема програмування на основі зворотних викликів полягає в тому, що, хоча написання таких програм може бути порівняно простим, воно може мати реальні наслідки в довгостроковій перспективі.

Наприклад, системи, засновані на анонімних зворотних викликах, дають дуже мало інформації, коли потрібно міркувати про них, підтримувати їх чи, найголовніше, з'ясувати, де і чому відбуваються збої в роботі і що спричиняє неправильну поведінку.

Бібліотеки та платформи, розроблені для Reactive Systems (такі як проект Akka і платформа Erlang), давно засвоїли цей урок і покладаються на довговічні адресовані компоненти, які легше масштабувати в майбутньому. Завдяки концепції адресованих повідомлень, що лежить в основі компонентної моделі, рішення з моніторингу мають ефективний спосіб представлення зібраних даних, використовуючи поширювані ідентифікатори [4].

Вибір хорошої парадигми програмування, яка забезпечує такі речі, як адресування і управління відмовами, виявився неоціненним у виробництві, так як він розроблений з урахуванням суворої реальності, щоб очікувати і приймати невдачу, а не втрачену причину спроби щоб запобігти цьому.

В цілому, реактивне програмування — дуже корисний метод реалізації, який може використовуватися в реактивній архітектурі. Це допоможе керувати тільки однією частиною історії, а саме — управляти потоком даних за допомогою асинхронного і неблокуючого виконання, зазвичай тільки в межах одного вузла або служби. При наявності декількох вузлів необхідно почати замислюватися про такі речі, як узгодженість даних, обмін даними між вузлами, координація, управління версіями, оркестрації, управління збоями, поділ інтересів.

Хоча в Reactive Programming основна увага приділяється асинхронному неблокуючому управлінню потоками даних між одним вузлом або службою,

складної архітектури Reactive System, потрібно набагато більше для успішного розгортання кількох служб у вузлах і кластерах [3].

Тому, щоб максимізувати цінність реактивного програмування, потрібно використовувати його як один з інструментів для створення реактивної системи. Побудова реактивної системи вимагає абстрагування ресурсів, специфічних для ОС, додавання асинхронних API і автоматичних вимикачів на існуючому і, можливо, застарілому програмному стеку. Слід враховувати той факт, що при створенні розподіленої системи, що складається з декількох сервісів, всі вони повинні працювати разом, забезпечуючи узгоджене і швидке реагування не тільки тоді, коли все працює так, як очікувалося, але також перед обличчям збоїв і непередбачуваним навантаженням.

2.3 Продуктивність реактивних систем

Оскільки більшість систем за своєю природою є складними, одним з найбільш важливих аспектів є забезпечення того, щоб системна архітектура приводила до мінімального зниження продуктивності як при розробці, так і при обслуговуванні компонентів, в той же час зменшуючи випадкову складність до мінімуму.

Це важливо, оскільки протягом життєвого циклу системи, якщо вона не спроектована належним чином, її обслуговування стає все важче і важче, потрібно все більше часу і зусиль для локалізації та усунення проблем [4].

Реактивні системи являють собою найбільш продуктивну системну архітектуру в контексті багатоядерних, хмарних і мобільних архітектур [4].

Ізоляція збоїв забезпечує перегородки між компонентами, запобігаючи каскадні збої і обмежуючи обсяг і серйозність збоїв.

Ієрархії супервізора пропонують кілька рівнів захисту в поєднанні з можливостями самовідновлення, що виключає безліч перехідних збоїв через будь-які експлуатаційні витрати.

Передача повідомлень і прозорість розташування дозволяють переводити компоненти в автономний режим і замінювати або перенаправляти їх, не впливаючи на роботу кінцевого користувача. Це знижує вартість збоїв, їх відносну терміновість, а також ресурси, необхідні для діагностики та усунення.

Реплікація знижує ризик втрати даних і зменшує вплив збою на доступність пошуку і зберігання інформації.

Еластичність дозволяє зберігати ресурси при коливаннях використання, що дозволяє мінімізувати експлуатаційні витрати при низькому навантаженні і мінімізувати ризик перебоїв або термінових інвестицій в масштабованість при збільшенні навантаження.

Таким чином, Reactive Systems дозволяє створювати системи, які добре справляються зі збоями, змінним навантаженням і все це при низькій вартості володіння з плином часу [4].

2.4 Швидка передача даних

Швидкий потік даних (обробка розподіленого потоку), з точки зору користувача, зазвичай управляється подіями з використанням локальних і потокових абстракцій, які розкривають API кінцевого користувача, які покладаються на конструкції реактивного програмування, такі як функціональні комбінатори і зворотні виклики.

Потім, під API кінцевого користувача, зазвичай використовується передача повідомлень і принципи проміжних вузлів реактивних систем, що підтримують розподілену систему етапів обробки потоків, довготривалих журналів подій, протоколів реплікації, хоча ці частини зазвичай не піддаються впливу розробника. Це хороший приклад використання реактивного програмування на рівні користувача і реактивних систем на рівні системи [19].

2.5 Мікросервіси

Архітектура на основі мікросервісов – це проектування системи автономних розподілених сервісів, часто з хмарою в якості призначеної платформи розгортання, з виграшем у цінності, що забезпечується застосуванням Reactive.

Проектування реактивних систем використовується між мікросервісами, дозволяючи створювати системи мікросервісів, які грають за правилами розподілених систем. Реагування через стійкість і еластичність стало можливим завдяки керованості повідомленнями [25].

Архітектурний стиль мікросервісов — це підхід, при якому єдиний додаток будується як набір невеликих сервісів, кожен з яких працює у власному процесі і веде комунікацію з іншими, використовуючи прості механізми. Ці сервіси побудовані навколо бізнес-потреб і розгортаються незалежно з використанням повністю автоматизованого середовища. Існує абсолютний мінімум централізованого управління цими сервісами. Самі по собі ці сервіси можуть бути написані на різних мовах і використовувати різні технології зберігання даних [25].

Протилежністю мікросервісов є моноліт (monolithic style) — додаток, побудований як єдине ціле. Enterprise додатки часто включають три основні частини: призначений для користувача інтерфейс (що складається в основному з HTML сторінок і javascript-а), база даних (як правило реляційної, з безліччю таблиць) і сервер. Серверна частина обробляє HTTP запити, виконує доменну логіку, запитує і оновлює дані в БД, заповнює HTML сторінки, які потім відправляються браузеру клієнта. Будь-яка зміна в системі призводить до перезбірки і перерозгортання нової версії серверної частини програми.

Монолітний сервер досить очевидний спосіб побудови подібних систем. Вся логіка по обробці запитів виконується в єдиному процесі, при цьому

можна використовувати можливості будь-якої мови програмування для поділу додатку на класи, функції і простори імен. Можна запускати і тестувати додаток на машині розробника і використовувати стандартний процес розгортання для перевірки змін перед публікацією їх в продакшн. Можна масштабувати монолітні додатки горизонтально шляхом запуску декількох фізичних серверів із балансувальником навантаження.

Монолітні додатки можуть бути успішними, але все більше людей розчаровуються в них, особливо в світлі того, що все більше додатків розгортаються в хмарі. Будь-які зміни, навіть найменші, вимагають перезборки і розгортання всього моноліту. З плином часу, стає важче зберігати хорошу модульну структуру, зміни логіки одного модуля мають тенденцію впливати на код інших модулів. Масштабувати доводиться весь додаток цілком, навіть якщо це потрібно тільки для одного модуля цього додатка.

Ці незручності призвели до архітектурного стилю мікросервісів — побудови додатків у вигляді набору сервісів. На додаток до можливості незалежного розгортання і масштабування кожен сервіс також отримує чітку фізичну межу, яка дозволяє різним сервісам бути написаними на різних мовах програмування. Вони також можуть розроблятися різними командами [25].

Не можна стверджувати, що стиль мікросервісов це інновація. Його коріння сягає далеко в минуле, як мінімум до принципів проектування, використаним в Unix. Але тим не менш, недостатньо людей беруть до уваги цей стиль, хоча багато програм отримають переваги якщо почнуть застосовувати цей стиль.

2.6 Властивості архітектури мікросервісів

Розбиття через сервіси

Довгий час серед розробників спостерігається бажання будувати системи шляхом з'єднання разом різних компонентів, багато в чому так само, як це відбувається в реальному світі. За останні пару десятків років був поміче-

ний великий ріст набору бібліотек, використовуваних в більшості мов програмування [25].

Говорячи про компоненти, слід було б дати визначення цьому терміну. Компонент - це одиниця програмного забезпечення, яка може бути незалежно замінена або оновлена.

Архітектура мікросервісів використовує бібліотеки, але їх основний спосіб розбиття додатку - ділення його на сервіси. Визначимо бібліотеки як компоненти, які підключаються до програми і викликаються нею в тому ж процесі, в той час як сервіси — це компоненти, що виконуються в окремому процесі і спілкуються між собою через веб-запити або remote procedure call (RPC) [25].

Головна причина використання сервісів замість бібліотек — це незалежне розгортання. Якщо розробляти додаток, що складається з декількох бібліотек, які працюють в одному процесі, то будь-яка зміна в цих бібліотеках призводить до перерозгортання всієї програми. Але, якщо додаток розбито на кілька сервісів, то зміни, що зачіпають будь-який з них, потребуватимуть перерозгортання тільки зміненого сервісу [25]. Зрозуміло, що певні зміни будуть зачіпати інтерфейси, які, в свою чергу, потребують певної координації між різними сервісами, але метою правильної архітектури мікросервісів є мінімізація необхідності в такій координації шляхом установки правильних кордонів між мікросервісами.

Інший наслідок використання сервісів, як компонентів — більш явний інтерфейс між ними. Більшість мов програмування не мають хорошого механізму для оголошення Published Interface. Часто, тільки документація і дисципліна запобігають порушенню інкапсуляції компонентів. Сервіси дозволяють уникнути цього через використання явного механізму віддалених викликів.

Тим не менше, використання сервісів подібним чином має свої недоліки. Дистанційні виклики працюють повільніше, ніж виклики в рамках процесу, і тому API повинен бути менш деталізованим (coarser-grained), що часто

призводить до незручності у використанні. Якщо вам потрібно змінити набір відповідальностей між компонентами, зробити це складніше через те, що вам потрібно перетинати кордони процесів.

У першому наближенні ми можемо спостерігати, що сервіси співвідносяться з процесами як один до одного. Насправді сервіс може містити безліч процесів, які завжди будуть розроблятися і розвиватися разом.

Організація навколо потреб бізнесу

Коли великі додатки розбиваються на частини, часто менеджмент фокусується на технологіях, що призводить до утворення UI команди, серверної команди і БД команди. Коли команди розбиті подібним чином, навіть невеликі зміни забирають багато часу через необхідність крос-командної взаємодії. Це призводить до того, що команди розміщують будь-яку логіку на тих шарах, до яких мають доступ [25].

Закон Конвея (Conway's Law) в дії : «Будь-яка організація, яка проектує якусь систему (в широкому сенсі) отримає дизайн, чия структура копіює структуру команд в цій організації» (Рис. 6).

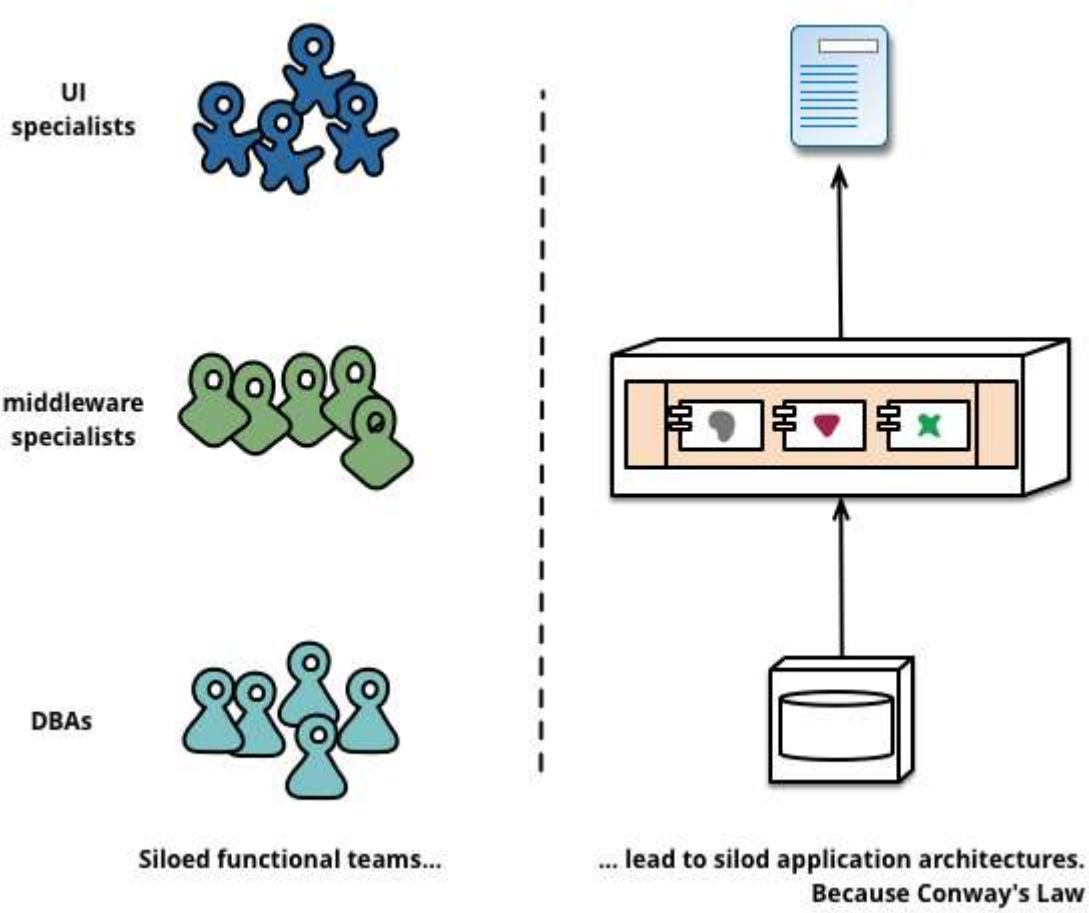

Рис. *6 Закон Конвея (Conway's Law) в действии*

Мікросервісний підхід до розбиття має на увазі розбиття на сервіси відповідно до потреб бізнесу. Такі сервіси включають в себе повний набір технологій, необхідних для цієї бізнес-потреби. Це призводить до формування крос-функціональних команд, що мають повний набір необхідних навичок: user-experience, бази даних і project management [25].

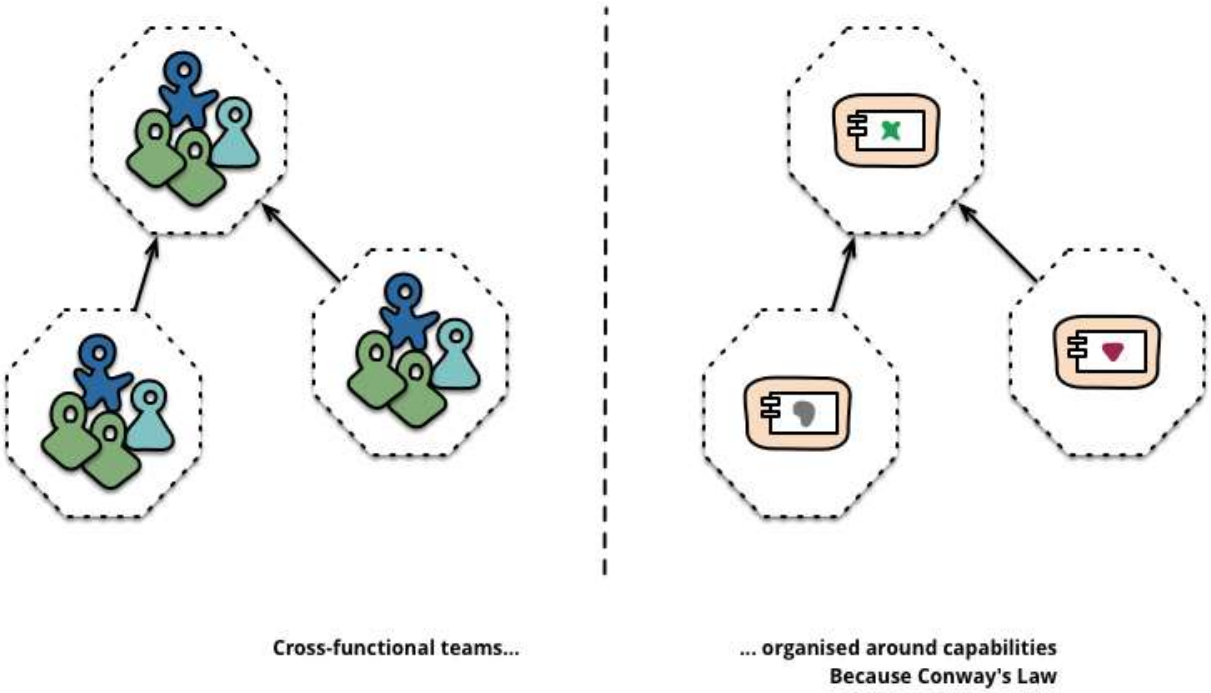

Рис. *7 Сервісні кордони, підкріплені межами команд*

Великі монолітні додатки теж можуть бути розбиті на модулі навколо бізнес потреб, хоча зазвичай цього не відбувається. Безумовно, великим командам рекомендується будувати монолітні додатки саме таким чином. Основна проблема тут в тому, що такі додатки мають тенденцію до організації навколо занадто великої кількості контекстів. Якщо моноліт охоплює безліч контекстів, окремим членам команд стає занадто складно працювати з ними через їх великий розмір. Крім того, дотримання модульних кордонів в монолітному додатку потребує суттєвої дисципліни [25]. Коли межі компонентів мікросервісів окреслені явно, то підтримка цих кордонів спрощується.

Продукти, а не проекти

Більшість компаній з розробки ПЗ використовують проектну модель, в якій метою є розробка якоїсь частини функціональності, яка після цього вважається завершеною. Після завершення ця частина передається команді підтримки і проектна команда розпускається.

Прихильники мікросервісів стверджують, що команда повинна володіти продуктом протягом усього терміну його життя [25]. Це призводить до того, що розробники регулярно спостерігають за тим, як їх продукт поводиться в продакшн, і більше контактують з користувачами, тому що їм доводиться брати на себе як мінімум частину обов'язків по підтримці.

Мислення в термінах продукту встановлює зв'язок з потребами бізнесу. Продукт — це не просто набір функціоналу, який необхідно реалізувати, це постійні відносини, мета яких —- допомогти користувачам збільшити їх бізнес-можливості [25].

Звичайно, цього можна також досягти і у випадку з монолітним додатком, але висока незалежність сервісів спрощує установку персональних відносин між розробниками сервісу і його користувачами.

Децентралізоване управління

Одним із наслідків централізованого управління є тенденція до стандартизації використовуваних платформ. Розбиваючи моноліт на сервіси, є вибір, як побудувати кожен з них. Звичайно, тільки тому що ви можете робити щось, не означає, що ви повинні це робити. Але розбиття системи подібним чином дає вам можливість вибору [25].

Команди, які розробляють мікросервіси, також вважають кращим інший підхід до стандартизації. Замість того, щоб використовувати набір визначених стандартів, написаних кимось, вони надають перевагу ідеї побудови корисних інструментів, які інші розробники можуть використовувати для вирішення схожих проблем. Ці інструменти, як правило, виокремлюють з коду одного з проектів і розшарюють між різними командами, іноді використовуючи при цьому модель внутрішнього open source. Тепер, коли git і github стали де-факто стандартною системою контролю версій, open source практики стають все більш і більш популярними у внутрішніх проектах компаній.

Крім децентралізації прийняття рішень про моделювання предметної області, мікросервіси також сприяють децентралізації способів зберігання

даних. У той час як монолітні додатки схильні до використання єдиної БД для зберігання даних, компанії часто воліють використовувати єдину БД для цілого набору додатків. Такі рішення, як правило, викликані моделлю ліцензування баз даних. Мікросервіси вважають за краще давати можливість кожному сервісу керувати власною базою даних для того, щоб створювати окремі екземпляри загальної для компанії СУБД і використовувати нестандартні види баз даних. Цей підхід називається Polyglot Persistence [25]. Ви також можете застосовувати Polyglot Persistence в монолітних додатках, але в мікросервісах такий підхід зустрічається частіше.

Децентралізація відповідальності за дані серед мікросервісов впливає на те, як ці дані змінюються. Звичайний підхід до зміни даних полягає в використанні транзакцій для гарантування консистентності при зміні даних, що знаходяться на декількох ресурсах.

2.7 Spring Framework 5 та мікросервіси

В Spring Framework 5 Reactive Streams укладений як контракт на передачу backpressure між асинхронними компонентами і бібліотеками [26].

Reactive Streams визначають специфікацію API, яка містить мінімальний набір інтерфейсів, які надають способи визначення операцій і сутностей для асинхронних потоків даних з неблокуючим зворотним тиском [15].

З введенням зворотного тиску Reactive Streams дозволяє передплатнику контролювати швидкість обміну даними від видавців.

API реактивних потоків офіційно є частиною Java 9 як java.util.concurrent.Flow.

Реактивні потоки використовуються в основному як рівень взаємодії.

Spring Framework використовує Reactor для власної реактивної підтримки. Із вказаного вище, Reactor -— це реалізація Reactive Streams, яка додатково розширює базовий контракт Reactive Streams Publisher з типами API-

інтерфейсів, які можна комбінувати з Flux і Mono, щоб забезпечити декларативні операції з послідовностями даних 0..N і 0..1 [27].

Spring Framework надає Flux і Mono в багатьох своїх власних реагуючих API. Однак на рівні додатків, як завжди, Spring надає вибір і повністю підтримує використання RxJava.

Spring WebFlux Framework є частиною Spring 5 і забезпечує підтримку реактивного програмування для веб-додатків. Spring WebFlux внутрішньо використовує Project Reactor і його реалізації Publisher — Flux і Mono.

Новий каркас підтримує дві моделі програмування:

1. Анотація на основі реактивних компонентів.
2. Функціональна маршрутизація і обробка.

Spring став де-факто середовищем розробки для створення додатків на основі Java. За своєю суттю Spring заснований на концепції впровадження залежностей. У звичайному додатку Java програма розбивається на класи, де кожен клас часто має явні зв'язки з іншими класами в додатку. Зв'язки — це виклик конструктора класу прямо в коді. Як тільки код скомпільовано, ці точки прив'язки змінити не можна.

Це проблематично у великому проекті, тому що ці зовнішні зв'язки крихкі і внесення зміни може призвести до множинних впливів на інший код «нижче за течією». Інфраструктура впровадження залежностей, така як Spring, дозволяє більш легко управляти великими Java проектами шляхом екстерналізації відносин між об'єктами в додатку через угоду (і анотації). Spring виступає посередником між різними Java класами застосунку і управляє їх залежностями. Spring дозволяє зібрати свій код разом, як набір з'єднаних разом кубиків Lego.

Швидке включення функцій в Spring підвищило його корисність, і фреймворк швидко став легшою альтернативою для розробників корпоративних додатків Java.

На даний момент існує тенденція, яка полягає у тому, що багато команд розробників відходять від монолітних додатків, в яких логіка представ-

лення, бізнесу і доступу до даних упаковані разом і розгорнуті як єдиний артефакт. Замість цього команди переходять до високорозподілених моделей, в яких додатки будуються як невеликі розподілені сервіси які можна легко розгорнути в хмарі. У відповідь на це зрушення команда розробників Spring запустила два проекти: Spring Boot і Spring Cloud.

Spring Boot — це переосмислення інфраструктури Spring. Поки він охоплює ядро функцій Spring, Spring Boot виключає багато з «корпоративних» функцій, наявних в Spring, а замість цього надає фреймворк, орієнтований на Java та орієнтований на REST [27].

За допомогою декількох простих анотацій Розробник Java може швидко створити мікросервіс REST, який можна зібрати і розгорнути без потреби в зовнішньому контейнері додатків.

Оскільки мікросервіси стали одним з найбільш поширених архітектурних шаблонів для створення хмарних додатків співтовариство розробників Spring надало нам Spring Cloud. Платформа Spring Cloud спрощує введення в дію і розгортання мікросервісів в приватній або публічній хмарі. Spring Cloud охоплює кілька популярних фреймворків хмарного управління мікросервісами в рамках загальної структури і робить використання і розгортання цих технологій таким же простим, як анотування коду.

2.8 Project Reactor як реалізація специфікації Reactive Streams

Spring 5 Framework представила Reactor як реалізацію специфікації Reactive Streams. Reactor — це реактивна бібліотека нового покоління для створення неблокуючих додатків на JVM.

Reactor розширює базовий контракт Reactive Streams Publisher і визначає типи API Flux і Mono для забезпечення декларативних операцій з послідовностями даних 0..N і 0..1 відповідно [28].

Spring Web Reactive використовує пропозицію Servlet 3.1 для неблокуючим введення-виведення і працює в контейнерах Servlet 3.1.

Project Reactor - це бібліотека, яка реалізує модель реактивного програмування. Вона побудована на основі специфікації реактивних потоків (Reactive streams), стандарту для створення реактивних додатків.

Reactor — це повністю неблокуюча основа реактивного програмування для JVM з ефективним управлінням вимогами. Reactor безпосередньо інтегрується з функціональними API-інтерфейсами Java 8, зокрема, Completable Future, Stream і Duration. Він пропонує складові API-інтерфейси асинхронної послідовності — Flux (для елементів [N]) і Mono (для елементів [1]) - і широко реалізує специфікацію Reactive Streams [2].

Ядро Reactor — реактивні основи для додатків і середовищ, а також реактивні розширення, засновані на API з типами Mono (1 елемент) і Flux (n елементів).

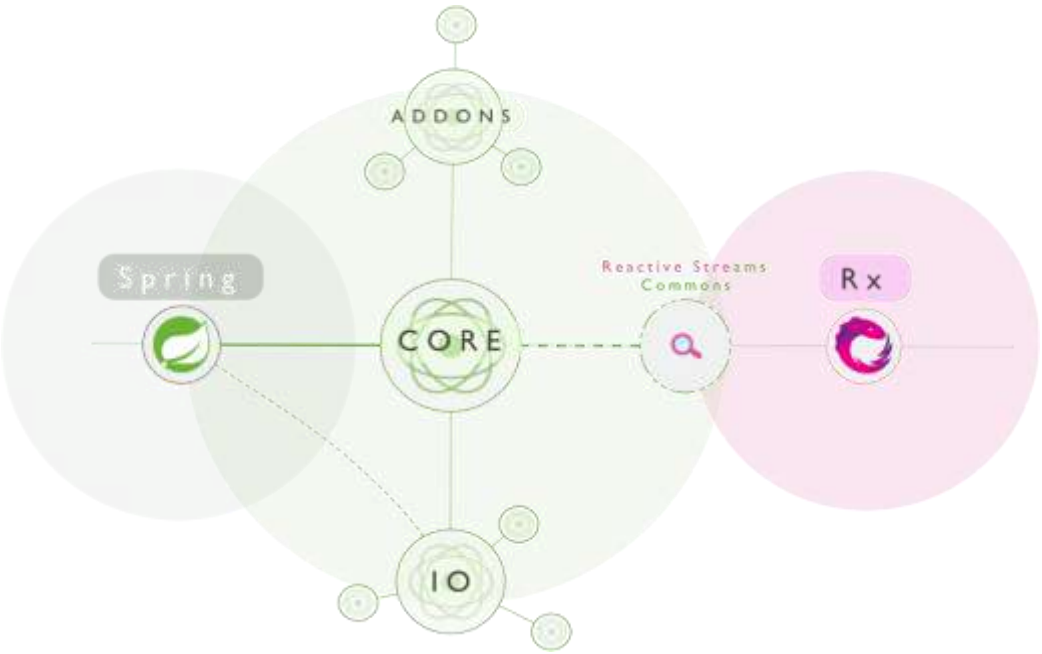

Рис. *8 Reactor core*

Reactor також підтримує не блокуючий міжпроцесовий зв'язок з проектом reactor-netty. Reactor Netty підходить для архітектури мікросервісів, мережевих backpressure-ready двигунів, для HTTP (включаючи Websockets), TCP та UDP. Реактивне кодування та декодування повністю підтримуються.

На вході у нього є один потік, який працює в нескінченному циклі. Завдяки селектору і канальному механізму, він перенаправляє дані з вхідних запитів у вхідні буфери і делегує обробку цих запитів виділеному пулу потоків асинхронних потоків. І також в зворотному напрямку [17].

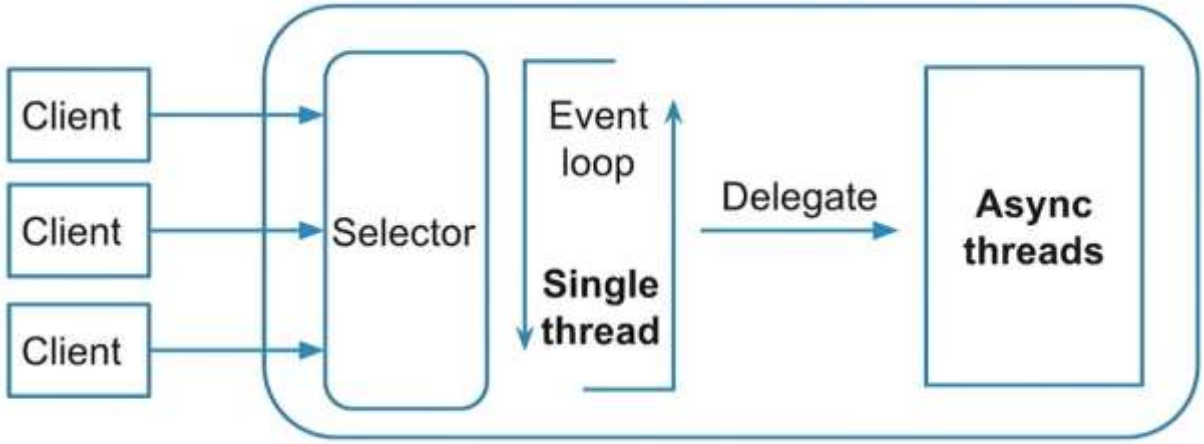

Рис. *9 Схема роботи Netty*

Flux і Mono

Flux - це імплементація інтерфейсу Publisher, представляє з себе послідовність з 0..N елементів, яка може (але не обов'язково) завершуватися (в т.ч. і з помилкою) [2].

У послідовності Flux (Рис. 10) є 3 допустимих значення: об'єкт послідовності, сигнал завершення або сигнал помилки (виклики методів onNext, onComplete і onError відповідно).

Кожне з 3 значень опціонально. Наприклад, Flux може представляти із себе нескінченну порожню послідовність (жоден метод не викликається), або кінцеву порожню послідовність (викликається тільки onComplete), або нескінченну послідовність значень (викликається тільки onNext).

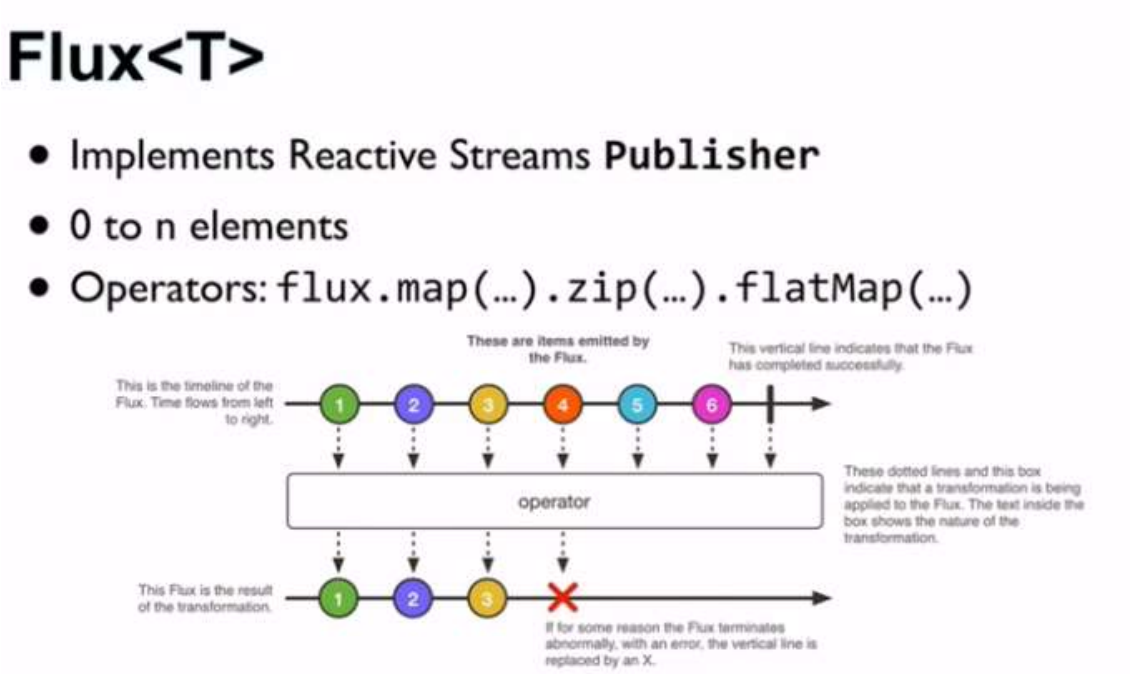

Рис. *10 Послідовність Flux*

Mono - це теж імплементація інтерфейсу Publisher (Рис. 11), вдає із себе якийсь асинхронний елемент або його відсутність Mono.empty (). На відміну від Flux, Mono може повернути не більше 1 елемента. Виклики onComplete () і onError (), як і в випадку з Flux, опціональні.

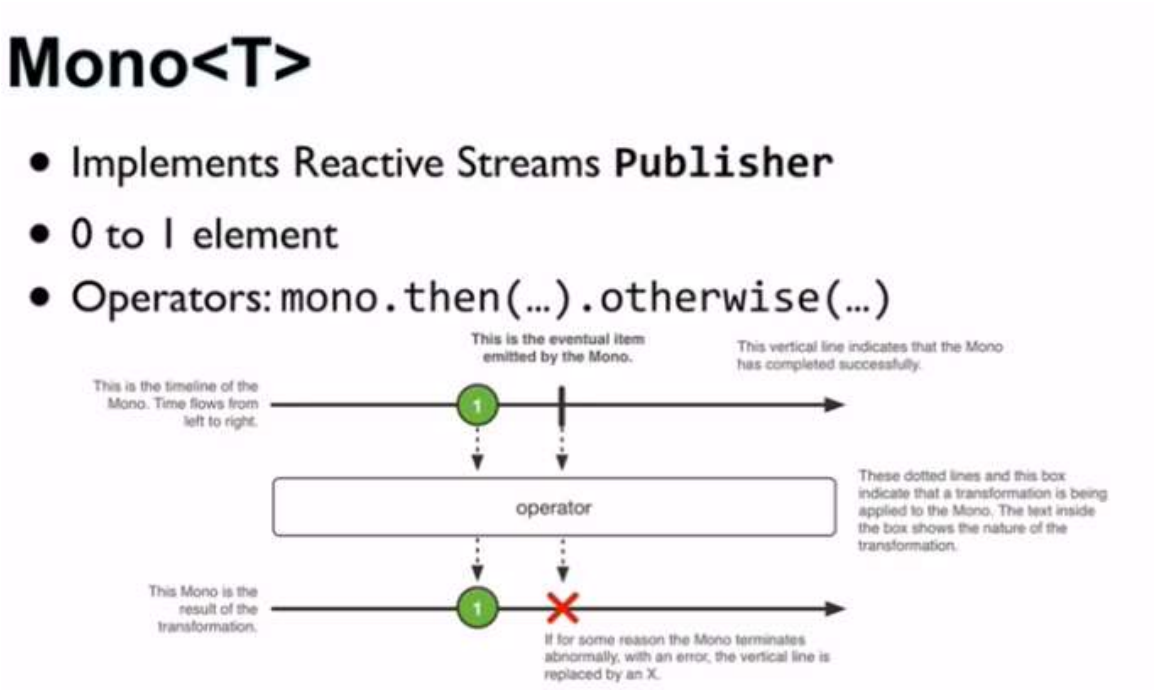

Рис. *11 Послідовність Mono*

Поділ на Flux і Mono допомагає поліпшити семантику реактивного API, роблячи його досить виразним, але не надмірним. Flux і Mono користуються своєю семантикою і перетікають один в одного. У Flux є метод single (), який повертає Mono <T>, а у Mono є метод concatWith (Mono <T>), який повертає вже Flux <T>.

Також у них є унікальні оператори. Деякі мають сенс тільки при N елементах в послідовності (Flux) або, навпаки, актуальні тільки для одного значення.

Принцип роботи

В офіційній документації Reactor порівнюється з конвеєром [28].

Flux і Mono реалізують Publisher інтерфейс з специфікації Reactive Streams.

Publisher видає якісь дані (матеріали). Дані йдуть по ланцюжку з операторів (конвеєрній стрічці), обробляються, в кінці виходить готовий продукт, який передається в потрібний Consumer / Subscriber і вживається вже там.

Оператор - це якийсь Publisher, який крім якоїсь своєї логіки містить посилання на вихідний Publisher, про якого йдеться. Виклики операторів створюють ланцюжок з Publisher.

Реактивне програмування виникло через бажання писати асинхронний неблокований код в читабельному вигляді. Ні код, написаний на зворотніх викликах, ні код, написаний за допомогою CompletableFuture, не може бути настільки легким для читання, як цього можна досягти за допомогою реактивності.

В основі підходу лежить ідея поділу компонентів на 2 типи: джерело подій (Publisher) і обробник подій (Subscriber) [2].

Subscriber підписується на події, які створює Publisher, а потім якимось чином їх обробляє. По суті це патерн Observer з надбудованими поверх можливостями та особливостями.

Існує ще одне поняття - Observer.

Він може підписатися на подію об'єкта і виконувати будь-які дії з отриманим результатом.

У одного Subject може бути багато передплатників.

Спілкування між Publisher і Subscriber відбувається через об'єкт Subscription (Рис. 12).

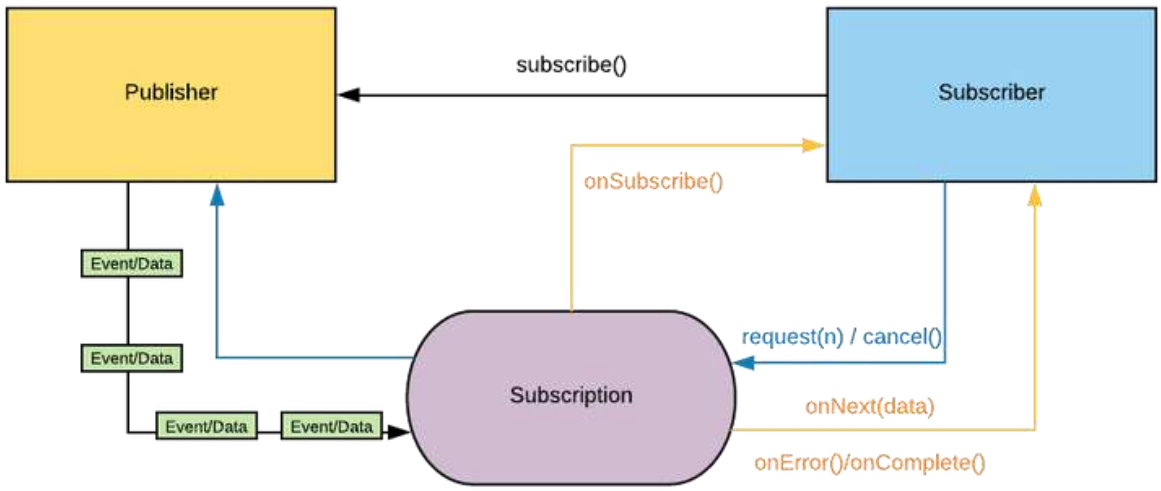

Рис. *12 Взаємодія Publisher і Subscriber*

Subcriber може регулювати швидкість поставки повідомлень від Publisher (тому що backpressure), а також скасовувати підписку.

Publisher'и можна об'єднувати в ланцюжки і комбінувати різними способами.

Інтерфейси Publisher, Subscriber і Subscription знаходяться в пакеті org.reactivestreams, який за замовчуванням був доданий в Java 9.

Вони задають специфікацію для власної реалізації реактивних потоків.

Бібліотека Project Reactor є такою реалізацією.

РОЗДІЛ 3 ПРОЕКТ ПРОГРАМНОЇ СИСТЕМИ

3.1 Архітектура системи

Основний контролер системи — Hub Controller є точкою входу в систему, він викликає сервіс Cards Service, у якому є список клієнтів, за рахунок яких відбувається «спілкування» між трьома модулями у системі. Сервіс проходить по списку клієнтів і з кожного клієнта збирає список потрібної інформації і передає кінцевому користувачеві.

Кожен клієнт у свою чергу звертається до відповідного мікросервісу. Кожен з мікросервісів має власний контролер, що є точкою входу в систему, і відслідковує запити. Ці контролери викликають відповідні сервіси, у яких описана бізнес-логіка, яка повинна бути ізольована від контролеру. Сервіси беруть дані через клієнтів, які звертаються до різних джерел (бази даних, віддаленого ресурсу та списку рекомендацій, що зберігається на девайсі), аби отримати потрібну інформацію.

Тобто кожен модуль системи складається з контролера, сервісу, клієнта та джерел інформації (Рис. 13).

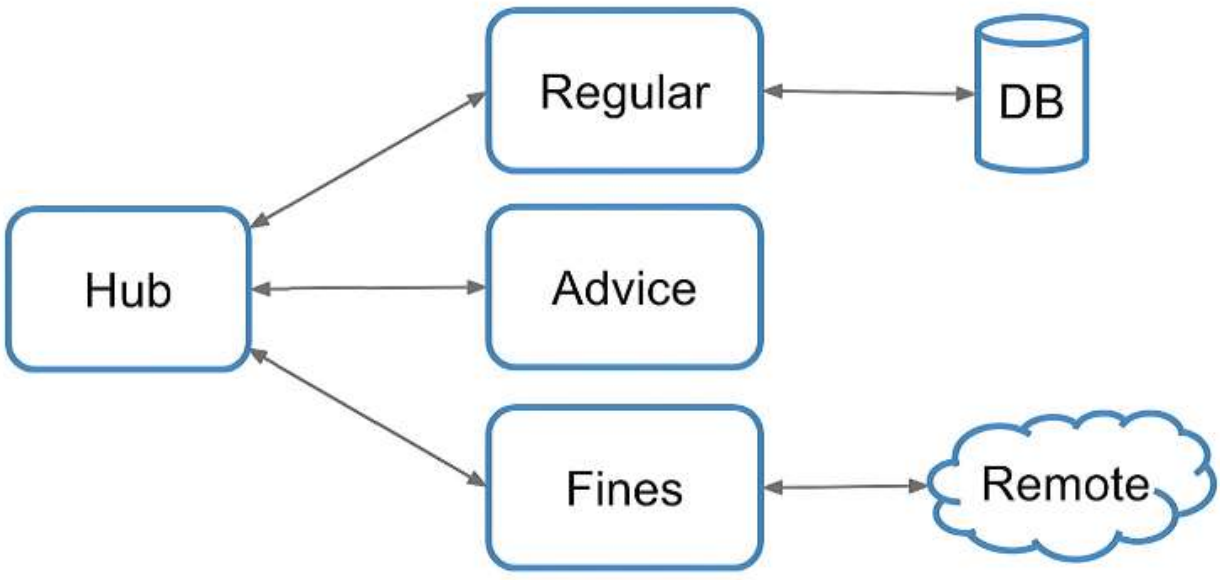

Рис. *13 Структура розробленої системи*

3.2 Засоби реалізації

Під час розробки програмної системи, через певні переваги, було використано наступні засоби:

1. Середовище розробки IntelliJ IDEA.
2. Мова програмування Java.
3. Фреймворк Spring 5 та WebFlux.
4. Бібліотека Project Reactor.
5. Мова програмуваня PHP.
6. Середовище розробки VS Code.
7. Система автоматичної збірки Gradle.

3.2.1 Середовище розробки IntelliJ IDEA

Під час розробки програмної системи було використано середовище розробки IntelliJ IDEA яке пристосоване для роботи з Java проєктами та застосунками. Основними перевагами даного програмного забезпечення є те, що воно надає всі важливі інструменти для розробки різноманітних застосунків. Тобто, можна не тільки використовувати це середовище в якості редактору Java коду, а й для роботи із базою даних, також існує можливість відслідковування дублікатів коду. Дане середовище підтримує велику кількість плагінів та фрейморків, що надає можливість без зайвих витрат редагувати будь-який проект з великим розмаїттям використаних технологій. Дуже зручним бонусом також є можливість перегляду XML, CSS, json та деяких інших файлів.

3.2.2 Мова програмування Java

Java (вимовляється Джава) — об'єктно-орієнтована мова програмування, випущена 1995 року компанією «Sun Microsystems» як основний компонент платформи Java. З 2009 року мовою займається компанія «Oracle», яка того року придбала «Sun Microsystems». В офіційній реалізації Java-програми

компілюються у байт-код, який при виконанні інтерпретується віртуальною машиною для конкретної платформи [29].

«Oracle» надає компілятор Java, а також віртуальну машину Java, які задовольняють специфікації Java Community Process, під ліцензією GNU General Public License.

Мова Java значно запозичила синтаксис із C та C++. Зокрема, за основу взято об'єктну модель C++, проте її було модифіковано. Було усунуто можливість появи деяких конфліктних ситуацій, що могли виникнути через помилки розробника та полегшено сам процес розробки об'єктно-орієнтованих програм. Ряд дій, які в C/C++ повинні здійснювати розробники було доручено віртуальній машині. Передусім Java розроблялась як платформонезалежна мова, тому вона має менше низькорівневих можливостей для роботи з апаратним забезпеченням, що в порівнянні, наприклад, з C++ зменшує швидкість запуску програми, але при якісно написаному коді Java може працювати так само швидко. На додаток, за необхідності низькорівневого доступу Java дозволяє викликати підпрограми, написані іншими мовами програмування.

Мова програмування Java вплинула на розвиток J++, що розроблялась компанією «Microsoft». Роботу над J++ було зупинено через судовий позов «Sun Microsystems», оскільки ця мова програмування була модифікацією мови Java. Пізніше в новій платформі «Microsoft» .NET випустили J#, щоб полегшити міграцію програмістів J++ або Java на нову платформу. З часом нова мова програмування C# стала основною мовою платформи, перейнявши багато чого з Java. J# востаннє включався в версію Microsoft Visual Studio 2005. Мова сценаріїв JavaScript має схожу із Java назву і синтаксис, але не пов'язана із Java.

3.2.3 Фреймворк Spring 5 та WebFlux

На жаль, методи реактивного програмування погано підтримувались у Spring Framework. Це стало обмеженням для розробників сучасних додатків і

знизило конкурентоспроможність фреймворка. Як наслідок зросла потреба в кардинальному поліпшенні фреймворка. Нарешті, додавання підтримки реактивності на всіх рівнях сприяло збільшенню привабливості Spring Фрамеворк і дало розробникам потужний інструмент для розробки реактивних систем. З цих причин розробники фреймворка вирішили реалізувати нові модулі, що концентрують у собі всю міць Spring Framework як основи для реактивних систем [30].

Багато чого змінилося з випуском Spring Framework 5 і нового проекту Reactive WebClient. Тепер завдяки підтримці WebClient з'явилася можливість використовувати неблокуючі методи взаємодії між службами. Також в Servlet 3.0 тепер є підтримка асинхронних взаємодій — клієнт-сервер, в Servlet 3.1 додана можливість неблокуючих записів в операціях введення / виводу, і в цілому нові асинхронні механізми в Servlet 3 прекрасно інтегровані в Spring.

Модуль Spring-Web-Reactive і Spring MVC мають багато спільних алгоритмів, але модуль Spring-Web-Reactive перевизначив багато контрактів Spring MVC, такі як HandlerMapping і HandlerAdapter, щоб зробити їх асинхронними і неблокуючих, а також включити реактивний HTTP-запит і відповідь (у вигляді RouterFunction і HandlerFunction).

Новий реактивний WebClient також був представлений в Spring 5 на додаток до існуючого RestTemplate.

HTTP-клієнти (наприклад, Reactor, Netty, Undertow), що підтримують реактивне програмування, адаптовані до набору реактивних абстракцій ClientHttpRequest і ClientHttpResponse, які представляють тіло запиту і відповіді як Flux <DataBuffer> з повною підтримкою зворотного тиску на стороні читання і запису.

Spring 5 Framework представив Reactor як реалізацію специфікації Reactive Streams. Також, Spring 5 представив платформу WebFlux, яка представляє собою повністю асинхронний і неблокуючий реактивний веб-стек, який дозволяє обробляти величезну кількість одночасних з'єднань [30].

Модуль WebFlux є альтернативою Spring MVC і являє собою реактивний підхід для написання веб-сервісів (Рис 14).

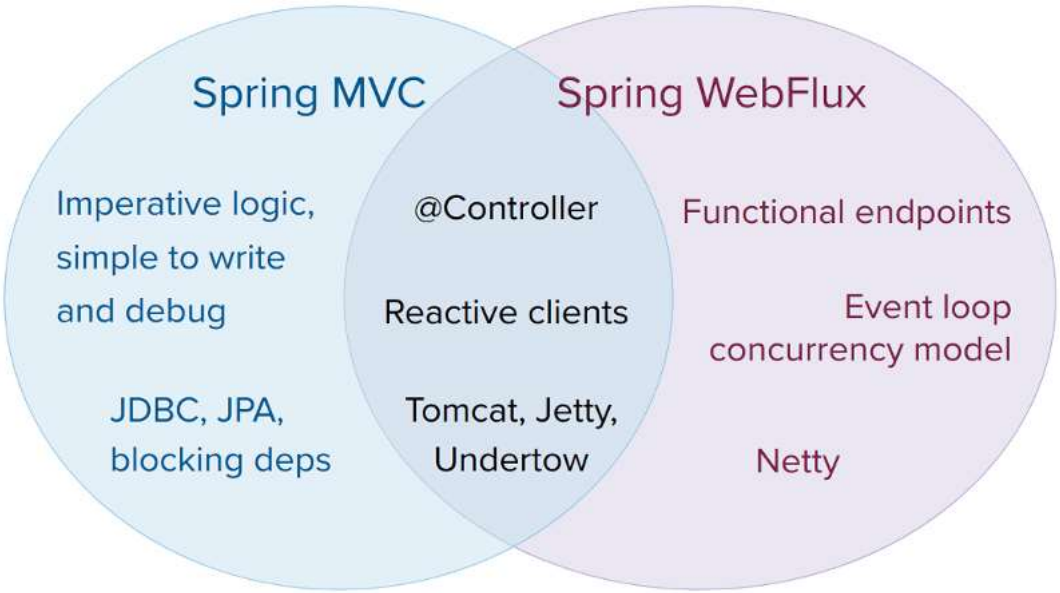

Рис. *14 Порівняння WebFlux та Spring MVC*

WebFlux позиціонує себе як мікрофреймворк. Цей новий мікрофреймворк підтримує анотовані контролери, функціональні кінцеві точки, WebClient (аналог RestTemplate в Spring Web MVC), WebSockets і багато іншого [31].

В основі WebFlux лежить бібліотека Reactor. WebFlux за замовчуванням використовує Netty.

3.2.4 Бібліотека Project Reactor

Бібліотека Reactor стрімко розвивалася з моменту її заснування, і в даний час це найсучасніша реактивна бібліотека.

С самого початку бібліотека Reactor створювалась з метою позбавити розробників від пекла зворотних викликів і глибоко вкладеного коду при створенні асинхронних конвеєрів. Головна мета бібліотеки – збільшити читабельність коду і забезпечити можливість компонування робочих процесів, визначаючи їх за допомогою інструментів з бібліотеки Reactor. Бібліотека проектувалася для досягнення максимальної ефективності при маніпулюван-

ні даними – як локальними, так і асинхронними запитами з можливими збоями в операціях введення / виводу. Саме з цієї причини оператори обробки помилок в Project Реактор такі гнучкі і допомагають писати відмовостійкий код [2].

Бібліотека Project Reactor проектувалася так, щоб не залежати від конкретного механізму паралельного виконання, тому в ній не використовуються ніякі моделі паралельного виконання. У той же час вона пропонує набір планувальників для управління потоками виконання практично будь-яким способом, і якщо жоден із запропонованих планувальників не відповідає наявним вимогам, то розробник може реалізувати свій планувальник з повноцінним низькорівневим управлінням [28].

3.2.5 Мова програмування PHP

PHP - це популярна мова програмування, особливо в середовищі веб-розробників. Автором початкової версії є Расмус Лердорф, ідея якого полягала в розробці набору інструментів для спрощення процесу створення динамічних веб-сторінок. Незважаючи на те, що сучасний PHP є мовою загального призначення, його найчастіше використовують як серверний інструмент для генерації HTML-коду, який потім інтерпретується веб-браузером [32].

Сьогодні виділяють три основні області використання PHP.

Для написання скриптів і повноцінних веб-додатків, що виконуються на стороні сервера. Це найпопулярніша сфера застосування, оскільки мова спочатку створювався саме для веб-розробок.

Для створення сценаріїв, виконуваних в командному рядку. Такі мініддатки можуть працювати на будь-якому ПК. Для їх виконання потрібно тільки парсер. Оскільки PHP містить потужні інструменти для роботи з рядками, такі сценарії найчастіше створюють для обробки текстових даних.

Для написання графічних інтерфейсів. PHP має безліч відгалужень, створених для реалізації різних завдань.

Під час створення проекту мова PHP була використана для написання веб-додатку, який значно поліпшив наочність тестування системи, спростив спосіб налаштування параметрів та виводу кінцевих результатів [32]. Завдяки додатку з веб-сторінки можна задавати параметри навантаження реактивної системи і перегляду результатів обробки даних за заданого навантаження.

3.2.6 Середовище розробки VS Code

Visual Studio Code — редактор вихідного коду, розроблений Microsoft для Windows, Linux і macOS. Позиціонується як «легкий» редактор коду для кроссплатформної розробки веб-і хмарних додатків. Включає в себе відладчик [33], інструменти для роботи з Git [34], підсвічування синтаксису, IntelliSense [35] і засоби для рефакторинга. Має широкі можливості для кастомізації: призначені для користувача теми, поєднання клавіш і файли конфігурації.

VS Code має велику кількість розширень для розробника. Для установки нового пакету потрібно перейти до вкладки "Extensions", ввести назву пакета в рядку пошуку, натиснути кнопку "Install". У редакторі вже є підтримка синтаксису і підказок стандартних функцій мови. Але без спеціального доповнення редактор не підказуватиме призначені для користувача функції з інших частин проекту. Тому для підтримки автодоповнення, аналізу коду, переходу до місця, де створена функція / клас / змінна, використовується доповнення PHP Intelephense. Щоб підказка не дублювалися необхідно відключити вбудовану в редактор підтримку коду для PHP: Extensions -> Search @builtin php -> PHP Language Features -> Disable [36].

3.2.7 Система автоматичної збірки Gradle

Gradle — система автоматичного складання, побудована на принципах Apache Ant і Apache Maven, але надає DSL на мовах Groovy і Kotlin замість традиційної XML-подібної форми подання конфігурації проекту. На відміну від Apache Maven, заснованого на концепції життєвого циклу проекту, і Apache Ant, в якому порядок виконання завдань (targets) визначається відно-

синами залежності (depends-on), Gradle використовує спрямований ациклічний граф для визначення порядку виконання завдань.

Gradle був розроблений для розширюваних многопроектной збірок, і підтримує інкрементальні збірки, визначаючи, які компоненти дерева збірки не змінилися і які завдання, залежні від цих частин, не вимагають перезапуску. Основні плагіни призначені для розробки і розгортання Java, Groovy і Scala додатків, але готуються плагіни і для інших мов програмування.

Альтернативою Gradle є система автоматичної збірки Maven. Ці дві системи збірки з одного боку різні, а з іншого боку мають і ряд подібностей. На цю тему на сайті Gradle є матеріал: "Migrating from Maven to Gradle" [37].

Як сказано в цьому керівництві, Gradle і Maven мають різницю у погляді на те, як збирати проект. Gradle заснований на графі завдань (task), які можуть залежати один від одного. Завдання виконують якусь роботу. Maven використовує модель певних фаз (phase), до яких приєднуються певні "цілі" (goals). У цих goals і виконується якась робота. Однак, при таких різних підходах обидві системи збирання слідують однією угодою і управління залежностями відбувається схоже. Щоб почати використовувати Gradle необхідно його завантажити.

Для початку потрібно встановити Gradle.

Існує безліч способів установки, в тому числі вручну Installing manually".

Головне, після виконання інструкції команда gradle -v показує версію встановленого Gradle.

3.2.8 Docker

Docker — програмне забезпечення для автоматизації розгортання і управління додатками в середовищах з підтримкою контейнеризації. Дозволяє «упакувати» додаток з усім його оточенням і залежностями в контейнер, який може бути перенесений на будь-яку Linux-систему з підтримкою cgroups в ядрі, а також надає середовище з управління контейнерами. Кожен

контейнер включає все необхідне для роботи програми: бібліотеки, системні інструменти, код і середу виконання [38]. Завдяки Docker можна швидко розгортати і масштабувати додатки в будь-якому середовищі і зберігати впевненість в тому, що код буде працювати.

До його переваг відносяться :

1. Прискорений процес розробки. Немає необхідності встановлювати допоміжні інструменти на зразок PostgreSQL, Redis, Elasticsearch: їх можна запускати в контейнерах.
2. Зручна інкапсуляція додатків.
3. Зрозумілий моніторинг.
4. Просте масштабування.

Коли ми встановлюємо Docker на локальну машину, то отримуємо клієнт (CLI) і http-сервер, що працює як daemon. Сервер надає REST API, а консоль просто перетворює введені команди в http-запити (Рис. 15).

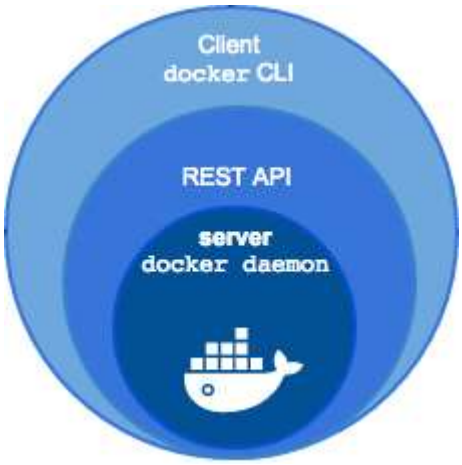

Рис. *15 Структура Docker на локальній машині*

В основі роботи Docker лежить стандартизований спосіб виконання коду. Docker — це операційна система для контейнерів. Подібно до того як віртуальна машина створює віртуальне уявлення апаратного забезпечення сервера (тобто усуває необхідність безпосередньо управляти таким), контейнери створюють віртуальне уявлення серверної операційної системи. Після уста-

новки на кожен сервер Docker надає доступ до простих команд, необхідних для збірки, запуску або зупинки контейнерів (Рис. 16).

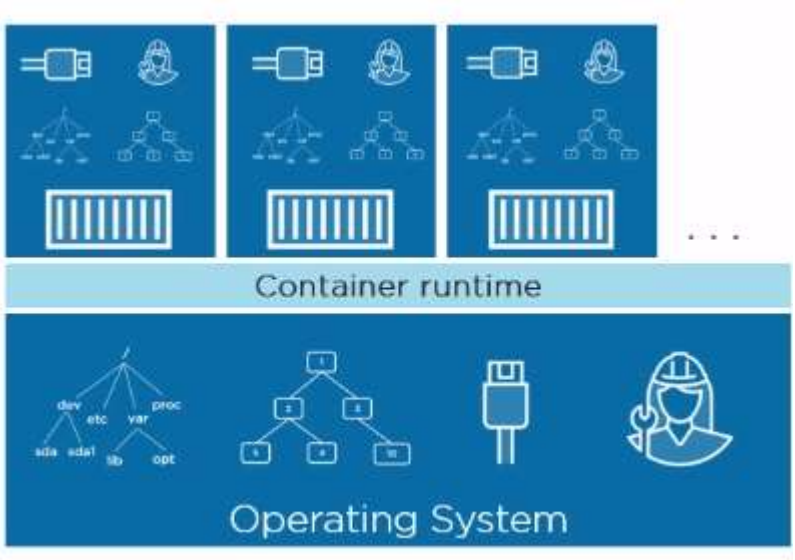

Рис. *16 Docker контейнери*

Compose — інструмент для створення і запуску багатоконтейнерних Docker додатків. У Compose, використовується спеціальний файл для конфігурації сервісів додатку. Потім, використовується проста команда, для створення і запуску всіх сервісів з конфігураційного файлу [38].

Compose чудовий для розробки, тестування і налаштування середовища, а також безперервної інтеграції.

Використання Compose зазвичай розділяється на три етапи:

1. Визначення оточення вашого застосування в Dockerfile, це можна зробити в будь-якому місці.
2. Визначення сервісів з яких буде складатися ваш додаток в docker-compose.yml, надалі вони зможуть бути запущені всі разом в ізольованому оточенні.
3. Виконання команди *docker-compose up* яка запустить весь додаток.

Docker застосовується для управління окремими контейнерами (сервісами), з яких складається програма.

Docker Compose використовується для одночасного управління декількома контейнерами, що входять до складу програми (Рис. 17). Цей інструмент пропонує ті ж можливості, що і Docker, але дозволяє працювати з більш складними додатками.

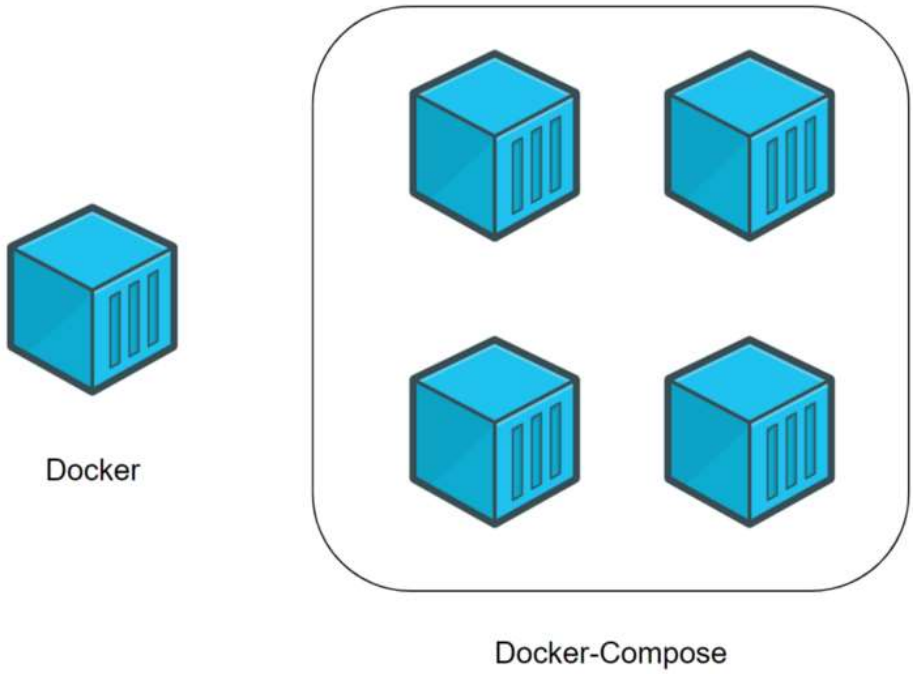

Рис. *17 Docker та Docker Compose*

3.2.9 MongoDB

MongoDB — це високопродуктивна документо-орієнтована база даних без схем даних, яка відноситься до нереляційних БД.

Структура: містить в собі безліч колекцій, вони ж містять безліч документів (об'єктів), які в свою чергу містять пари ключ-значення [39]. Документ має динамічну схему, що означає:

1. Документи в одній і тій же колекції не повинні мати однакове безліч пар ключ-значення.
2. Типи їх можуть бути різними.

Структура документа:

```
{
   field1: value1,
   field2: value2,
   field3: value3,
   ...
   fieldN: valueN
}
```

Графічно структура сутностей в реляційних БД виглядає так (Рис. 18):

Рис *18 Структура сутностей в реляційних БД*

У роботі з MangoDB потрібно прийняти для себе нову модель відображення (Рис. 19):

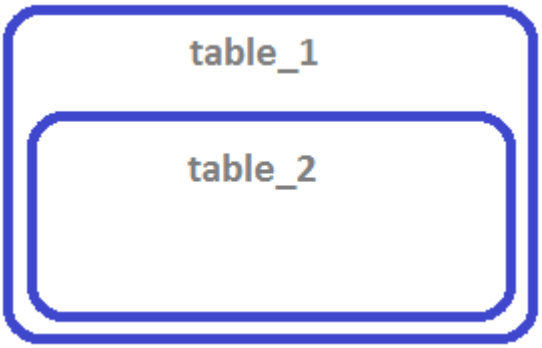

Рис. *19 Структура MangoDB*

Характеристики [39]:

1. Гнучкість. За рахунок зберігання даних в JSON документах.
2. Потужність. За рахунок збереження безлічі характеристик RDBMS, таких як вторинний ключ, динамічні запити, сортування, легка агре-

гація і інші. Це дає більшу функціональність, яку забезпечують реляційні бази.

3. Швидкість і масштабованість. Зберігаючи пов'язану інформацію в одному документі, запити можуть виконуватися набагато швидше. Autosharding дозволяє швидке масштабування за рахунок лінійного підключення «машин».

4. Легкість у використанні. MongoDB легка в установці, налаштуванні і використанні.

3.2.10 Apache

Apache — це HTTP сервер, що має високу надійність і гнучкість. Під HTTP-сервером слід розуміти програмне забезпечення для обробки HTTP-запитів. Основна робота Apache це обробка і відповідь HTTP-запитів і генерування змісту динамічних сторінок [41]. Apache має механізм для визначення віртуальних хостів, завдяки чому за однією IP-адресою може знаходиться необмежена кількість сайтів, саме таким чином працює віртуальний хостинг. Також існує велика кількість модулів, що дозволяють йому працювати з більшістю популярних мов програмування і розширювати функціональність. На сьогоднішній день Apache є одним з найпопулярніших ПЗ для веб-сервера. Його використовують хостинг-провайдери по всьому світу.

3.2.11 WireMock

WireMock — це симулятор API на основі хттп, це інструмент віртуалізації послуг або імітації сервера, це утиліта, бібліотека на java для створення заглушок над веб-сервісами [41]. Він створює HTTP-сервер, до якого ми могли б підключитися, як до реального веб-сервісу.

Це дозволяє коректно виконувати работу, коли API, від якого ви залежите, не існує або не є повним. Він підтримує тестування крайніх випадків і режимів відмови, які справжній API не може забезпечити надійно. Використання такої утиліти також може скоротити час збірки з годин до хвилин.

3.3 Вимоги до апаратного та програмного забезпечення

Основною задачею застосунку є демонстрація двох підходів розроби програмного забезпечення: реактивного та імперативного. Обробка даних відбувається на машині користувача. Тому були визначені наступні вимоги до апаратного та програмного забезпечення.

Мінімальні вимоги до апаратного забезпечення:

- Двох-ядерний процесор з базовою тактовою частотою 2.0 GHz.
- Оперативна пам’ять ємністю не менше 8 Гб.
- Жорсткий диск або SSD 240 Гб.

Рекомендовані вимоги до апаратного забезпечення:

- Чотирьох-ядерний процесор з базовою тактовою частотою 3.0 GHz.
- Оперативна пам’ять ємністю не менше 12 Гб.
- Жорсткий диск або SSD 240 Гб.

Вимоги до програмного забезпечення:

- ОС Linux.
- ПЗ для автоматизації розгортання та управління Docker.
- Система автоматичної збірки Gradle.
- Службова програма командного рядка cURL.
- Середовище розробки IntelliJ IDEA, версія Community.
- Веб-браузер.

3.4 Опис функціональних можливостей

Оскільки основною метою кваліфікаційної роботи є дослідження стану питання, існуючих алгоритмів, практик, підходів та методів вирішення про-

блеми, а не розробка кінцевого програмного продукту, то на даний момент функціональні можливості системи орієнтовані на зручність проведення досліджень та експериментів за навантаження, а не для використання у якості готової програмної системи для впровадження у існуючі програмні продукти або побудови нових продуктів на базі розробленого програмного забезпечення. Однак, реалізований підхід можна взяти за основу для створення реактивних систем та їх компонентів. Було реалізовано тестовий застосунок для тестування розподілення навантаження на ресурси комп'ютера. За допомогою простого і досить зрозумілого інтерфейсу на веб сторінці можна корегувати параметри тестування системи, а саме: обирати тип команди, кількість підключень, та час тестування, або запустити програму з параметрами за замовчуванням (Рис 20). Вихідними даними є таблиця з результатами досліджень та масив інфомації.

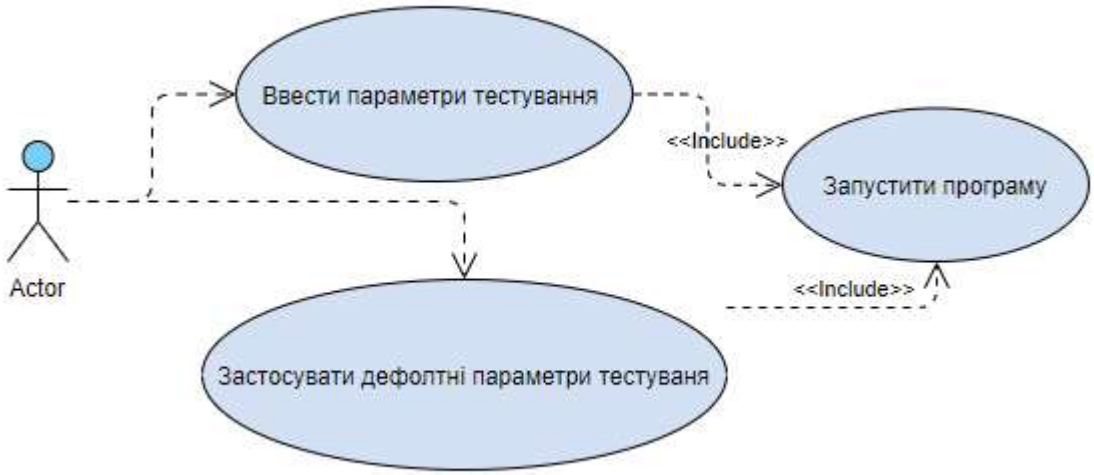

Рис. *20 Діаграма варіантів використання*

3.5 Модулі і алгоритми

Під час створення програмного застосунку було розроблено наступні модулі:

1. Модуль Cards Hub – головний модуль системи, що містить основний контролер.

2. Cards модулі – модулі мікросервісів, віддають головному модулю інформацію про користувачів, яку беруть з різних джерел.

3. Модуль Commons – допоміжний модуль, у якому знаходяться класи, що використовують кожен з модулів системи.

4. Утиліта RX Loader, за допомогою якої можна робити запити до основного модулю системи та підраховувати статистичні данні за результатами тестування навантаження.

3.5.1 Модуль Cards Hub

Hub є головним модулем системи. Основний контролер системи – Hub Controller є точкою входу в систему, він викликає сервіс Cards Service, який проходить по списку клієнтів (AdviceCardClient, FinesCardClient, RegularCardClient) і збирає усю інформацію, яка буде передана кінцевому користувачеві. У лістингу 1 наведено реактивну реалізацію основного методу loadCards контролеру Hub Controller, у якому викликається сервіс Cards Service, а у лістингу 2 наведено реактивну реалізацію методу loadCards сервісу Cards Service.

Лістинг *1 Метод loadCards контролеру Hub Controller, у якому викликається сервіс Cards Service*

```
public Flux<Card> loadCards(@RequestHeader("userId") String
userId,
                            @RequestHeader("longitude")
BigDecimal longitude,
                            @RequestHeader("latitude")
BigDecimal latitude,
                            @RequestParam("currentDate")
Long currentDate) {
    return cardsService.loadCards(UserData.builder()
            .currentDate(currentDate)
            .userId(userId)
            .geoPosition(GeoPosition.builder()
                    .latitude(latitude)
                    .longitude(longitude)
```

```
                .build())
            .build());
}
```

Лістинг *2 Метод loadCards сервісу Cards Service*

```
public Flux<Card> loadCards(UserData userData) {
    final Stream<Card> fluxList = cardClients.stream()
            .map(client -> getCards(userData, client))
            .flatMap(cardFlux -> cardFlux.toStream());
    return Flux.fromStream(fluxList);
}
```

У лістингах 3 і 4 наведено імперативну реалізацію методу loadCardsBasic контролеру Hub Controller та методу loadCardsBasic сервісу Cards Service відповідно.

Лістинг *3 Метод loadCardsBasic контролеру Hub Controller*

```
public List<Card> loadCardsBasic(@RequestHeader("userId")
String userId,

@RequestHeader("longitude") BigDecimal longitude,
                                    @RequestHeader("latitude")
BigDecimal latitude,

@RequestParam("currentDate") Long currentDate) {
    return cardsService.loadCardsBasic(UserData.builder()
            .currentDate(currentDate)
            .userId(userId)
            .geoPosition(GeoPosition.builder()
                    .latitude(latitude)
                    .longitude(longitude)
                    .build())
            .build());
}
```

Лістинг *4 Метод loadCardsBasic сервісу Cards Service*

```
public List<Card> loadCardsBasic(UserData userData) {
    return cardClients.stream()
            .parallel()
            .flatMap(client -> getCardsBasic(userData, cli-
ent).stream())
            .collect(Collectors.toList());
}
```

3.5.1 Cards модулі

Модулі Advice Cards, Fines Cards, Regular Cards ідентичні за своєю структурою. Вони різняться за рахунок джерел інформації з яких беруться дані для передачі через мікросервіси кінцевому користувачеві. Наприклад, для модулів Fines Cards та Regular Cards дані беруться з імітованого віддаленого ресурсу та бази даних відповідно, що знаходяться у Docker контейнерах , а для модуля Advice Cards дані генеруються сервісом. Тож, взаємодія між контролерами та сервіами у всіх цих модулів однакова, взаємодія між сервісами та клієнтами трохи відрізняється.

Модуль Advice Cards

Сервіс цього модулю генерує відповіді на запити системи із певного списку можливих варіантів. У лістингу 5 наведено реалізацію методу loadAdvices із класу AdviceController, який викликає AIAdviceService, аналогічним чином реалізовані методи loadFines та loadRegular у контролерах модулів Fines Cards та Regular Cards. У лістингу 6 наведено імперативну реалізацію методу loadAdvicesBasic.

Лістинг *5 Метод loadAdvices, що повертає масив рекомендацій*

```
public Flux<AdviceCard>
loadAdvices(@RequestHeader("userId") String userId,

@RequestHeader("longitude") BigDecimal longitude,

@RequestHeader("latitude") BigDecimal latitude,

@RequestParam("currentDate") Long currentDate) {
    return adviceService.defineAdvices(UserData.builder()
            .currentDate(currentDate)
            .userId(userId)
            .geoPosition(GeoPosition.builder()
                    .latitude(latitude)
                    .longitude(longitude)
                    .build())
            .build());
}
```

Лістинг *6 Метод loadAdvicesBasic, що повертає масив рекомендацій*

```
public List<AdviceCard>
loadAdvicesBasic(@RequestHeader("userId") String userId,

@RequestHeader("longitude") BigDecimal longitude,

@RequestHeader("latitude") BigDecimal latitude,

@RequestParam("currentDate") Long currentDate) {
    return
adviceService.defineAdvicesBasic(UserData.builder()
            .currentDate(currentDate)
            .userId(userId)
            .geoPosition(GeoPosition.builder()
                    .latitude(latitude)
                    .longitude(longitude)
                    .build())
            .build());
}
```

У лістингу 7 наведено реактивну реалізацію методу defineAdvices сервісу AIAdviceService, що генерує масив рекомендацій для кінцевого користувача і по одній віддає рекомендації головному модулю.

Лістинг *7 Метод defineAdvices*

```
    public      Flux<AdviceCard>      defineAdvices(UserData
userData) {
             final     Stream<AdviceCard>     cardStream     =
IntStream.range(0, 10)
                     .mapToObj(ind -> AdviceCard.builder()
                             .userId(userData.getUserId())

.amount(BigDecimal.valueOf(rnd.nextLong()))

.adviceType(AdviceType.fromInt(rnd.nextInt(AdviceType.value
s().length - 1)))

.id(UUID.randomUUID().toString())

.executionUrl(adviseProperties.getExecuteUrl())
```

```
                                .type(CardType.ADVISE)
                                .build());
                return Flux.fromStream(cardStream);
        }
```

У лістингу 8 наведено імперативну реалізацію методу defineAdvicesBasic сервісу AIAdviceService, який також генерую список рекомендацій для кінцевого користувача, але віддає головному модулю системи сформований кінцевий список.

Лістинг *8 Метод defineAdvicesBasic*

```
public List<AdviceCard> defineAdvicesBasic(UserData
userData) {
    return IntStream.range(0, 10)
            .mapToObj(ind -> AdviceCard.builder()
                    .userId(userData.getUserId())

.amount(BigDecimal.valueOf(rnd.nextLong()))

.adviceType(AdviceType.fromInt(rnd.nextInt(AdviceType.value
s().length - 1)))
                    .id(UUID.randomUUID().toString())

.executionUrl(adviseProperties.getExecuteUrl())
                    .type(CardType.ADVISE)
                    .build())
            .collect(Collectors.toList());
}
```

Модуль Fines Cards

У модулі Fines Cards сервіс використовує клієнта аби отримати дані з імітованого віддаленого ресурсу. Реактивну реалізацію методу loadFines класу RemoteFinesService, що обробляє дані отримані від клієнта описано у лістингу 9.

Лістинг *9 Реалізація методу loadFines класу RemoteFinesService*

```
public Flux<FineCard> loadFines(UserData userData) {
    return FinesClient.getFines(userData.getUserId())
            .flatMapIterable(res -> res)
            .map(fine ->
                    FineCard.builder()
                            .userId(userData.getUserId())
```

```
.dueDate(fine.getDueDate().getTime())
                                .amount(fine.getAmount())
                                .fineType(fine.getFineType())
                                .id(fine.getId())

.executionUrl(properties.getExecuteUrl())
                                .type(CardType.FINES)
                                .build());
}
```

Імперативний варіант реалізації описано у лістингу 10.

Лістинг *10 Реалізація методу loadFinesBasic класу RemoteFinesService*

```
public List<FineCard> loadFinesBasic(UserData userData) {
        return
eGovClient.getFinesBasic(userData.getUserId()).stream()
                .map(fine ->
                        FineCard.builder()

.userId(userData.getUserId()).dueDate(fine.getDueDate().get
Time())

.amount(fine.getAmount().fineType(fine.getFineType())
                                    .id(fine.getId())
 .executionUrl(properties.getExecuteUrl())
                                    .type(CardType.FINES)
                                    .build()
                ).collect(Collectors.toList());
    }
}
```

У методі getFines класу FinesClient клієнт отримує дані з віддаленого ресурсу за userId, який знаходиться у контейнері докера за адресою http://localhost:9999, що відображено у лістингу 11, імперативну реалізацію описано у лістингу 12.

Лістинг *11 Метод getFines класу FinesClient*

```
public Mono<List<FineDTO>> getFines(String userId) {
    return client.get()
            .uri(uriBuilder ->
uriBuilder.path("/fines").queryParam("userId",
userId).build())
            .exchange()
            .flatMap(res ->
res.bodyToMono(FinesResponse.class));
}
```

Лістинг *12 Метод getFinesBasic класу FinesClient*

```
public List<FineDTO> getFinesBasic(String userId) {
    return restTemplate.getForObject(baseURL +
"/fines?userId=" + userId, FinesResponse.class);
}
```

Модуль Regular Cards

Модуль Regular Cards використовує репозиторій бази даних RegularRepositoryReactive для отримання даних користувача, реактивну реалізацію можна побачити у лістингу 13, імперативну – у лістингу 14. Сам запит до бази даних відображено у лістингу 15, імперативна версія запиту виглядає аналогічно, але повертає List замість Flux.

Лістинг 13 *Метод loadRegular сервісу PersistenceRegularService*

```
public Flux<RegularCard> loadRegular(UserData userData) {
    Date from = new Date(userData.getCurrentDate() - DAY);
    Date to = new Date(userData.getCurrentDate() + DAY);
    return
reactiveResuarRepository.findByUserIdAndDueDateBetween(user
Data.getUserId(), from, to)
            .map(doc ->
                    RegularCard.builder()
                            .userId(userData.getUserId())

.dueDate(doc.getDueDate().getTime())
                            .amount(doc.getAmount())

.targetAccount(doc.getTargetAccount())
                            .id(doc.getId())

.executionUrl(properties.getExecuteUrl())
                            .type(CardType.REGULAR)
                            .build()
            );
}
```

Лістинг 14 *Метод loadRegular сервісу PersistenceRegularService*

```
public List<RegularCard> loadRegularBasic(UserData
userData) {
    Date from = new Date(userData.getCurrentDate() - DAY);
    Date to = new Date(userData.getCurrentDate() + DAY);
    return
regularRepository.findByUserIdAndDueDateBetween(userData.ge
tUserId(), from, to).stream()
```

```
            .map(doc ->
                    RegularCard.builder()
                            .userId(userData.getUserId())

.dueDate(doc.getDueDate().getTime())
                            .amount(doc.getAmount())

.targetAccount(doc.getTargetAccount())
                            .id(doc.getId())

.executionUrl(properties.getExecuteUrl())
                            .type(CardType.REGULAR)
                            .build()
            ).collect(Collectors.toList());
}
```

Лістинг *15 Запит до бази даних з інтерфейсу RegularRepositoryReactive*

```
Flux<RegularDocument> findByUserIdAndDueDateBetween(String
userId);
```

3.5.2 Модуль Commons

У модулі Commons знаходяться класи, що використовують усі інші модулі сисеми. Цей модуль можна вважати допоміжним.

Класи, які знаходяться у модулі Commons:

1. Card, що містить поля id, userId, type, amount, executionUrl.
2. CardType – перелік, є такі типи як FINES, REGULAR, ADVISE.
3. GeoPosition, що містить поля longitude, latitude.
4. UserData, що містить поля userId, geoPosition, currentDate.

3.5.4 Утиліта RX Loader

Утиліта RX Loader відправляє запити головному модулю системи Hub і підраховує статистичні дані, отримані в ході тестування навантаження системи. Для того щоб утиліта почала відправляти запити потрібно активувати команду load або load_basic для реактивної і імперативної системи відповідно, а як параметри задати число connections (підключень), розподілених на 4

потоки і час в мілісекундах. Наприклад: load 10 10s для тестування реактивної системи, або load_basic 10 10s для імперативної.

Результатом роботи буде таблиця з обробленими статистичними даними. Серед цих даних будуть такі: мінімальне і максимальний час за який прийшла перша відповідь на запит (пакет даних), максимальний і мінімальний часовий проміжок між відповідями, мінімальне і максимальне загальне час, за которо були отримані всі відповіді за сесію.

Так само будуть отримані дані про те скільки всього було відправлено потоків за сесію, скільки з них завершилося успішно, скільки з помилкою, скільки в середньому запитів було оброблено в 1 секунду.

3.6 Структури даних

Модуль системи Regular Cards використовує репозиторій бази даних RegularRepositoryReactive для отримання даних користувача. Для роботи з цим модулем використовується документ бази даних MongoDB. Кожний запис у документі бази має власний ID, аби отримати всі записи, які стосуються потрібного користувача у запиті до бази передається userID, тобто ID користувача. На початку розробки програмного застосунку було прийняте рішення зімітувати у базі даних платежі різного характеру, які користувачам потрібно сплатити до певної кінцевої дати, тому документ бази містить наступні дані:

1. id — ID запису.
2. targetAccount — акаунт користувача.
3. Date dueDate — кінцева дата сплати платежу.
4. String userId — ID користувача.
5. BigDecimal amount — сума платежу.

3.7 Проект інтерфейсу

Головні показники роботи системи можна отримати, задавши налаштування тестування навантаженням на веб сторінці з простим інтерфейсом (Рис 21). На першій сторінці користувачеві потрібно ввести у поле command команду load, що запускає реактивні методи системи або load_basic, що запускає методи, реалізовані в імперативному стилі; у полі connections потрібно ввести кількість одночасних запитів, які буде опрацьовувати система; у полі time потрібно задати час у секундах, який виділено системі для опрацювання запитів, до числа, що означає кількісь секунд обовязково потрібно додати літеру s без пробілу. Коли уси налаштування зафіксовані, потрібно натиснути кнопку Run.

Рис. *21 Інтерфейс сторінки з налаштуваннями*

На другій сторінці (Рис. 22) користувачеві буде виведена таблиця з результатами тестування навантаження. Для всіх показників таблиці виводиться мінімальний та максимальний показник відстежений за всю сесію, а також середнє значення та стандартне відхилення на основі вибірки, що обчислюється за формулою $STDev=\sqrt{[(\sum(x - x)^2) / n]}$ Серед показників у таблиці:

1. First — час, через який система отримала першу відповідь по запитам від мікросервісів , вимірюється у мілісекундах.
2. Distanse — час, який пройшов між відповідями на запити, порівнюються відповіді усіх мікросервісів, вимірюється у мілісекундах.

3. Total — час, за який було отримано усі відповіді від усіх сервісів, вимірюється у мілісекундах.
4. Event count — кількість відповідей на кожен із запитів, виводиться мінімальна та максимальна кількість, вимірюється у одиницях.

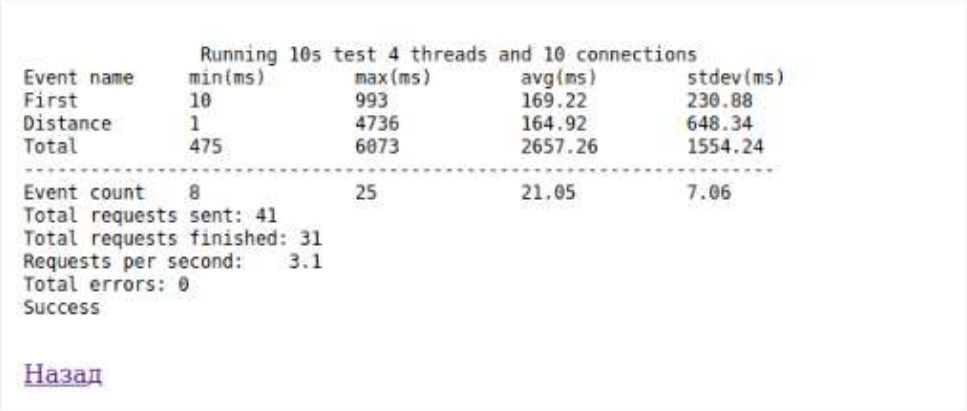

Рис. *22 Таблиця результатів*

Поза таблицею можна побачити інформацію про те, скільки всього запитів було надіслано за сесію, скільки запитів встигло отримати відповідь, скільки в середньому запитів було оброблено в 1 секунду, скільки було помилок з’єднання.

РОЗДІЛ 4 ПОРІВНЯЛЬНИЙ АНАЛІЗ РЕАКТИВНОЇ ТА ІМПЕРАТИВНОЇ СИСТЕМИ

4.1 Порівняльний аналіз отриманих статистичних даних

При розробці програмного застосунку обидві системи було протестовано різним навантаженням за однакові проміжки часу. Для тестування кількість підключень збільшувалась від 10 до 170 з кроком 10 під час кожного наступного тесту, а час тестування залишався незмінним і складав 60 секунд.

4.1.1 Порівняння часу отримання перших пакетів даних від систем

У реактивній системі кожен з мікросервісів віддає дані головному модулю частинами «по готовності», тобто, як тільки якийсь з трьох сервісів системи отримає першу порцію даних, з усього об’єму даних, що в нього запитували, він одразу передає цю порцію даних головному модулю, а головний модуль кінцевому користувачеві. Так відбувається з усіма подальшими порціями даних, які кожен з мікросервісів передає головному модулю. На відміну від реактивної реалізації, під час роботи імперативної версії системи кожен з мікросервісів накопичує у себе всю запитувану інформацію і віддає її головному мудулю тільки коли отримає останній пакет даних з усього об’єму. Аналогічно до цього головний модуль віддає весь об’єм даних отриманий від усіх мікросервісів тоді, коли отримає дані від останнього мікросервісу. Під час аналізу отриманих статистичних даних після тестування обох систем за показником First було отримано вибірки даних за мінімальним, максимальним та середнім часом, за якого було отримано перші пакети даних.

Розглянемо графік, на якому порівняно мінімальний час, за який приходять перші пакети даних реактивної системи і час, за який імперативна система віддає всі дані, залежність часу від кількості підключень (Рис. 23).

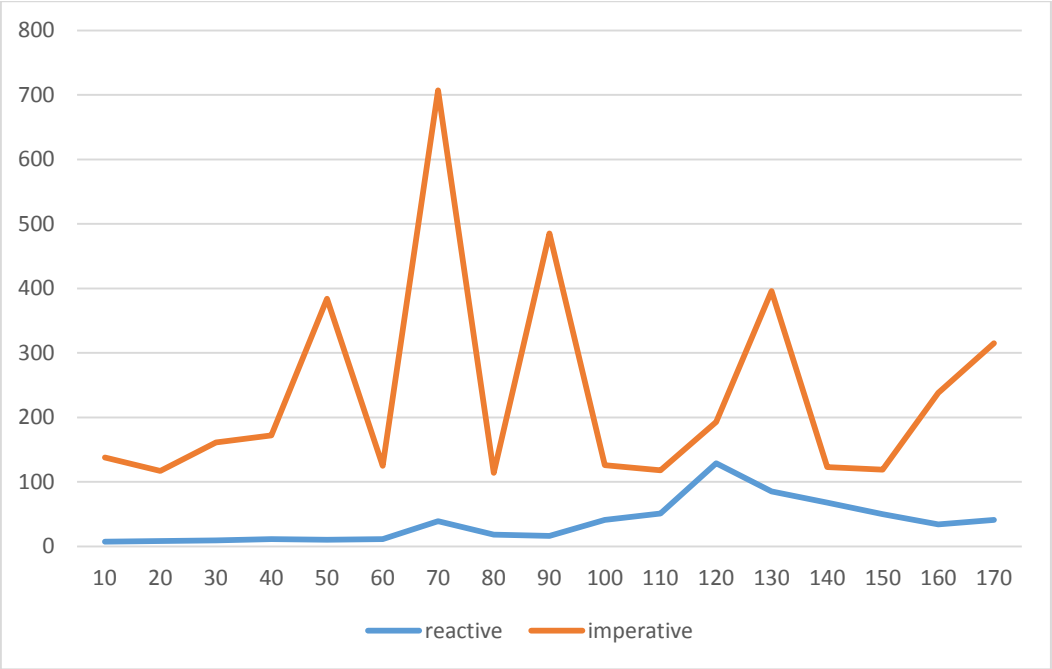

Рис *23 Мінімальні показники отримання першого пакету даних для реактивної та імперативної реалізації*

З графіку добре видно, що мінімальний час за який приходять перші пакети даних реактивної системи в десятки разів менший ніж найменший час, за який імперативна система віддасть всі дані.

Порівняємо тепер максимальний (Рис. 24) та середній (Рис. 25) час для обох систем, залежність часу від кількості підключень.

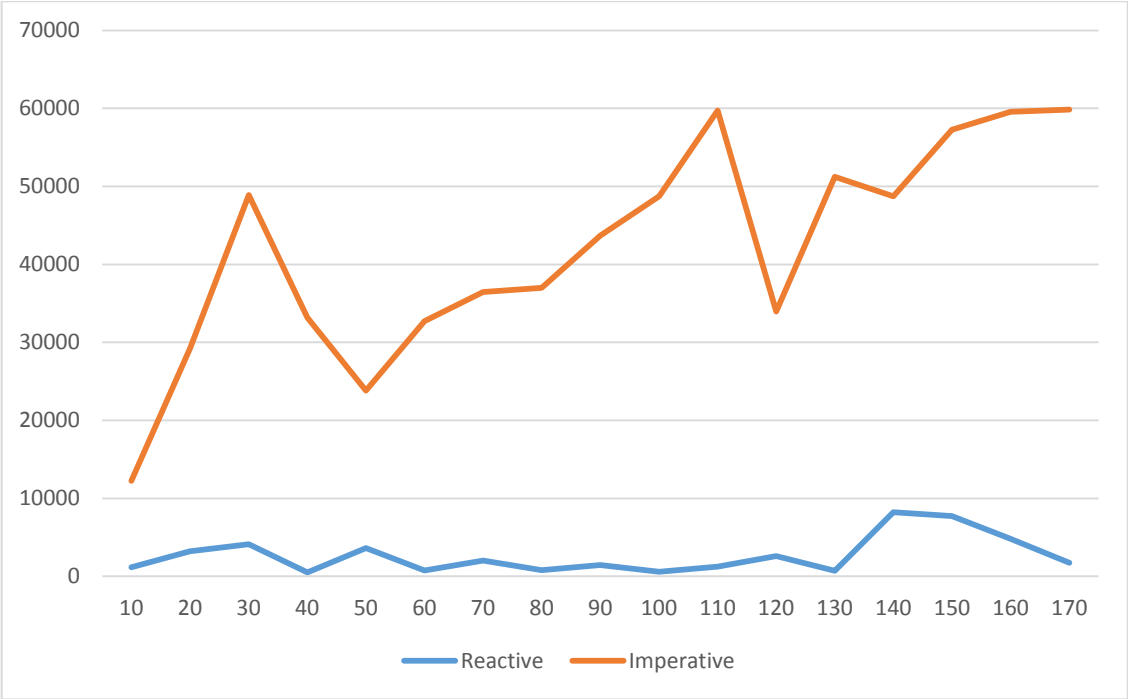

Рис. *24 Максимальні показники отримання першого пакету даних для реактивної та імперативної реалізації*

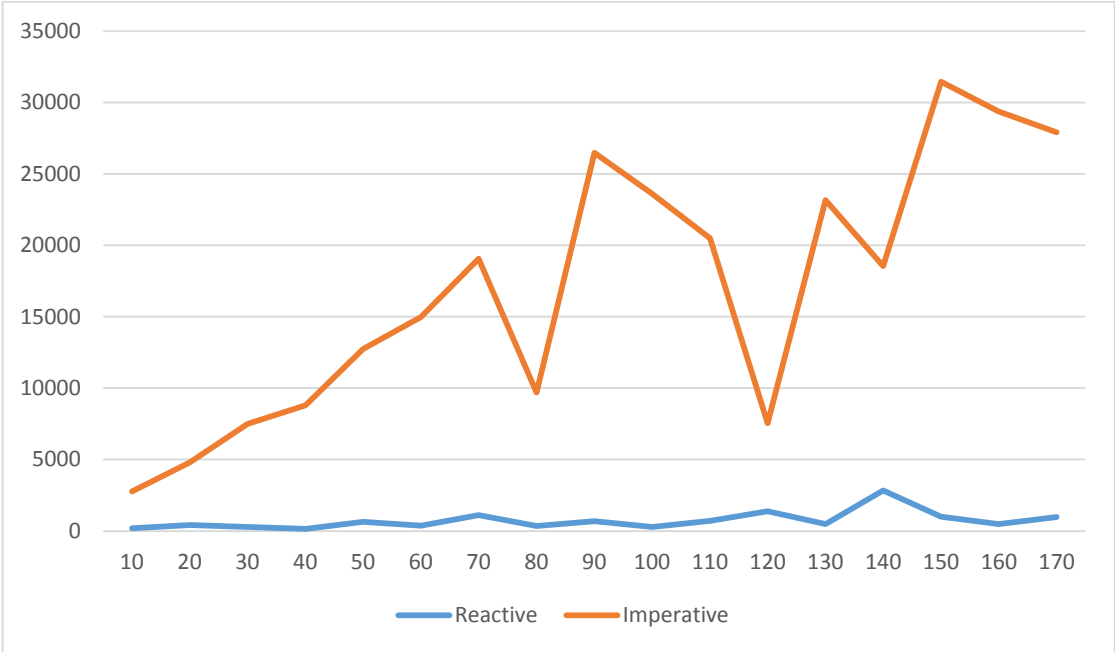

Рис. *25 Середні показники отримання першого пакету даних для реактивної та імперативної реалізації*

З двох наведених вище графіків видно, що максимальний і середній час за який приходять перші пакети даних реактивної системи також в рази

менший ніж максимальний та середній час, за який імперативна система віддасть всі дані.

4.1.2 Порівняння часу отримання всіх пакетів даних

Для порівняння статистичних даних про отримання усього об’єму інформації розглянемо показник Total. Під час аналізу отриманих статистичних даних після тестування обох систем було отримано вибірки даних за мінімальним, максимальним та середнім часом, за якого було отримано всі пакети даних які повинна була видати система.

Розглянемо графік, на якому порівняно мінімальний час, за який приходять всі пакети даних реактивної системи і час, за який імперативна система віддає всі дані, залежність часу від кількості підключень (Рис. 26).

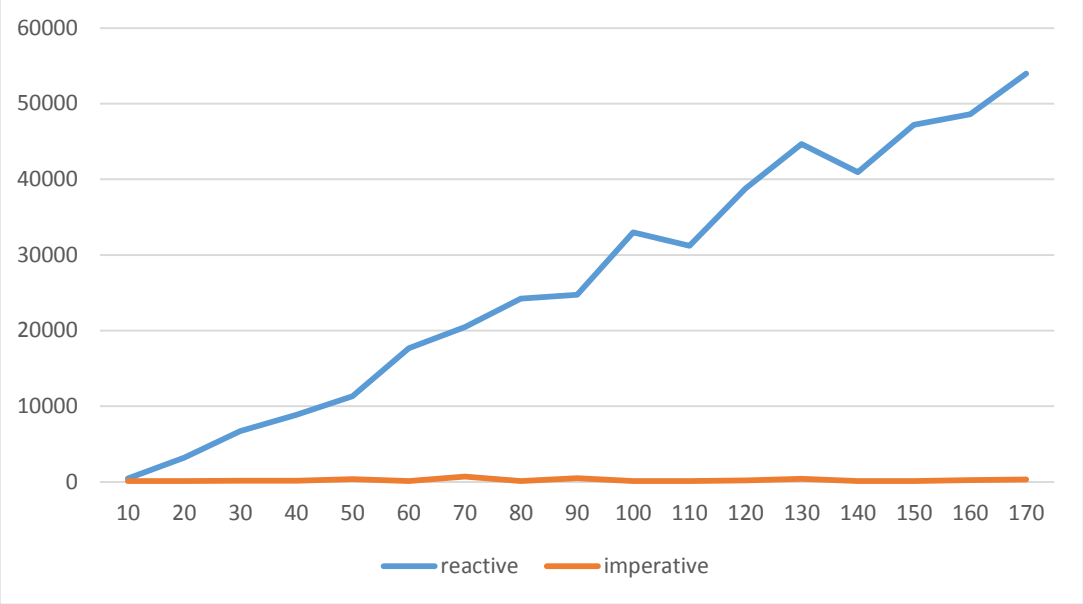

Рис *26 Мінімальні показники отримання всіх пакетів даних для реактивної та імперативної реалізації*

З графіку добре видно, що мінімальний час за який приходять всі пакети даних реактивної системи в десятки разів більший ніж найменший час, за який імперативна система віддасть всі дані.

Порівняємо тепер максимальний (Рис. 27) та середній (Рис. 28) час для обох систем, залежність часу від кількості підключень.

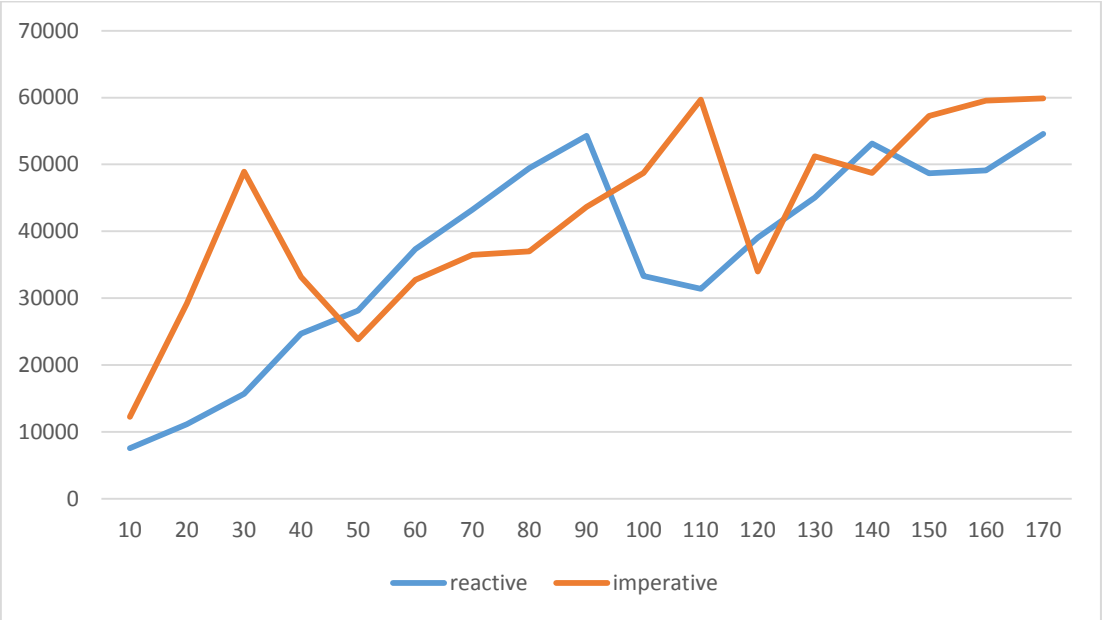

Рис *27 Максимальні показники отримання всіх пакетів даних для реактивної та імперативної реалізації*

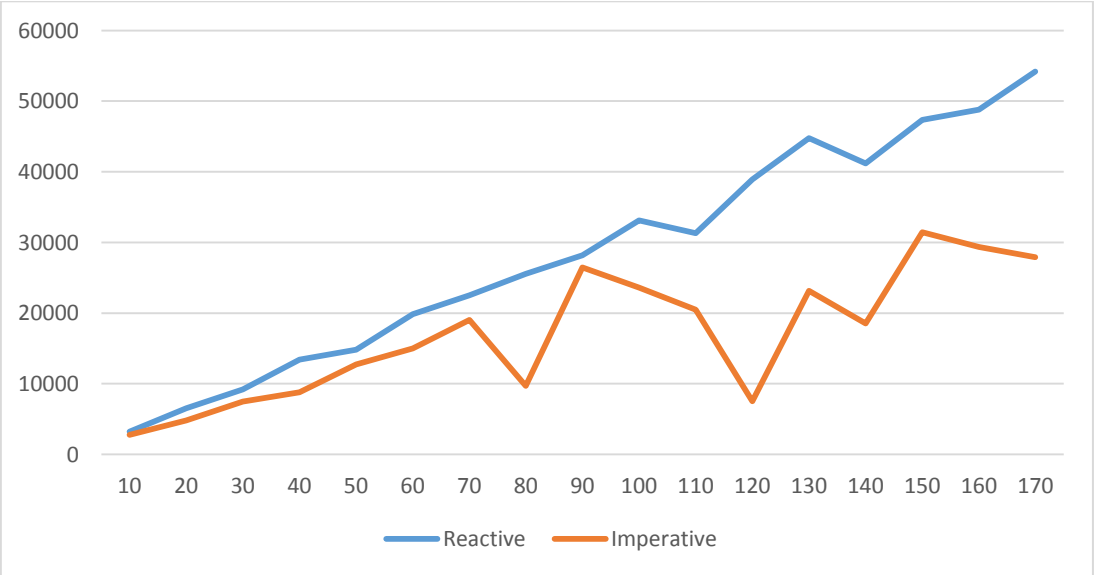

Рис *28 Середні показники отримання всіх пакетів даних для реактивної та імперативної реалізації*

З двох наведених вище графіків видно, що максимальний і середній час за який приходять всі пакети даних реактивної системи або більший, або бли-

зький до максимального та середнього, за який імперативна система віддасть всі дані.

З описаних вище отриманих та порівняних статистичних даних можна зробити наступні висновки: в середньому, щоб отримати всю інформацію реактивною системою нам потрібно буде більше часу, але обробляти її ми можемо почати до того, як нам прийшла остання відповідь, таким чином, в момент часу, коли реактивна система віддасть нам частину відповідей, у нас вже буде на руках половина всієї інформації, яка нас цікавила, а імперативна система до цього часу не видасть ще нічого. Відповідно, використовуючи реактивну систему ми зможемо почати працювати з отриманими даними раніше, так як раніше почнемо отримувати пакети даних, при цьому ми отримаємо всі пакети даних трохи повільніше, проте, у випадку імперативної системи, в цілому весь обсяг даних прийде нам трохи швидше, але ми не зможемо почати працювати з даними доки не прийдуть абсолютно всі пакети даних.

4.1.3 Дослідження середнього квадратичного відхилення

Стандартне відхилення або середнє квадратичне відхилення —один із найпоширеніших показників розсіювання (розкиду) значень випадкової величини відносно її математичного сподівання, тобто центру розподілу. Має ту ж розмірність, що і випадкова величина. В літературі для позначення стандартного відхилення використовується літера грецької абетки сигма σ.

За визначенням середнє квадратичне відхилення є додатнім квадратним коренем із дисперсії. Як і дисперсія характеризує розсіяння значень навколо центру розподілу: більшому значенню стандартного відхилення відповідає більший їх розкид. Формула середнього квадратичного відхилення $STDev=\sqrt{[(\sum(x - x)^2) / n]}$.

На графіку на рисунку 29 видно, що середнє квадратичне відхилення по часу отримання перших пакетів даних у реактивної системи значно менше, ніж у імперативної, що свідчить про те, що серед зібраних статистичних даних отриманих в результаті тестування реактивної системи менший роз-

кид, ніж серед даних отриманих після тестування імперативної системи. Аналогічне можна стверджувати і відносно порівняння середнього квадратичного відхилення по часу отримання всіх пакетів даних системами (Рис. 30).

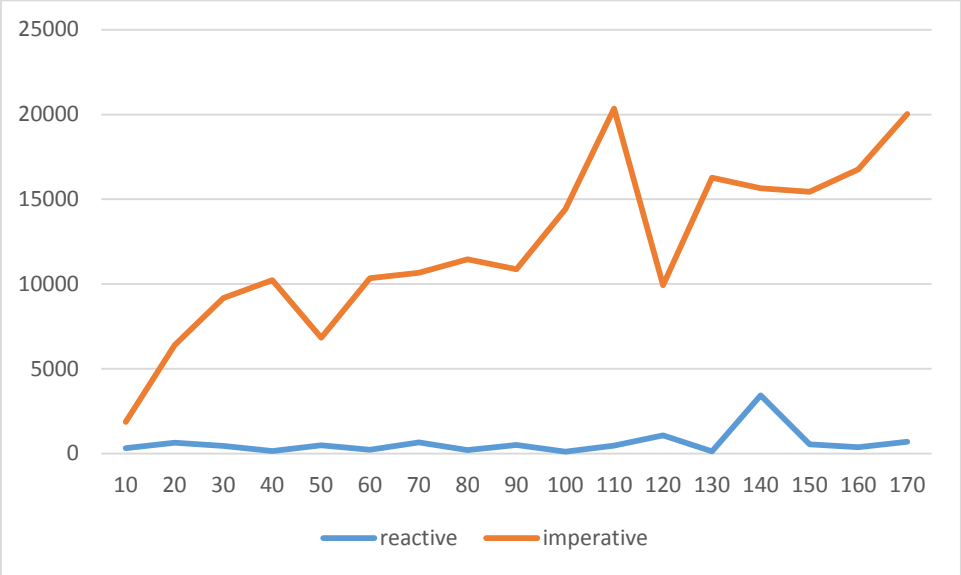

Рис. *29 Середнє квадратичне відхилення по часу отримання перших пакетів даних*

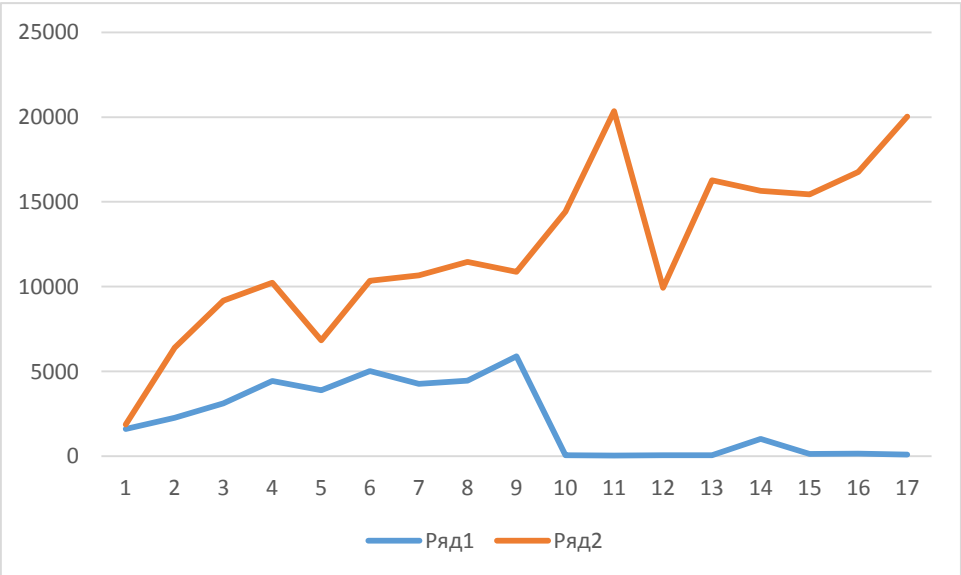

Рис. *30 Середнє квадратичне відхилення по часу отримання всіх пакетів даних*

4.2 Дослідження зміни часу між відповідями мікросервісів реактивної системи

Для дослідження зміни часу між відповідями (дистанцією) мікросервісів реактивної системи під час тестування кількість підключень також збільшувалась від 10 до 170 з кроком 10 під час кожного наступного тесту, а час тестування залишався незмінним і складав 60 секунд. За отриманими статистичними даними було побудовано графік (Рис. 31), на якому порівняно середній та максимальний час між відповідями, оскільки мінімальний час при кожному тесті складав 1 мс. На графіку відображено залежність часу в мілісекундах від кількості підключень.

З графіку добре видно, що не зважаючи на те, що максимальний час очікування між відповідями збільшується при зростанні кількості одночасних підключень, проте середній час очікування майже не змінюється, це свідчить по те, що навіть за великої кількості підключень максимальний час очікування між відповідями мікросервісів буде траплятись вкрай рідко.

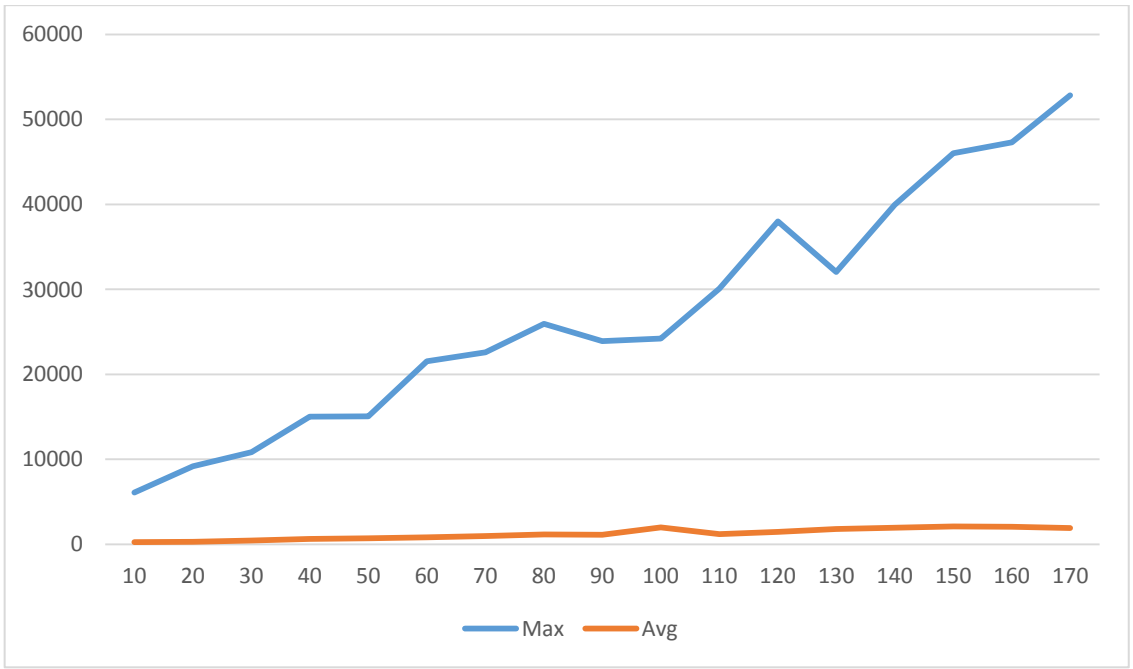

Рис. *31 Максимальний та середній показники часу очікування відповідей реактивної системи*

4.3 Дослідження показників реактивної системи під час збільшення навантаження

Для дослідження поведінки системи за умови великої кількості підключень було проведено тестування за 500 підключень (Рис. 32) та за 1000 підключень (Рис. 33).

```
Running 200s test 4 threads and 500 connections
Event name      min(ms)         max(ms)         avg(ms)         stdev(ms)
First           117             3039            1833.61         953.53
Distance        1               124356          6296.89         19132.34
Total           163511          164401          164105.26       166.72
---------------------------------------------------------------------
Event count     6               25              15.86           9.21
Total requests sent: 1000
Total requests finished: 500
Requests per second:    2.777777777777777
Total errors: 0
Success
```

Рис. *32 Результати тестування системи за 500 підключень*

```
Running 500s test 4 threads and 1000 connections
Event name      min(ms)         max(ms)         avg(ms)         stdev(ms)
First           105             3070            2451.70         447.34
Distance        1               312493          11628.32        46119.80
Total           325132          329028          326343.78       350.82
---------------------------------------------------------------------
Event count     6               25              16.36           8.68
Total requests sent: 2000
Total requests finished: 1000
Requests per second:    2.0
Total errors: 0
Success
```

Рис. *33 Результати тестування системи за 1000 підключень*

З отриманих результатів видно наступне:

1. Середній час отримання перших пакетів даних за 500 підключень не перевищує 2 секунди, а за 1000 підключень – не перевищує 2,5 секунди.
2. Середня дистанція між пакетами даних за 500 підключень складає приблизно 6 секунд, а за 1000 підключень – близько 12 секунд.
3. Середній час отримання всіх пакетів даних за 500 підключень складає близько 16 секунд, за 1000 підключень – близько 33 секунд.

4. Найбільші значення середнього квадратичного відхилення спостерігається у дистанції між пакетами даних в обох випадках, у всіх інших параметрів це значення знаходиться у межах 1000 мс.

З описаних вище характеристик результатів тестування можна зробити висновок, що навіть за великої кількості підключень реактивна система видає задовільні показники роботи , єдиною величиною з великим розкидом є дистанція між пакетами даних.

ВИСНОВКИ

1. Аналіз джерел з питань реактивного програмування, розподілу навантаження комп’ютера та проектування реактивних систем.

2. Досліджено сучасні реактивні технології та фреймворки для створення реактивних систем, визначено їх основні переваги та недоліки та можливі області застосування.

3. Шляхом аналізу можливостей парадигми реактивного програмування та проведення теоретичних і практичних досліджень було сформовано набір методів для дослідження, проведено їх порівняння та аналіз, а також було визначено стек технологій та інструментів розробки, визначено їх відносну продуктивність, ступінь інтеграції та зручність використання.

4. Досліджена можливість використання реактивного програмування для вирішення проблем пікових навантажень та оптимального розподілення ресурсів.

5. Спроектовано та реалізовано програмний застосунок на мовою програмування Java для порівняльного аналізу реактивного та імперативного підходу до створення систем .

6. На основі отриманих статистичних даних проведено порівняльний аналіз показників розроблених реактивної та імперативної систем.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Reactivemanifesto.org. Манифест реактивного програмування. URL: http://www.reactivemanifesto.org/ (дата звернення 20.11.2020)

2. Докука О., Лозинский И. Практика реактивного программирования в Spring 5. Москва : ДМК Пресс, 2019. 508с.

3. Kuhn R. Reactive Design Patterns. Manning Publications, 2017. 387с.

4. McKee H. Designing Reactive Systems. O'Reilly Media, 2017. 475с.

5. Salvaneschi G., Eugster P., Mezini M. «Programming with implicit flows» IEEE Software, 2014. С. 85– 89.

6. A. Jbara and D. Feitelson «How Programmers Read Regular Code: A Controlled Experiment Using Eye Tracking» in International Conference on Program Comprehension, 2015. С. 176– 179.

7. Mezzalira L. Front-End Reactive Architectures. Apress, 2018. 195с.

8. Gustafson, J.L. Reevaluating Amdahl’s Law. CACM, 31(4. 5), 1988. С. 532– 533.

9. Merriam-webster.com. Американський тлумачний словник англійської мови. URL: https://www.merriam-webster.com/ (дата звернення 20.11.2020).

10. Little J. D. C., Graves S. C., Little's Law Building Intuition . International Series in Operations Research & Management Science, 2008. 81с.

11. Офиційний сайт ReactiveX. URL: http://reactivex.io/ (дата звернення 20.11.2020)

12. Tomasz Nurkiewicz & Ben Christensen. Reactive Programming with RxJava: Creating Asynchronous, Event-Based Applications, First Edition. – “O’REILLY Media”, 2017. – 345.

13. Maier I., Odersky M. «Higher-Order Reactive Programming with Incremental Lists» in European Conference on Object-oriented Programming. Springer-Verlag, 2013. С. 707– 708.

14. Daniels P. P.. Atencio L. RxJS in Action. Manning Publications, 2017. 352с.

15. Reactive-streams.org. Reactive Streams. URL: https://www.reactive-streams.org/ (дата звернення 20.11.2020)

16. Roestenburg R., Bakker R., Williams R. Akka in Action. Manning Publications, 2016. 448 с.

17. Maurer N., Wolfthal M. A. Netty in Action. Manning Publications, 2015. 296 с.

18. RxNetty vs Tomcat Performance Results. URL: https://github.com/NetflixSkunkworks/WSPerfLab/blob/master/testresults/RxNetty_vs_Tomcat_April2015.pdf (дата звернення 21.11.2020)

19. DeVore D. K., Walsh S., Hanafee B. Reactive Application Development. Manning Publications, 2018. 288 с.

20. Netflixtechblog.com. Reactive Programming in the Netflix API with RxJava. URL: https://netflixtechblog.com/reactive-programming-in-the-netflix-api-with-rxjava-7811c3a1496a (дата звернення 21.11.2020)

21. Richard Blewett, Andrew Clymer. Pro Asynchronous Programming with .NET, First Edition. – «Apress», 2013. 675с.

22. Channel9.msdn.com. Event processing at all scales with Reactive Extensions. URL: https://channel9.msdn.com/events/TechDays/Techdays-2014-the-Netherlands/Event-processing-at-all-scales-with-Reactive-Extensions (дата звернення 22.11.2020)

23. Developers.soundcloud.com. Concurrency in Android using RxJava. URL: https://developers.soundcloud.com/blog/hassle-free-concurrency-in-android-using-rxjava (дата звернення 22.11.2020)

24. Blog.couchbase.com. Why Couchbase chose RxJava for the new Java SDK. URL: https://blog.couchbase.com/why-couchbase-chose-rxjava-new-java-sdk/ (дата звернення 22.11.2020)

25. Bruce M., Pereira P. A. Microservices in Action. Manning Publications, 2018. 392 с.

26. Carnell J. Spring Microservices in Action. Manning Publications, 2017. 384 с.

27. Docs.spring.io. Reactive Web Applications. URL: https://docs.spring.io/spring/docs/5.0.0.M4/spring-framework-reference/html/web-reactive.html (дата звернення 24.11.2020)

28. Projectreactor.io Project Reactor URL: https://projectreactor.io/ (дата звернення 24.11.2020)

29. Kishori Sharan, Java Language Features. Manning, 2018, 895 с.

30. Craig Walls, Spring in Action 5th Edition. Manning, 2019, 494 с.

31. Docs.spring.io Web on Reactive Stack URL : https://docs.spring.io/spring-framework/docs/current/reference/html/web-reactive.html (дата звернення 25.11.2020)

32. Reiersol D., Baker M., Shiflett C. PHP in Action. Manning Publications, 2007. 552 с.

33. Code.visualstudio.com Debugging URL : https://code.visualstudio.com/docs/editor/debugging (дата звернення 25.11.2020)

34. Code.visualstudio.com Git support URL : https://code.visualstudio.com/docs/editor/versioncontrol#_git-support (дата звернення 25.11.2020)

35. Code.visualstudio.com IntelliSenseURL : https://code.visualstudio.com/docs/editor/intellisense (дата звернення 25.11.2020)

36. Code.visualstudio.com PHP in Visual Studio Code URL : https://code.visualstudio.com/docs/languages/php (дата звернення 25.11.2020)

37. gradle.org Gradle Build Tool URL : https://gradle.org/ (дата звернення 25.11.2020)

38. Nickoloff J., Kuenzli S. Docker in Action, Second Edition. Manning Publications, 2019. 336 с.

39. Banker K., Bakkum P., Verch S., Garrett D., Hawkins T. MongoDB in Action, Second Edition. Manning Publications, 2016. 480 с.

40. Petersen T. Web Development with Apache and Perl. Manning Publications, 2002. 424 c.

41. Wiremock.org WireMock User Documentation URL : http://wiremock.org/docs/ (дата звернення 25.11.2020)

42. Вуколова А. І., магістрантка, Полякова Н.П., доц., канд. техн. наук. — науковий керівник. Використання реактивного програмування для підвищення рівня абстракції коду програм. Молода наука-2020 : зб. наук. праць студентів, аспірантів і молодих вчених. Запоріжжя : ЗНУ, 2020. Т. 5. С. 80–81.

43. Вуколова А. І., магістрантка, Полякова Н.П., доц., канд. техн. наук — науковий керівник. Розробка реактивної системи для швидкої передачі даних. Матеріали XXV науково-технічної конференції сту-дентів, магістрантів, аспірантів, молодих вчених та викладачів. Запоріжжя : ЗНУ, 2020. С. 160–161.

Декларація
академічної доброчесності
здобувача ступеня вищої освіти ЗНУ

Я, Вуколова Анастасія Ігорівна, студент _2_ курсу, форми навчання денної, Інженерного навчально-наукового інституту, спеціальність 121 Інженерія програмного забезпечення, адреса електронної пошти vukolovaanastasia7@gmail.com, — підтверджую, що написана мною кваліфікаційна робота на тему «**Використання реактивного програмування для оптимізації ресурсів комп'ютера при пікових навантаженнях**» відповідає вимогам академічної доброчесності та не містить порушень, що визначені у ст.42 Закону України «Про освіту», зі змістом яких ознайомлений.

- заявляю, що надана мною для перевірки електронна версія роботи є ідентичною її друкованій версії;

згоден/згодна на перевірку моєї роботи на відповідність критеріям академічної доброчесності у будь-який спосіб, у тому числі за допомогою інтернет-систем, а також на архівування моєї роботи в базі даних цієї системи.

30.11.2020 _____ А.І. Вуколова

30.11.2020 _____ Н.П. Полякова