ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

НАВЧАЛЬНО–НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра інженерії програмного забезпечення

Пояснювальна записка

до магістерської роботи
на ступінь вищої освіти магістр

на тему: **«ЗАСТОСУВАННЯ МЕТОДОЛОГІЇ РЕАКТИВНОГО ПІДХОДУ ПРИ РОЗРОБЦІ ВИСОКОНАВАНТАЖЕНИХ СИСТЕМ ІЗ ЗАСТОСУВАННЯМ PROJECT REACTOR JAVA»**

Виконав: студент 6 курсу, групи ПДМ–61
спеціальності
121 Інженерія програмного забезпечення
(шифр і назва спеціальності/спеціалізації)

Деревець А.В.
(прізвище та ініціали)

Керівник Шевченко С.М.
(прізвище та ініціали)

Рецензент _____
(прізвище та ініціали)

Київ –2022

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ
НАВЧАЛЬНО–НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра Інженерії програмного забезпечення
Ступінь вищої освіти – «Магістр»
Спеціальність підготовки – 121 «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного забезпечення
_____Негоденко О.В.
“____” _____2022 року

З А В Д А Н Н Я
НА МАГІСТЕРСЬКУ РОБОТУ СТУДЕНТА

Деревцю Артему Валерійовичу

(прізвище, ім’я, по батькові)

1. Тема роботи: «Застосування методології реактивного підходу при розробці високонавантажених систем із застосуванням Project Reactor Java»

Керівник роботи: Шевченко С.М., к.п.н., доцент
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

Затверджені наказом закладу вищої освіти від «11» жовтня 2021 року №170.
2. Строк подання студентом роботи
3. Вхідні дані до роботи.
Науково–технічна література з теми дослідження;
Технології високонавантажених систем;
Мова програмування Java
4. Зміст розрахунково–пояснювальної записки(перелік питань, які потрібно розробити).
4.1 Порівняльний аналіз технологій високонавантажених систем
4.2 Вимоги та оцінка якості системи.
4.3 Розробка тестів навантаження та визначення критеріїв для оцінки системи.
4.4 Опис результатів.

5. Перелік демонстраційного матеріалу (назва основних слайдів)

1. Актуальність проблеми
2. Аналіз технологій високонавантажених систем
3. Критерії до розробки високонавантажених систем
4. Розробка тестів навантаження для експериментальної частини
5. Результати експериментальної роботи

6. Дата видачі завдання_____

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської роботи	Строк виконання етапів роботи	Примітка
1	Підбір науково–технічної літератури		
2	Вимоги до результатів дослідження		
3	Розробка додатків для дослідження		
4	Розробка тестів навантаження та визначення критеріїв для оцінки системи		
5	Проведення дослідження при різних параметрах навантаження та обробка отриманих результатів		
6	Вступ, висновки, реферат		
7	Розробка обов‘язкових демонстраційних матеріалів		
8	Попередній захист роботи		
9	Здача роботи		

Студент _____ _____

(підпис) (прізвище та ініціали)

Керівник роботи _____ _____

(підпис) (прізвище та ініціали)

РЕФЕРАТ

Текстова частина магістерської роботи: _85_ с.., _30_рис., _6_ дод., _22_ джерел.

РЕАКТИВНИЙ ПІДХІД, ВИСОКОНАВАНТАЖЕНІ СИСТЕМИ, ТЕХНОЛОГІЯ SPRING MVC, ТЕХНОЛОГІЯ VERTX, ТЕХНОЛОГІЯ SPRING WEBFLUX (PROJECT REACTOR), КРИТЕРІЇ РОЗРОБКИ ВИСОКОНАВАНТАЖЕНИХ СИСТЕМ.

Об'єкт дослідження – функціювання високонавантажених систем на Java.

Предмет дослідження – high load системи на базі Project Reactor Java.

Мета роботи – удосконалення методології реактивного підходу на базі Project Reactor Java.

Методи дослідження – методи теорії інформації та системного аналізу застосовувалися для дослідження технологій високонавантажених систем; методи моделювання, теорії надійності та тестування використовували для підтвердження основних положень теоретичних досліджень.

Дане дослідження присвячено проблемі розробки високонавантажених систем на Java. З метою вироблення принципів стандартизування до розробок таких систем у роботі проаналізовані існуючі технології, а саме технологія Spring MVC, технологія Vertx, технологія Spring WebFlux. Визначені вимоги до розробок та критерії, на основі яких проведена експериментальна робота по дослідженню поведінки високонавантажених систем при різних показниках навантаження. Розроблено тести навантаження для систем, які виконують однакові функції, але написані за допомогою різних підходів. Доведено, що у разі великої кількості запитів до системи, швидкодія та надійність технології Project Reactor має найкращі показники, що підтверджено за допомогою метрик і візуально описано графіками.

Галузь використання результатів дослідження – розробка програмного забезпечення високонавантажених систем.

ЗМІСТ

ВСТУП

Під час розробки систем компанії розроблюють систему під конкретні вимоги, які ставляться на початку використання продукту. Дуже часто при розробці систем не враховеться багата факторів, які з'являться при реальному використанні системи. Буває так, що система розроблена правильно, але вона не витримує одночасно мільйон користувачів, які використовують цю систему для своїх потреб. Це є проблемою функіональних вимог до системи, які дуже часто не враховуються при проектуванні системи.

Для пришвидшення роботи системи, збільшення рівню надійності, а також витримування більших об'ємів даних через систему та їх обробки необхідно розробити підходи, які будуть описувати проблеми та способи їх вирішення через аналіз та моніторинг системи при роботі з системами.

На даний момент існує досить багато продуктів, які займаються розробкою інструментів для вирішення проблеми розробки високонавантажених систем, але поріг входження в таку розбоку високий і необхідно вивчити дуже багато аспектів.

У зв'язку з цифровізацією суспільства, проблема автоматичного вилучення значень з електронних документів може виникати у підприємств будь-якої галузі. На даний час, для вилучення даних часто використовується робота звичайних операторів ПК, що займає багато часу. Через людський фактор не виключається і наявність помилок у вилучених даних.

Перед існуючими системами все частіше постає завдання розроблювати систему таким чином, щоб можна було витримувати навантаження і це не було критичним для замовника. Все частіше замовник хоче мати систему, яка машстабується в залежності від навантаження і може використовувати тільки ті ресурси, які необхідні в конкретний момент часу.

Об'єкт дослідження – функціювання високонавантажених систем на Java.

Предмет дослідження – high load системи на базі Project Reactor Java.

Мета роботи – удосконалення методології реактивного підходу на базі Project Reactor Java.

Методи дослідження – методи теорії інформації та системного аналізу застосовувалися для дослідження технологій високонавантажених систем; методи моделювання, теорії надійності та тестування використовували для підтвердження основних положень теоретичних досліджень.

Наукова новизна роботи. Проведено порівняльний аналіз систем, написаних з використанням блокуючого Sevlet API, і систем, які використовують асинхронний неблокуючий підхід на базі Event Loop; досліджено і представлено їх характеристики, виділено слабкі та сильні сторони всіх підходів. Проведено аналіз працездатності систем при різних показниках навантаження. Результати підтверджено за допомогою метрик, які показують поведінку систем при різному навантаженні, та візуально описано графіками.

Практична значущість результатів. Розроблено програмне забезпечення систем, які виконують однакові функції, але написані за допомогою різних підходів. Створено тести навантаження для перевірки роботи створених систем.

Апробація результатів магістерської роботи.

Методологія реактивного підходу при розробці високонавантажених систем із застосуванням Project Reactor Java // Конференція «Світ телекомунікації та інформатизації», Збірник матеріалів XIII Міжнародної науково-технічної конференції студентства та молоді, 21.10.21, Київ, ДУТ.

Публікації. Високонавантажені системи: суть, вимоги, критерії до розробки // Зв'язок, ДУТ

1. АНАЛІЗ СУЧАСНОГО СТАНУ В ТЕОРІЇ І ПРАКТИЦІ ВИСОКО НАВАНТАЖЕНИХ СИСТЕМ

1.1 Високо навантажені системи, їх суть і характеристики

Програмісти, системні архітектори та інші IT – інженери, використовуючи слово highload (хайлоад), зазвичай мають на увазі: навантаження, високонавантажена програма. Highload у ролі слова “перевантаження” вживається рідко, найчастіше можна почути інше модне слово – overhead (оверхед). Але все одно подібне використання має місце. Зараз слово highload найчастіше використовують у виразі “розробка highload систем”, що означає створення навантажених додатків. Сама по собі високонавантажена програма – це не просто шматок коду, який може витримувати велику кількість умовного навантаження (запитів, відвідувачів), а це ціла грамотно спланована IT–інфраструктура. Це може бути як один сервер, так і мережа серверів, пов'язана між собою, Це є унікальним для конкретного бізнесу.

Різниця між програмою та цілою інфраструктурою дуже проста – останнє можна легко горизонтально масштабувати до будь–якої теоретично досяжної кількості клієнтів, запитів або будь–якого іншого показника. Коли проект вважається високонавантаженим? Питання, яке хвилює багатьох розробників та адміністраторів – коли закінчується «звичайний» проект і починається highload? Як тільки сайт або сервіс перестає справлятися зі зростаючим навантаженням – перед розробниками проекту постають завдання як підтримувати навантаження в пікові години.

Це може бути як невеликий сайт на бюджетному віртуальному сервері з 256 Мб оперативної пам'яті і 10 запитами в хвилину, так і жирний сервіс, що крутиться на повноцінному сервері з 256 Гб оперативної пам'яті і обробляє сотні тисяч запитів за хвилину. І в тому й іншому випадку потрібно починати вживати заходів щодо прискорення свого програмного продукту. Повільні сайти та сервіси нікому вже не потрібні!

Відрізнятимуться лише методи оптимізації та масштабування. Якщо у разі найслабшого тарифу хостингу проблема вирішиться переходом на наступний тарифний план, то у разі топового виділеного сервера доведеться купувати ще одну дорогу серверну стійку, або займатися низькорівневим тюнінгом і оптимізаціями або взагалі переписування проекту вкористовуючи більш швидкі підходи.

Ще один критерій, за яким можна відрізнити highload–проекти, – це вартість хвилини простою системи. Наприклад, ціна хвилини простою великого інтернет–магазину може становити сотні тисяч карбованців, так як покупці навряд чи чекатимуть, поки магазин знову запрацює, набагато простіше зробити замовлення в іншому магазині. А якщо при цьому покупець перейшов з платної реклами, а сайт недоступний – витрачені рекламні бюджети стають прямим збитком. Отже, який сайт вважається високонавантаженим? Наприклад той, у якому використання нового не оптимізованого функціоналу вимагатиме подвоїти кількість серверів для роботи сайту, тим самим збільшивши витрати. Або зворотна ситуація, коли оптимізація будь–якого компонента системи дозволить скоротити навантаження на залізо, що дозволить знизити кількість серверів, тим самим заощадити бюджет.

З іншого боку, будь–який сервіс прогнозу погоди має набагато меншу вартість простою. Навряд чи через відсутність погодного віджету або показ не найактуальніших даних з кешу будь–яка компанія зазнає якихось значних збитків. Навіть якщо такий сервіс має десятки та сотні тисяч звернень за секунду. Чи буде такий сервіс вважатися високонавантаженим?

Під визначення хайлоад можуть підпадати зовсім різні проекти з навантаженням, що відрізняється на порядки. Навіть подібні послуги в рамках різних компаній в одному випадку будуть високонавантаженим, а в іншому – ні. Вгадати заздалегідь потрапить той чи інший проект у зал слави highload неможливо, як би того не хотіли замовники, інвестори та розробники.

Багато роботодавців, а точніше технічних директорів, архітекторів і тимлідів хочуть набрати в свої команди найкращих фахівців за мінімальні кошти. Часто однією з важливих вимог у своїх вакансіях вказують досвід роботи з високими

навантаженнями від 10 000 запитів на хвилину. Але чи такому фахівцеві буде цікаво працювати над таким проектом, особливо якщо його доля ще не ясна, а високі навантаження ще далеко попереду?

Поганий той керівник, який не вірить у успіх свого проекту і не мріє високою відвідуваністю з не менш високим прибутком. Однак, займатися всілякими оптимізаціями та написанням ідеального і в той же час максимально швидкого коду може бути марнуванням часу. Адже не відомо, як відреагують споживачі на ваш проект. Історія знає багато випадків, коли найкращі та оптимізовані проекти не викликали жодного інтересу у спільноти та закривалися з мільйонними збитками.

Спочатку потрібно зрозуміти одну просту аксіому: високі навантаження – поняття відносне. Їх не можна виміряти кількістю запитів, що надходять на сервер або швидкістю роботи веб–сайту. Адже жодної «середньої» кількості запитів, як і «середнього» сайту, не існує. Один веб–ресурс зможе нормально обробляти тисячу запитів за секунду, а інший обвалиться на сотому підключенні. Отже, справа тут зовсім не в кількісних показниках.

Багато програм сьогодні відносяться до категорії високонавантажених даними (data–intensive), на відміну від високонавантажених обчислень (compute–intensive). Чиста продуктивність CPU – просто обмежуючий фактор для цих додатків, а основна проблема полягає в обсязі даних, їх складності та швидкості зміни. Високонавантажена даними додаток (DIA) зазвичай створюється зі стандартних блоків, що забезпечують функцію, що часто потрібна. Наприклад, багатьом програмам потрібно:

- зберігати дані, щоб ці або інші програми могли знайти їх надалі (бази даних);
- запам'ятовувати результат ресурсомісткої операції для прискорення читання (кеші);
- надавати користувачам можливість шукати дані за ключовим словом або фільтрувати їх у різний спосіб (пошукові індекси);
- надсилати повідомлення іншим процесам для асинхронної обробки (потокова обробка);

- іноді «перемелювати» великі обсяги накопичених даних (пакетна обробка).

Все описане виглядає дуже очевидним лише тому, що інформаційні системи дуже вдала абстракція: Ми використовуємо їх весь час, навіть не замислюючись. При створенні програми більшість розробників і не думають про створення з нуля нової підсистеми зберігання, оскільки бази даних – інструмент, який чудово підходить для цього завдання.

Але у житті не все так просто. Існує безліч систем баз даних з різноманітними характеристиками, оскільки у різних додатків різні вимоги. Існує багато підходів до кешування, кілька способів побудови пошукових індексів і т.д. При створенні програми необхідно визначитися з тим, за допомогою яких інструментів та підходів найкраще вирішувати існуюче завдання. Крім того, іноді буває непросто підібрати потрібну комбінацію інструментів, коли потрібно зробити систему. Ця робота проведе великий екскурс у світ як теоретичних принципів, так і практичних аспектів інформаційних систем, а також можливостей їх використання для створення високонавантажених даними додатків.

У цьому розділі буде розглянуто основи того, чого хочемо досягти: масштабованості та зручності супроводу інформаційних систем, що означають зазначені поняття, окреслимо окремі підходи до роботи з інформаційними системами. У наступних розділах ми продовжимо наш пошаровий аналіз, розглянемо різні проектні рішення, які доцільно взяти до уваги під час роботи над DIA.

1.2 Існуючі підходи при розробці навантажених систем на Java

Зазвичай бази даних, черги, кеші тощо вудносяться до абсолютно різних категорій інструментів. Хоча бази даних та черги повідомлень виглядають схожими – і ті та інші зберігають дані протягом деякого часу, – патерни доступу їм абсолютно різняться, що означає різні характеристики продуктивності, отже, дуже різні реалізації.

Так навіщо ж валити їх усі в одну купу під таким збірним терміном, як інформаційні системи? В останні роки з'явилося багато нових інструментів для зберігання та обробки даних. Вони оптимізовані для безлічі різних сценаріїв використання і не вкладаються у звичайні категорії [1]. Наприклад, існують сховища даних, що застосовуються як черги повідомлень (Redis) та черги повідомлень з відповідним базам даних рівнем надійності (Apache Kafka). Кордони між категоріями поступово розмиваються.

Крім того, все більше додатків пред'являють такі жорсткі чи широкі вимоги, що окрема утиліта вже не здатна забезпечити усі їхні потреби в обробці та зберіганні даних. Тому робота розбивається на окремі завдання, які можна ефективно виконати за допомогою окремого інструменту, та ці різні інструменти поєднуються кодом програми.

Наприклад, за наявності керованого додатком шару кешування (шляхом використання Memcached або аналогічного інструменту) або сервера повнотекстового пошуку (такого як Elasticsearch або Solr), окремо від бази даних, синхронізація цих кешів та індексів з основною базою даних стає обов'язком коду програми. На рисунку 1.1 загалом показано, як усе зазначене могло б виглядати.

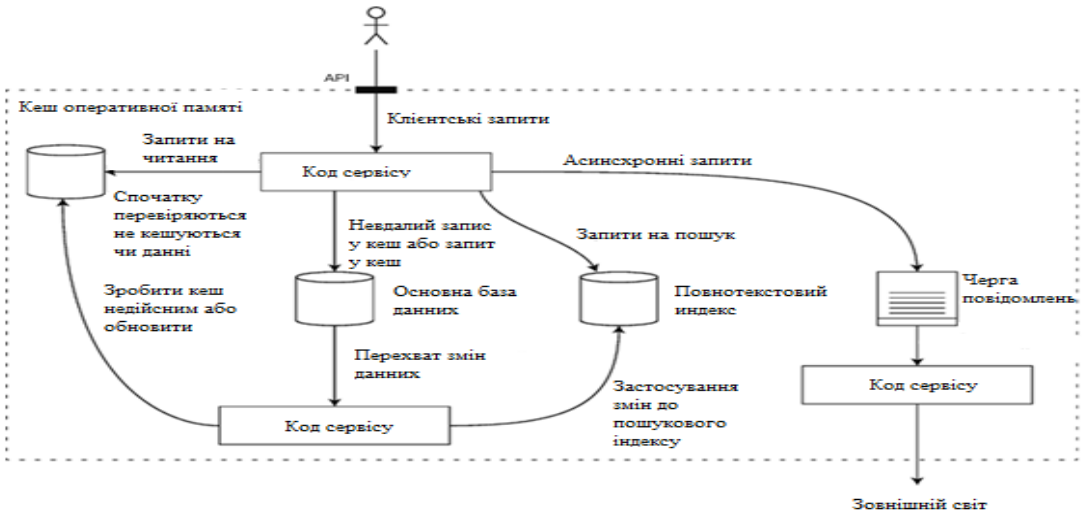


Рисунок 1.1 Приклад схеми високонавантаженої системи

Якщо для надання сервісу поєднується кілька інструментів, то інтерфейс сервісу або програмний інтерфейс програми (API) зазвичай приховує подробиці реалізації від клієнтських програм. Фактично можна створити нову спеціалізовану інформаційну систему з дрібніших, універсальних компонентів. Об'єднана інформаційна система, що вийшла може гарантувати певні речі: наприклад, що кеш буде коректним зроблений недійсним або оновлений під час запису, внаслідок чого зовнішні клієнти побачать несуперечливі результати. Ви тепер не лише розробник програми, а й архітектор інформаційної системи.

При проектуванні інформаційної системи чи сервісу виникає багато непростих питань. Як забезпечити правильність та повноту даних, у тому чисельності при внутрішніх помилках? Як забезпечити однаково хорошу продуктивність для всіх клієнтів навіть у разі погіршення робочих характеристик Деякі частини системи? Як забезпечити масштабування для обліку зростання навантаження? Яким має бути добрий API для цього сервісу?

Існує безліч факторів, що впливають на конструкцію інформаційну системи, включаючи навички та досвід залучених у проектування фахівців, успадковані системні залежності, терміни постачання, ступінь прийнятності різних видів ризику для вашої компанії, законодавчі обмеження та ін. Ці фактори дуже залежать від конкретної ситуації.

1.2.1 Підходи для рішення проблем як впоратися з навантаженням

Обговоривши параметри для опису навантаження та метрики для оцінки продуктивності, можна всерйоз перейти до вивчення питання масштабованості: як зберегти хорошу продуктивність навіть у разі певного збільшення параметрів навантаження? Архітектура, що підходить для певного рівня навантаження, ймовірно, не зможе впоратися із десятикратним її збільшенням. Якщо мати справу з сервісом, що швидко росте, то, ймовірно, доведеться переглядати архітектуру при кожному зростанні навантаження на порядок або навіть ще частіше.

Найчастіше проводять диференціацію між вертикальним масштабуванням – переходом більш потужну машину – і горизонтальним масштабуванням розподілом навантаження з кількох меншим машинам. Розподіл навантаження по кількох машинах відомо також під назвою архітектури, що не передбачає поділу ресурсів. Системи, які здатні працювати на окремій машині зазвичай простіше, а висококласні машини можуть виявитися дуже недешевими, так що при високому робочому навантаженні часто не можна уникнути горизонтальне масштабування. Насправді хороша архітектура зазвичай є прагматичну суміш цих підходів: наприклад, може виявитися простіше і дешевше використовувати кілька потужних комп'ютерів, ніж багато маленькі віртуальні машини.

Деякі системи здатні адаптуватися, тобто вміють автоматично додавати обчислювальні ресурси при виявленні приросту навантаження, тоді як інші системи необхідно масштабувати вручну (людина аналізує продуктивність і вирішує, чи потрібно додати до системи додаткові машини). Здатні до адаптації системи корисні у разі непередбачуваного характеру навантаження, але вручну системи, що масштабуються, простіше і доставляють менше несподіванок при експлуатації.

Хоча розподіл сервісів без збереження стану за кількома машинами особливих складнощів не становить, перетворення інформаційних систем із збереженням стану з одновузлових у розподілені може спричинити значні складності. Тому донедавна вважалося розумним тримати базу даних одному вузлі (вертикальне масштабування) до того часу, поки вартість масштабування чи вимоги з високої доступності не змусять зробити її розподіленої.

У міру вдосконалення інструментів та абстракцій для розподілених систем ця думка зазнає змін, принаймні для окремих видів додатків. Цілком можливо, що розподілені інформаційні системи в майбутньому стануть «золотим стандартом» навіть для сценаріїв застосування, в яких не йдеться про обробку великих обсягів даних або трафіку. Архітектура великомасштабних систем зазвичай дуже сильно залежить від додатка – не існує такої речі, як одна архітектура, що масштабується,

на всі випадки життя (на сленгу іменована чарівним масштабуючим соусом). Проблема може полягати в кількості читань, кількості записів, об'ємі даних, їх складності, вимогах до часу відгуку, патернах доступу або (часто) будь–якої суміші всього перерахованого, а також багато в чому іншому. Наприклад, система, розрахована на обробку 100 000 з/с по 1 Кбайт кожен, виглядає зовсім не так, як система, розрахована на обробку 3 з/хв. по 2 Гбайт кожен – хоча пропускна здатність обох систем у сенсі обсягу даних однакова.

Хороша архітектура, що масштабується для конкретного додатка, базується на припущеннях про те, які операції будуть виконуватися часто, а які – рідко, тобто на параметрах навантаження. Якщо ці припущення виявляться невірними, то робота архітекторів масштабування виявиться в кращому разі марною, а в гіршому призведе до зворотних результатів. На ранніх стадіях зазвичай важливіша швидка робота існуючих можливостей у дослідному зразку системи або неперевіреному програмному продукті, ніж його масштабованість під гіпотетичне майбутнє навантаження.

Незважаючи на залежність від конкретної програми, архітектури, що масштабуються, зазвичай створюються на основі універсальних блоків, організованих за добре відомими патернами.

1.2.2 Необхідність супроводження систем

Широко відомо, що вартість програмного забезпечення складається здебільшого із витрат не на початкову розробку, а на поточний супровід, виправлення помилок, підтримання працездатності його підсистем, розслідування відмов, адаптацію до нових платформ, модифікацію під нові сценарії використання, повернення технічного боргу та додавання нових можливостей.

Проте, на жаль, багато людей, які працюють над програмними системами, ненавидять супровід так званих успадкованих систем мабуть, тому, що доводиться виправляти чужі помилки або працювати зі застарілими платформами або системами, які змусили робити те, для чого вони ніколи не були призначені. Кожна

успадкована система неприємна по–своєму, тому так складно дати якісь загальні рекомендації про те, що з ними робити.

Однак можна і потрібно проектувати програмне забезпечення так, щоб максимально мінімізувати «головний біль» під час супроводу, а отже, уникати створення успадкованих систем своїми руками. Відштовхуючись від міркування, приділимо особливу увагу трьом принципам проектування програмних систем:

- Зручність експлуатації. Полегшує обслуговуючому персоналу підтримання безперешкодної роботи системи.
- Простота. Полегшує розуміння системи новими інженерами шляхом максимально можливого її спрощення. (Зверніть увагу: це не те ж саме, що простота інтерфейсу користувача.)
- Можливість розвитку. Спрощує розробникам внесення у майбутньому змін до системи, адаптацію її для непередбачених сценаріїв використання під час зміни вимог. Відома під назвами "розширюваність" (extensibility), «модифікованість» (modifiability) та «пластичність» (plasticity).

Як і у разі надійності та масштабованості, не існує простих рішень для досягнення цих цілей. Слід мати на увазі зручність експлуатації, простоту і можливість розвитку при проектуванні систем.

1.2.3 Вимоги для зручної експлуатації високонавантаженої системи

Вважається, що «хороший обслуговуючий персонал часто може обійти обмеження поганого (або недосконалого) програмного забезпечення, але хороше програмне забезпечення не здатне надійно працювати під управлінням поганих операторів» [2]. Хоча деякі аспекти експлуатації можна і потрібно автоматизувати, виконання цієї автоматизації та перевірка правильності її роботи все одно залишається насамперед завданням обслуговуючого персоналу.

Цей персонал важливий для безперебійної роботи програмної системи. Хороша команда операторів зазвичай відповідає за пункти, наведені нижче, і не тільки [3]:

- моніторинг стану системи та відновлення сервісу у разі його погіршення;
- з'ясування причин проблем, наприклад, відмов системи або зниження продуктивності;
- підтримання актуальності програмного забезпечення та платформ, включаючи виправлення безпеки;
- відстеження впливу різних систем одна на одну, щоб уникнути проблемних змін до того, як вони завдадуть шкоди;
- попередження та вирішення можливих проблем до їх виникнення (наприклад, планування продуктивності);
- введення в експлуатацію рекомендованих практик та інструментів для розгортання, керування конфігурацією тощо;
- виконання складних робіт із супроводу, наприклад перенесення додатку з однієї платформи на іншу;
- підтримка безпеки системи при змінах конфігурації;
- формування процесів, які б забезпечили прогнозованість операцій та стабільність операційного середовища;
- збереження знань організації про систему, незважаючи на відхід старих працівників та прихід нових.

Зручність експлуатації означає полегшення виконання стандартних завдань, завдяки чому обслуговуючий персонал може зосередити зусилля на чомусь важливішому. Інформаційні системи здатні робити багато для полегшення виконання стандартних завдань, у тому числі:

- забезпечують хороший моніторинг та надають інформацію про поведінку системи та те, що відбувається в ній під час роботи;
- забезпечують хорошу підтримку автоматизації та інтеграції зі стандартними утилітами;

- дозволяють не залежати від окремих машин (завдяки чому можна відключати деякі з них для технічного обслуговування за умови збереження безперебійної роботи системи в цілому);
- надають якісну документацію та зрозумілу операційну модель («якщо я виконаю дію X, то в результаті відбудеться дія Y»);
- забезпечують розумну поведінку за замовчуванням, але разом з тим і можливості для адміністраторів за необхідності встановити стандартні налаштування;
- запускають самовідновлення в міру можливості, але водночас і дозволяють адміністраторам вручну керувати станом системи за необхідності;
- демонструють передбачувану поведінку, мінімізуючи несподіванки.

Код невеликих програмних проектів може бути чудово простим і виразним, але в міру зростання проекту може стати дуже складним і важким для розуміння. Подібна складність уповільнює роботу над системою, ще збільшуючи вартість супроводу. Проект, що загрузнув у складності, іноді описують як великий ком бруду [4].

Існують різні можливі симптоми зайвої складності: стрибко–образне зростання простору станів, тісне зчеплення модулів, заплутані
залежності, неузгоджені найменування та термінологія, «милиці» для вирішення проблем з продуктивністю, виділення окремих випадків для обходу проблем та багато іншого. У літературі про це було сказано чимало [5–8].

Коли складність досягає рівня, що дуже ускладнює супровід, часто відбувається перевищення бюджетів та зрив термінів. У складному програмному забезпеченні збільшується і шанс внесення помилок під час виконання змін: коли розробникам складніше розуміти систему та обговорювати її, набагато простіше упустити будь–які приховані припущення, ненавмисні наслідки та неочікувані взаємодії. І навпаки, зниження складності різко підвищує зручність супроводу ПЗ, а отже, простота має бути основною метою створюваних систем.

Спрощення системи обов'язково означає скорочення її функціональності. Воно може означати виняток побічної складності. Мозлі та Маркс [9] визначають складність як побічну, якщо вона виникає внаслідок конкретної реалізації, а не є невід'ємною частиною розв'язуваного програмного забезпечення завдання (з погляду користувачів).

Один із найкращих інструментів для виключення побічної складності це – абстракція. Хороша абстракція дозволяє приховати більшу частину подробиць реалізації за акуратним та зрозумілим фасадом. Хорошу абстракцію можна також використовувати для широкого діапазону різних додатків. Таке багаторазове застосування не тільки ефективніше реалізації заново одного і того ж кілька разів, а й призводить до більш якісного ПЗ, у міру вдосконалення абстрактного компонента, що використовується всіма додатками.

Наприклад, високо–рівневі мови програмування – абстракції, що приховують машинний код, регістри CPU та системні виклики. SQL – теж абстракція, що приховує складні структури даних, що знаходяться на диску і в оперативній пам'яті, конкурентні запити від інших клієнтів і незгоди, що виникають після фатальних збоїв. Звичайно, при створенні продукту за допомогою мови програмування високого рівня ми, як і раніше, використовуємо машинний код, просто не задіємо його безпосередньо, оскільки абстракція мови програмування дозволяє не замислюватися про це.

Проте знайти хорошу абстракцію дуже непросто. У сфері розподілених систем, незважаючи на наявність безлічі хороших алгоритмів, далеко не так ясно, як об'єднати їх в абстракції, які дали б можливість зберегти прийнятний рівень складності системи. Імовірність того, що системні вимоги назавжди залишаться незмінними, прагне до нуля. Набагато ймовірніше їх постійне перетворення: відкриватимуться нові факти, виникатимуть непередбачені сценарії застосування, змінюватимуться комерційні пріоритети, користувачі вимагатимуть нові можливості, нові платформи – замінюватимуть старі, змінюватимуться правові та нормативні вимоги, зростання системи потребуватиме архітектурних змін тощо. .

У термінах організаційних процесів робочі патерни Agile забезпечують інфраструктуру адаптації до змін. Спільнота творців Agile розробила також технічні інструменти і патерни, корисні при проектуванні програмного забезпечення в середовищі, що часто змінюється, наприклад при розробці через тестування (test–driven development, TDD) та рефакторингу.

Більшість обговорень цих методів Agile обмежується досить невеликими локальними масштабами (пара файлів вихідного коду в одному додатку).

Ступінь зручності модифікації інформаційної системи та адаптації її до вимог, що змінюються, тісно пов'язана з її простотою та абстракціями: прості та зрозумілі системи зазвичай легше змінювати, ніж складні. Але через виняткову важливість цього поняття використовується інший термін для швидкості адаптації на рівні інформаційних систем – можливість розвитку (evolvability) [10].

1.3 Огляд технології Spring MVC для розробки “highload” системи

Фреймворк Spring MVC забезпечує архітектуру патерна Model – View – Controller (Модель – Відображення (далі – Вид) – Контролер) за допомогою слабко пов'язаних готових компонентів. Паттерн MVC поділяє аспекти програми (логіку введення, бізнес–логіку та логіку UI), забезпечуючи вільний зв'язок між ними. Фреймворк складається з таких частин:

- Model (Модель) інкапсулює (об'єднує) дані програми, в цілому вони будуть складатися з POJO («Старих добрих Java–об'єктів», або бінів).
- View (Відображення, Вид) відповідає за відображення даних Моделі, як правило, генеруючи HTML, які бачимо у своєму браузері.
- Controller (Контролер) обробляє запит користувача, створює відповідну Модель і передає її для відображення.

Вся логіка роботи Spring MVC побудована навколо DispatcherServlet, який приймає та обробляє всі HTTP–запити (з UI) та відповіді на них. Робочий процес

обробки запиту DispatcherServlet'ом проілюстровано на рис 1.2. Нижче наведено послідовність подій, що відповідає вхідному HTTP–запиту:

- Після отримання HTTP–запиту DispatcherServlet звертається до інтерфейсу HandlerMapping, який визначає, який Контролер має бути викликаний, після чого надсилає запит у потрібний Контролер.
- Контролер приймає запит і викликає відповідний службовий метод, що базується на GET або POST. Викликаний метод визначає дані Моделі, що базуються на певній бізнес–логіці та повертає в DispatcherServlet ім'я Вида (View).
- За допомогою інтерфейсу ViewResolver DispatcherServlet визначає, який Вид потрібно використовувати на підставі отриманого імені.
- Після того, як Вид (View) створено, DispatcherServlet відправляє дані Моделі у вигляді атрибутів у Вид, який зрештою відображається в браузері.

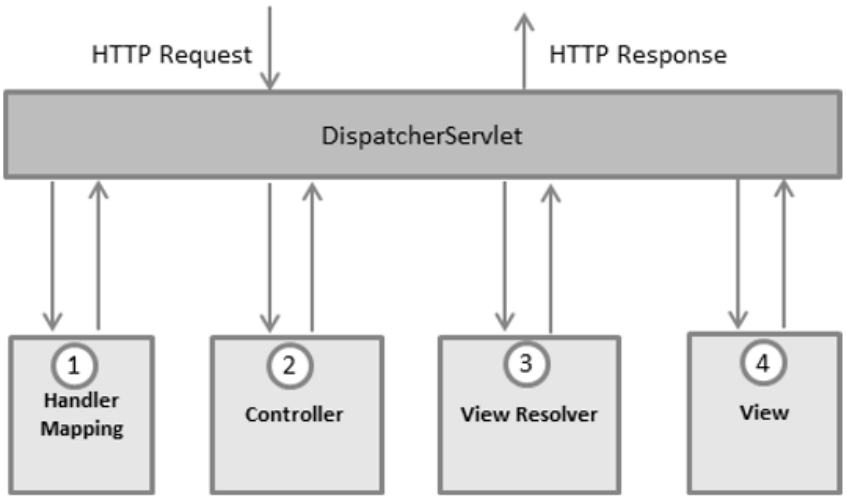


Рисунок 1.2 Схема приймання запитів у Spring MVC

При написанні веб–додатків на Java з використанням Spring або без нього (MVC/Boot) в основному йдеться про написання додатків, які повертають два різні формати даних:

- HTML → Ваша веб – програма створює HTML–сторінки, які можна переглядати у браузері.
- JSON/XML → Ваш веб – програма надає сервіси RESTful, які генерують JSON або XML. Сайти з великою кількістю Javascript або навіть інші веб–сервіси можуть використовувати дані, які надають ці сервіси.

Так, є й інші формати даних та варіанти використання, їх ігноруємо. На найнижчому рівні кожна веб–програма Java складається з одного або декількох HttpServlets. Вони генерують ваш HTML, JSON або XML. Фактично кожен окремий фреймворк з 1 мільйона доступних веб–фреймворків на Java (Spring MVC, Wicket, Struts) побудована на основі HttpServlets. Як вже згадувалося вище, багато веб–фреймворки Java засновані на сервлетах, тому Spring MVC також потрібен сервлет, який обробляє кожен вхідний HTTP–запит (тому DispatcherServlet також називається фронт–контролером).

Що ж точно означає обробляти HTTP–запит, точно? Потрібно уявити «робочий процес реєстрації користувача», при якому користувач заповнює форму та відправляє її на сервер та у відповідь отримує невелику HTML сторінку про успішну реєстрацію. У цьому випадку DispatcherServlet повинен виконати такі дії:

- Необхідно переглянути URI вхідного запиту HTTP та будь–які параметри запиту. Наприклад: POST/register?name=john&age33.
- Він повинен потенційно перетворювати вхідні дані (параметри/тіло запиту) в симпатичні маленькі об'єкти Java і перенаправити їх у клас контролер або REST контролер, який ви написали.
- Ваш контролер зберігає нового користувача в базі даних, можливо надсилає електронного листа і т.д. Він, швидше за все, делегує це іншому сервісному класу, але припустимо, що це відбувається всередині контролера.
- Він повинен взяти будь–який висновок з вашого контролера і перетворити його назад на HTML/JSON/XML.

Весь процес виглядає таким чином, знехтувавши для простоти великою кількістю проміжних класів, тому що DispatcherServlet не виконує всю роботу сам. Діаграма взаємодії при обробці http запиту описано на рис 1.3.

Рисунок 1.3 Діаграма взаємодії опрацювання запиту у Spring MVC

Коли розробляються REST сервіси, все трохи інакше. Ваш клієнт, будь то браузер або інший веб–сервіс, буде (зазвичай) створювати запити JSON або XML. Клієнт відправляє, скажімо, запит JSON, ви обробляєте його, а потім відправник чекає на повернення JSON. Таким чином, відправник може надіслати вам цей фрагмент JSON як частину тіла HTTP–запиту.

Це також означає, що вам не потрібна вся ця обробка моделі та уявлення, які доводилося робити при рендерингу HTML у ваших контролерах. Для RESTful сервісів у вас немає бібліотеки шаблонів, що читає шаблон HTML і заповнює його даними моделі, щоб згенерувати вам відповідь JSON. Натомість потрібно перейти безпосередньо з HTTP запит → Java об'єкт і з Java об'єкт → HTTP відповідь. Це те, що Spring MVC забезпечує при написанні REST контролера. Як написати REST контролер? Перше, що потрібно зробити для виведення XML/JSON, це написати інструкцію @RestController замість Controller. (Хоча @RestController є Controller але який повертає файл формату JSON або XML). Приклад коду представлений в лістингу 1.1.

Лістинг 1.1 Приклад Rest контроллеру написаний на Spring MVC

```
@RestController
public class BankController {
```

```
    @GetMapping("/transactions/{userId}")
    public List<Transaction> transactions(String userId) {
                List<Transaction> transactions = Collections.emptyList();
        return transactions;
    }
```

У порівнянні з HTML використання JSON / XML трохи простіше, тому що не потрібний рендеринг Model та View. Натомість контролери безпосередньо повертають об'єкти Java, які Spring MVC буде зручно серіалізувати в JSON/XML або будь–який інший формат, який користувач запросив за допомогою HttpMessageConverters. Однак ви повинні переконатися у двох речах, однак:

- Є відповідні сторонні бібліотеки на шляху класів.
- Надіслані правильні заголовки Accept або Content–Type з кожним запитом.

1.4 Огляд технології Vertx для розробки “highload” системи

Vert.x – це реактивний інструментарій з відкритим кодом, створений розробниками Eclipse. Це відмінний спосіб створення реактивних програм, які не блокують ресурси. Він підтримує як JVM, так і мови, відмінні від JVM, наприклад Groovy. Переваги Vertx:

- Велика екосистема реактивних моделей – реактивні драйвери баз даних, обмін повідомленнями, потік подій, метрики, трасування тощо.
- Підтримка зворотного виклику, обіцянок/ф'ючерсів для реалізації асинхронних операцій.
- Підтримка пов’язаних програм Kotlin.
- Автоматичні вимикачі для запобігання простоям.
- Підтримка хмарних програм.

Специфікація OpenAPI – це стандартний формат, що визначає структуру та синтаксис REST API. Його головна перевага полягає в тому, що люди і машини можуть інтерпретувати мову без доступу до вихідного коду.

Розробники часто використовують його для проектування прототипів, тестування відповідей на кінцеві точки API, створення документації і т. д. Наприклад, розгляньте можливість отримання yaml файлу конфігурації служби. Не інженери можуть прочитати його та підтвердити бізнес–очікування, а команди фронтенд та бекенд можуть почати працювати незалежно. Таким чином, спільна робота у команді стає простішою.

Vert.x розроблений для реалізації шини подій, яка дозволяє різним частинам програми взаємодіяти неблокуючим/потокобезпечним способом. Частини цього було змодельовано після методології Актора, виставленої Eralng та Akka. Він також призначений для того, щоб повною мірою використати переваги сучасних багатоядерних процесорів та вимоги одночасного програмування. Таким чином, за замовчуванням всі Vert.x VERTICLES за замовчуванням реалізовані як однопотокові. На відміну від NodeJS, Vert.x може виконувати багато вершин у багатьох потоках. Крім того, ви можете вказати, що деякі статті є робітниками і можуть бути багатопотоковими. І щоб дійсно додати певну родзинку в торт, Vert.x має низькорівневу підтримку багатовузлової кластеризації шини подій за допомогою Hazelcast. Він включає безліч інших дивовижних функцій, яких занадто багато, щоб перераховувати тут, але ви можете прочитати більше в офіційних документах Vert.x.

Перше, що потрібно знати про Vert.x, як і NodeJS, ніколи не блокувати поточний потік. Все в Vert.x за замовчуванням налаштовано на використання зворотних викликів/ф'ючерсів/обіцянок. Замість синхронних операцій Vert.x надає асинхронні методи для виконання більшості операцій введення–виведення та інтенсивної роботи процесора, які можуть блокувати поточний потік. Тепер зворотні виклики можуть бути потворними та болючими для роботи, тому Vert.x додатково надає API на основі RxJava, який реалізує ті ж функціональні можливості з використанням шаблону Observer. Vert.x спрощує використання існуючих класів та методів.

В Vertx маршрутизація http запитів здійснюється за допомогою Router. Приклад ініціалізації маршруту вказаний у лістингу 1.2

Лістинг 1.2 Ініціалізація маршруту у Vertx

```
@PostConstruct
public void start() throws Exception {
    Router router = Router.router(vertx);
    router.route().handler(BodyHandler.create());
    router.get("/v1/customer/:id")
            .produces("application/json")
            .blockingHandler(this::getCustomerById);
    router.put("/v1/customer")
            .consumes("application/json")
            .produces("application/json")
            .blockingHandler(this::addCustomer);
    router.get("/v1/customer")
            .produces("application/json")
            .blockingHandler(this::getAllCustomers);
    vertx.createHttpServer().requestHandler(router::accept).listen(8080);
    }
}
```

Друге, що потрібно знати про Vert.x, це те, що він складається з вершин, модулів та вузлів. Vert є найменшою одиницею логіки Vert.x і зазвичай представлені одним класом. Модуль представляє групу пов'язаних функціональних можливостей, які в сукупності можуть представляти цілу програму або тільки частину більшої розподіленої програми.

Vert.x побудований як імплементація вже класичного патерну Reactor із маленькою модифікацією, яку розробники прозвали Multi–Reactor. Щоб зрозуміти патерн Multi–Reactor, достатньо знати відомий патерн Reactor. Класичний Reactor говорить про те, що є певний Event Loop, як правило однопотоковий, який відповідає за обробку подій. Всі запити клієнтів заходять як події. Далі виконується обробник Handler, який підписаний на відповідні події. Буде погано, якщо обробник заблокує Event Loop надовго. Тому довгограючі завдання делегуються Worker–потокам та виконуються, не блокуючи Event Loop. У свою чергу, Multi–

Reactor розширює цей шаблон (паттерн), додаючи ще кілька потоків (додаткові Event Loop–и). Таким чином, формується шина подій (Event Bus), яка вміє масштабуватися під ресурси конкретної машини. Як правило, кількість потоків Event Loop визначається за формулою «кількість ядер процесора*2». Всього Vert.x – це один великий Event Bus, з яким ми спілкуємося за допомогою. Схема паттерну Multi Reactor представлена на рис 1.4.

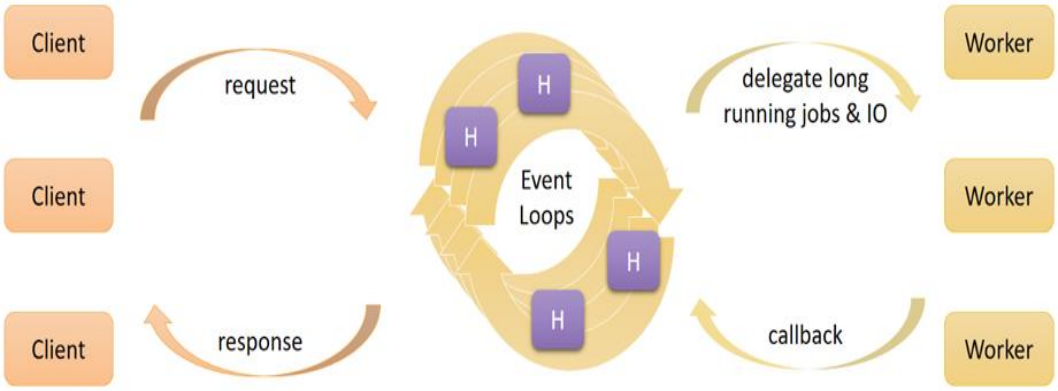


Рисунок 1.4 Паттерн Multi Reactor

Крім реактивної архітектури, він пропонує свою модель розгортки (deployment) додатків. Ця модель називається Verticle. Що це таке? Ще одна адаптація якогось класичного патерну? Чи не повірите, але майже так. Verticle – це контейнер (не Docker, звичайно, це не контейнер для програми). Це клас, який несе в собі якийсь логічний шматок Vert.x коду, частина вашої програми. Трохи далі ми дізнаємося, навіщо потрібне таке збочення. А поки що давайте розберемося, як ця штука працює і як вона взагалі деплоїться.

По суті, Verticle це контейнер для обробників подій (handler). Так як весь код нагадує набір безлічі callback–ів, їх можна логічно зібрати у вертикли і цим структурувати додаток. Насправді, вертикли бувають трьох типів: Standard, Worker та Multi–Threaded Worker. Стандартний вертикл, точніше код усередині нього, виконується у потоці Event Loop, блокуючи його на час виконання. Worker–

вертикли виконуються на Worker–потоках. Але справа в тому, що одноразово один Worker–вертикл може виконуватися тільки на одному потоці. Якщо вам потрібна можливість виконати вертикл паралельно кількох потоках, тоді вам потрібен Multi–Threaded Worker Verticle. Створюються всі ці вертикли дуже просто: потрібно вказати лише тип, лістинг створення вертиклів представлено на лістингу 1.3.

Лістинг 1.3 Приклад створення Verticle

```
DeploymentOptions options = new DeploymentOptions().setWorker(true);
Vert.x.deployVerticle("io.orkhan.MyFirstVerticle", options);
```

Що якщо нам недостатньо однієї програми? Що якщо нам потрібно масштабувати наш додаток на кілька серверів в мережі? Це, звісно, можна зробити стандартними підходами. Однак у випадку Vert.x це завдання також можуть вирішити вертикли. За допомогою вертиклів можна масштабувати програму шляхом кластерування.

Таким чином, можна сказати, що базова структура нашої програми має таку форму яка схематично зображена на рис. 1.5

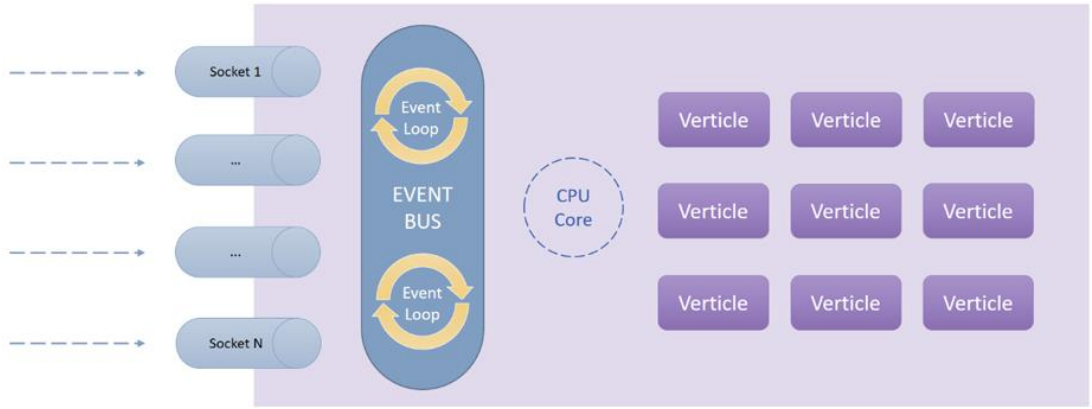


Рисунок 1.5 Схема Vertx системи

В екосистемі Vert.x кластер є надбудовою над готовими рішеннями, такими як Hazelcast, Infinispan, Ignite, Zookeeper, Atomix та інші. Точніше, Vert.x

використовує вищезгадані підсистеми для синхронізації та організації свого кластера. За промовчанням використовується Hazelcast. Інші можна підключити з постачання, крім Atomix, який потрібно окремо підключати (оскільки він є 3rd–party–імплементацією і не входить до Vert.x). У тому числі можуть бути доступні інші варіанти Cluster Manager, що надаються сторонніми постачальниками. Налаштування самого кластер–менеджера, наприклад Hazelcast, доступне в документації Vert.x. Важливо розуміти, що кластер складається з безлічі екземплярів Vert.x–програм, тобто це JVM–програми, в яких запущений Vert.x.

Найголовніше, це не те, що це можна зробити з коду, а те, що це також можна зробити з командного рядка. Це дозволяє спростити автоматизацію процесу розгорнення. Наприклад, командою $vertx run MyVerticle можна просто розгорнути та запустити вертикл. З ключом –ha можна включити режим High Availability, в якому вертикли, що впали, будуть автоматично розгортатися на інших примірниках в кластері. Цей режим особливо цікавий із додатковим ключем –hagroup, який дозволяє розділяти вертикли на групи. Наприклад, якщо різні дата–центри виділити різні групи, вертикли в одному дата–центрі будуть розгортатися тільки на інстансах цього дата–центру.

Також необхідно сказати про балансування навантаження. Один вертикл можна розгортати (deploy) багато разів незалежно від того, запущений він у кластері чи локально. В обох випадках навантаження буде ділитися між запущеними копіями вертикла за алгоритмом Round–Robin (такий собі спрощений load balancing). Модель такого балансування схематично зображена на рис. 1.6.

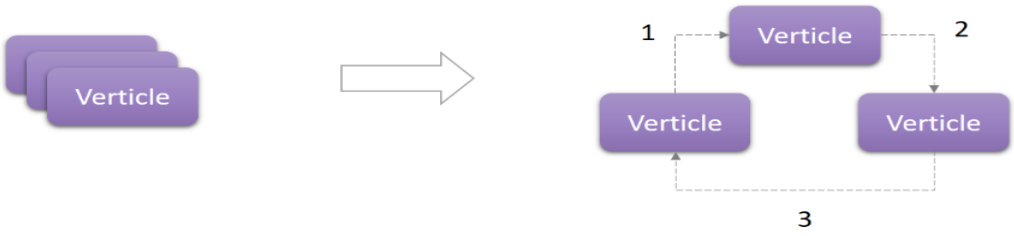


Рисунок 1.6 Схема балансування навантаження Vertx

Можно підсумувати те що Vertx має дуже багато функціоналу для розробки високо навантажених систем. І це не лише одна бібліотека. Адже Vert.x – це ціла екосистема. Далі у статті я наведу стислий огляд інших бібліотек, а детальніше про них можна прочитати в офіційній документації.

Перша яка є обов'язковою для побудови REST API– це Vert.x Web, яка надає цей роутер, використаний у прикладі вище. Справа в тому, що Vert.x Core дозволяє розробляти низькорівневі HTTP–сервери і клієнти. А роутер вже надає можливість розробки веб–сервісів на зручному високому рівні з надбудовами, які полегшують завдання. Наприклад, якщо для розробки HTTP–сервера достатньо одного методу, в коді якого треба буде парсити запит і розуміти, що з ним робити далі, то за допомогою роутера ми можемо розділити GET, POST, PUT та інші запити. У тому числі в Web доступний ще й WebClient, який дозволить досить зручно консьюміть інші веб–сервіси, дозволяючи встановити таймаути, ширяти в обидва боки JSON (під капотом старий добрий jackson) та багато іншого.

Авторизацію та аутентифікацію дозволить зробити Vert.x Auth, що підключається. Він вміє працювати з OAuth2, Shiro, JWT та багатьма іншими. По суті Vert.x Auth інтегрується з роутером з Vert.x Web, що дуже зручно.

Далі за допомогою Vert.x Microservices додаток можна додати розширений service discovery, скористатися вбудованим circuit breaker–ом і отримувати конфігурацію з безлічі доступних джерел. Що мені дуже сподобалося, це те, що з одного боку Vert.x вміє інтегруватися із зовнішнім discovery–сервером, наприклад Consul. З іншого боку, у Vert.x сервісом можна назвати будь–який handler, доступний (підписаний) на event bus, що дозволяє поблишити та діскаверити все, що завгодно.

1.5 Огляд технології Spring WebFlux (Project Reactor) для розробки "highload" системи

Модуль WebFlux з'явився у 5–й версії фреймворку Spring. Цей мікрофреймворк є альтернативою Spring MVC і відображає реактивний підхід для

написання веб–сервісів. В основі WebFlux лежить бібліотека Project Reactor, що дозволяє легко запрограмувати неблокуючі (асинхронні) потоки (streams), що працюють із введенням/виведенням даних.

Рис. 1.7 побудований графік який демострує перевагу у продуктивності реактивних веб–сервісів порівняно із звичайними блокуючими.

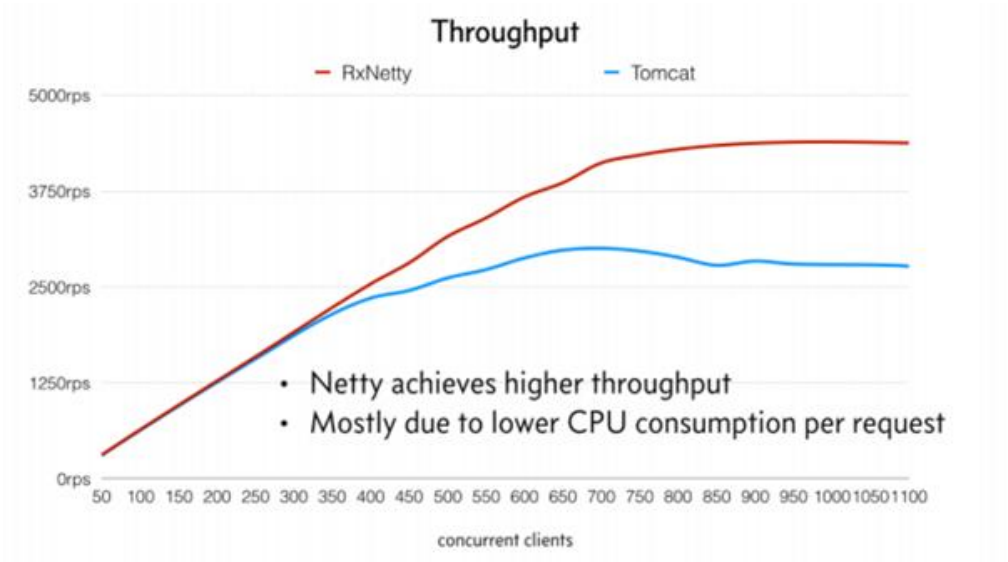


Рисунок 1.7 Графік продуктивності реактивних і блокуючих систем

Слід врахувати, що WebFlux для роботи потрібен інтегрований в Spring сервер Netty. Вбудовані Tomcat та Jetty не підходять. Наступна діаграма ілюструє особливості оточення, у якому працює WebFlux.

При завантаженості сервера в 300 і більше користувачів Netty починає перевершувати Tomcat за кількістю запитів, що одночасно обробляються. При граничній завантаженості сервера в реактивному режимі може одночасно обслуговуватися вдвічі більше користувачів. За іншими джерелами перевага не така переконлива, але помітна. Найбільший ефект реактивне програмування дає за вертикального масштабування.

Важлива особливість реактивності ще тому, що вона дає слабку зв'язність. Наприклад, якщо між клієнтом та сервісом відбувається розрив з'єднання, воно

відтворюється під час відновлення Інтернету. Обмеження в часі не потрібно, так як з'єднання відбувається для окремого потоку, що не впливає на інші. Але при цьому реактивний підхід має бути реалізований на обох сторонах.

Програмувати програму з безліччю потоків, що не блокують, досить незвичне завдання. Потрібний особливий підхід та стиль програмування, заснований на лямбда–виразах з використанням реактивних бібліотек. Бібліотека Project Reactor, що входить у WebFlux, відрізняється від реактивної бібліотеки RxJava (реалізованої, наприклад, Android) тим, що більше підходить для бекенд. Наприклад, усунуто деякі проблеми, які можуть спричинити нестачу пам'яті. Модуль Spring Webflux схематично зображено на рис 1.8.

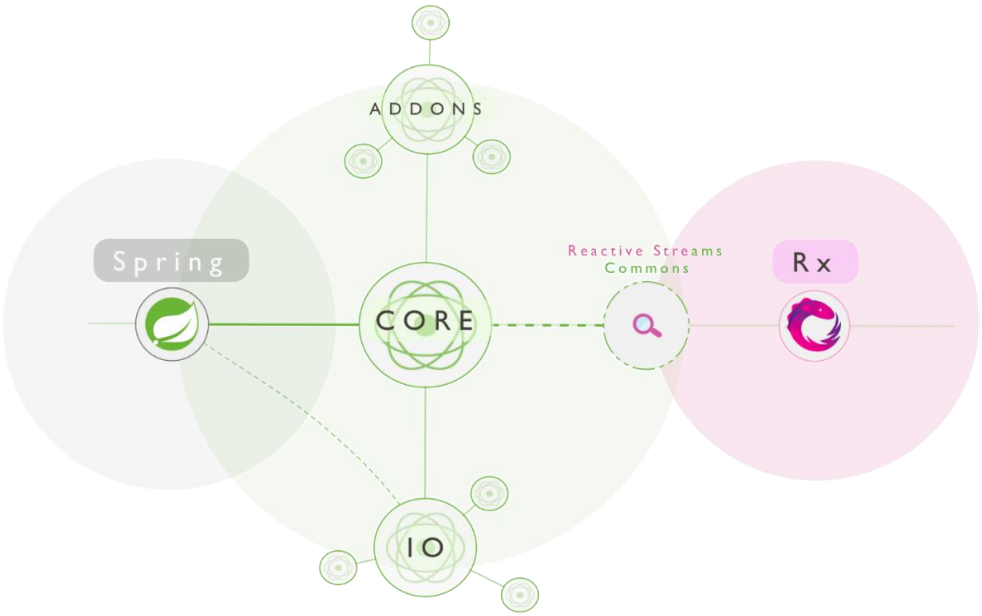


Рисунок 1.8 Модуль Spring WebFlux

До незручностей реактивного програмування слід зарахувати більш обмежений інструментарій роботи з реляційними БД. Наприклад, JPA–бібліотеки Hibernate та EclipseLink не підтримують реактивність. Неможливо в Java описувати складні зв'язки між об'єктами, як це робиться зазвичай тегами @Entity, @OneToMany, @ManyToMany, @JoinTable і т.д. Не вдасться програмно описати

обмеження цілісності таблиць. Хоча це надає великої гнучкості використовуваної БД. Для прив'язки класів до таблиць достатньо тегів @Table та @Column. Мови JPQL (HQL) не підходять для реактивних запитів до БД. Але при цьому доступний нативний SQL. Замість JDBC потрібно буде R2DBC.

Reactor – це повністю неблокуюча основа реактивного програмування для JVM з ефективним керуванням вимогами (у формі керування "протисуненням"). Він безпосередньо інтегрується з функціональними API–інтерфейсами Java 8, зокрема, Completable Future, Stream та Duration. Він пропонує складові API–інтерфейси асинхронної послідовності – Flux (для елементів [N]) та Mono (для елементів [1]) – та широко реалізує специфікацію Reactive Streams.

Reactor – це реалізація парадигми реактивного програмування, яку можна описати так. Реактивне програмування – це парадигма асинхронного програмування, пов'язана з потоками даних та поширенням змін. Це означає, що стає можливим легко виражати статичні (наприклад, масиви) або динамічні (наприклад, випромінювачі подій) потоки даних через мову програмування.

Всі приклади та реалізації Mono та Flux, що стосуються Reactor та WebFlux, які ми розглядаємо, будуть однаковими. Далі ми розберемося з усім у контексті WebFlux, побудованого на Reactor. Деталі проектів:

- Ядро Reactor – Реактивні основи для програм та середовищ, а також реактивні розширення, що базуються на API з типами Mono (1 елемент) та Flux (n елементів).
- Reactor Netty – пропонує неблокуючі та готові до backpressure TCP/HTTP/UDP клієнти та сервери на основі Netty фреймворку.
- Reactor Addons – Міст RxJava 2 Observable, Completable, Flowable, Single, Maybe, Scheduler, а також SWT Scheduler, Akka Scheduler і так далі.

Реактивні типи не призначені для обробки запитів або даних швидше. Їхня сила полягає в їх здатності одночасно обслуговувати більше запитів та більш ефективно обробляти операції із затримкою, такі як запит даних з віддаленого сервера.

Вони дозволяють забезпечити кращу якість обслуговування та передбачуване планування пропускної спроможності, спочатку маючи справу з часом та затримкою, не витрачаючи більше ресурсів. На відміну від традиційної обробки, яка блокує поточний потік під час очікування результату, Reactive API запитує лише той обсяг даних, який він здатний обробити та надає нові можливості, оскільки має справу з потоком даних, а не з окремими елементами (об'єктами).

Реактивні API, такі як Reactor, призначені для обробки синхронних, так і асинхронних операцій і дозволяють буферизувати, об'єднувати або застосовувати широкий спектр перетворень до ваших даних. Спочатку API Reactive були розроблені лише для роботи з потоками даних типу Flux. Але згодом було представлено і потік Mono. Mono є реактивним еквівалентом CompletableFuture. Flux та Mono реалізують Publisher інтерфейс зі специфікації Reactive Streams. Основним завданням Reactive Streams є обробка backpressure. Я не став перекладати це слово, щоб не припуститися помилок у розумінні. Backpressure – це механізм, який дозволяє одержувачеві запитувати, скільки даних він хоче отримати. Тобто, одержувач починає отримувати дані лише тоді, коли він готовий їх опрацювати.

Щоб реалізувати висконавантажену систему необхідно достеменно розібратись як це все працює всередині і досконало розуміти всі принципи які застосовуються в реактивному програмуванні. Без цього розуміння розробник не зможе наповну використовувати данний принцип і отримати хоч якийсь виграш від використання досить складного фреймворку. Необхідно описати основний принцип роботи основних компонентів Reactive Streams специфікації.

Publisher видає якісь дані (матеріали). Дані йдуть по ланцюжку з операторів (конвеєрної стрічки), обробляються, в кінці виходить готовий продукт, який передається в необхідний Consumer/Subscriber і використовується вже там.

Оператор – це якийсь Publisher, який, крім якоїсь своєї логіки, містить посилання на вихідний Publisher, до якого застосовується. Виклики операторів створюють ланцюжок із Publisher. Реактивне програмування виникло через

бажання писати асинхронний код, що не блокується, в читаному вигляді. Ні код, написаний на колбеках, ні код, написаний за допомогою CompletableFuture, не може бути настільки легким для читання, як цього можна домогтися за допомогою реактивності.

В основі підходу лежить ідея поділу компонентів на 2 типи: джерело подій (Publisher) та обробник подій (Subscriber). Subscriber підписується на події, які створює Publisher, потім якимось чином їх обробляє. По суті, це патерн Observer з надбудованими поверх можливостями та особливостями. Існує ще одне поняття – Observer. Він може передплатити подію об'єкта і виконувати будь–які дії з отриманим результатом. Один Subject може мати багато передплатників. Спілкування між Publisher та Subscriber відбувається через об'єкт Subscription. На рисунку 1.9 схематично зображена найчастіша взаємодія цих компонентів між собою при обробці данних.

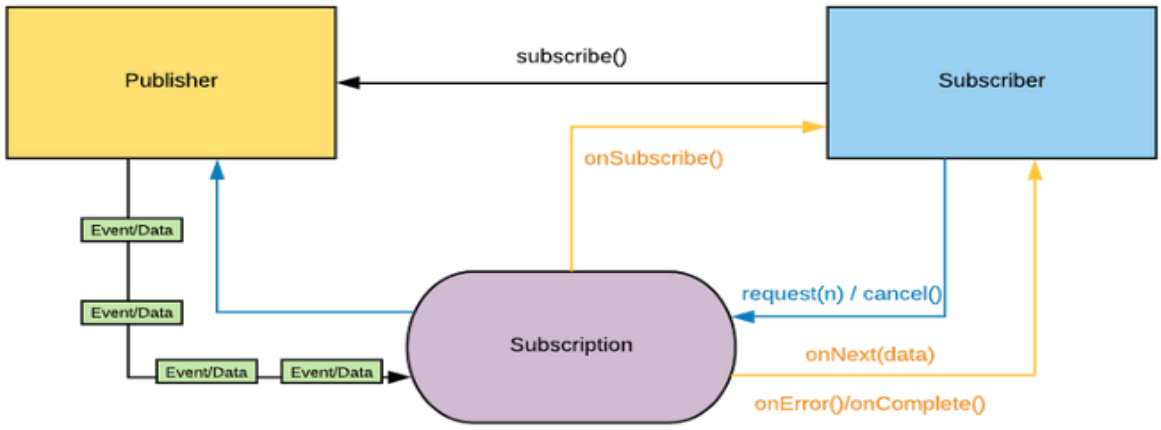


Рисунок 1.9 Схема взаємодії компонентів в Reactive Streams специфікації

Subcriber може регулювати швидкість доставки повідомлень від Publisher (т.к. backpressure), а також скасовувати передплату. Publisher'и можна об'єднувати у ланцюжки та комбінувати різними способами. Інтерфейси Publisher, Subscriber та Subscription знаходяться в пакеті org.reactivestreams, який за умовчанням був

доданий до Java 9. Вони задають специфікацію для реалізації реактивних потоків. Бібліотека Project Reactor є такою реалізацією. Можлівість використовувати реактивні потоки стала доступною в Spring 5. Spring 5 представив платформу WebFlux, яка є повністю асинхронним і неблокуючим реактивним веб–стеком, який дозволяє обробляти величезну кількість одночасних з'єднань. Модуль WebFlux є альтернативою Spring MVC і є реактивним підходом для написання веб–сервісів. На рис 1.10 зображена діаграма обох частин фреймвокру зі спільними та частинами які є різними.

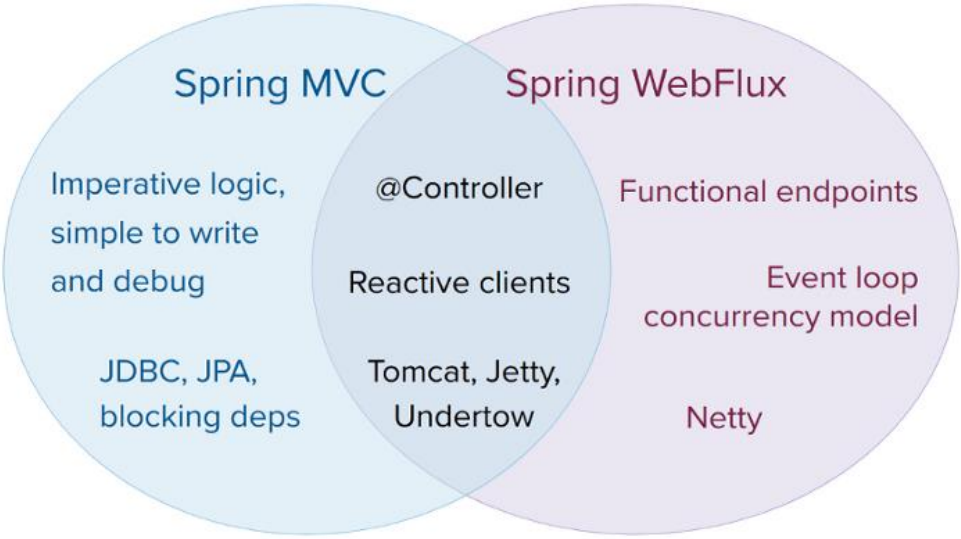


Рисунок 1.10 Схема модулів Spring MVC і Spring WebFlux

WebFlux позиціонує себе як мікрофреймворк. Цей новий мікрофреймворк підтримує анотовані контролери, функціональні кінцеві точки, WebClient (аналог RestTemplate у Spring Web MVC), WebSockets та багато іншого. В основі WebFlux лежить бібліотека Reactor. Вам потрібний Spring Boot версії 2+ для використання модуля Spring WebFlux. За замовчуванням WebFlux використовує Netty. Tomcat не підтримує реактивність.

Також в контексті реактивного програмування важливою частиною є технологія яка дозволяє зробити асинхронне програмування можливим на легким

на рівні мережі на взаємодії з навколишнім світом. Такою технологією є Netty. Netty – це платформа клієнт–сервер NIO, яка дозволяє швидко та легко розробляти мережеві додатки, такі як сервери протоколів і клієнти. Це значно спрощує та впорядковує мережеве програмування, таке як сервер сокетів TCP та UDP. Коротко внутрішня архітектура Netty представлена на рис. 1.11.

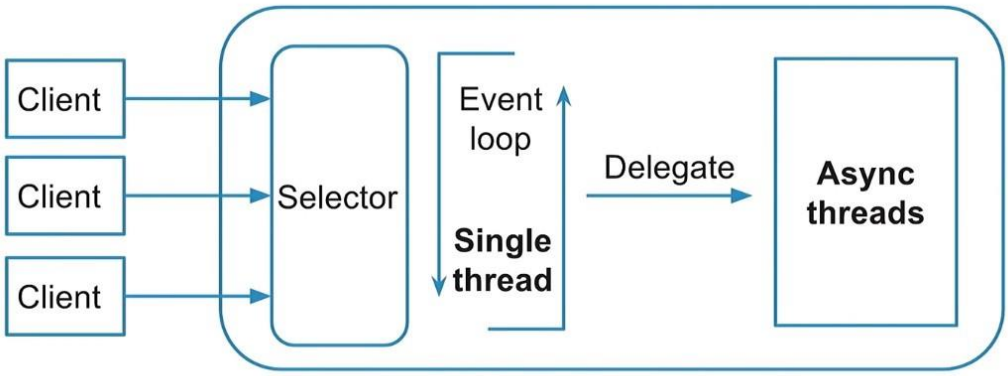


Рисунок 1.11 Архітектура Netty

На вході Netty має один потік, який працює в нескінченному циклі. Завдяки селектору та канальному механізму, він перенаправляє дані з вхідних запитів у вхідні буфери та делегує обробку цих запитів виділеного пулу потоків асинхронних потоків. І також у зворотному напрямку.

Висновки до розділу

У цьому розділі було розглянуто основні підходи архітектурні можливості щоб розробити високонавантажену систему. Були розглянуті основні концепції та складнощі звязані з розробкою таких систем. Відмічається певна схожість рішень які є розроблені різними компаніями для того щоб бути стійкими та витримувати навантаження в залежності від ситуації. Було вибрано технології для порівняння та проведення досліду.

2. ПРИНЦИП РОЗРОБКИ ВИСОКОНАВАНТАЖЕНИХ СИСТЕМ

2.1 Постановка завдання

У цьому розділі буде продемостровано порівняння MVC та Reactor підходу при розробці веб додатків за допомогою Spring Boot та Java. Для цього прикладу було написано кілька легких мікросервісів, які просто передають запити до деяких базових служб через HTTP і повертають їх назад. Основним обмеженням є те, що ми не контролюємо сервіси з якими ми інтегруємось. На рисунку 2.1 представлений основний ланцюг взаємодій:

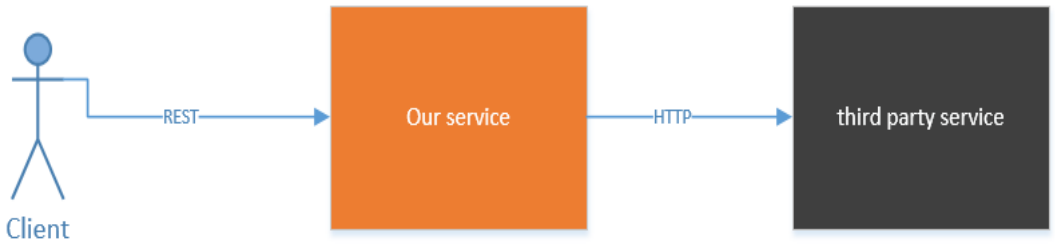


Рисунок 2.1 Основний ланцюг взаємодії

Тож, звичайно, необхідно почати з Spring Boot і написали прості RestControllers. Було зроблено POC проєкт, і результати були хорошими. Сторонній сервіс мав угоду про рівень обслуговування з часом відповіді служби, і ми використовували це значення для тестів продуктивності. Час відгуку стороннього сервісу був досить хорошим ~ близько 10–100 мс. Ми також вирішили використовувати ЦП як політику масштабування для наших мікросервісів, яка працювала в Docker як сервіс AWS ECS. Було налаштовано авто–масштабування в AWS і почали працювати.

Але не все пройшло гладко. Було виявлено досить неприємну закономірність. Ми часто перезапускали завдання AWS ECS через час очікування перевірки

працездатності. Крім того, було цікаво, що масштабування працює не так добре, і ми завжди мали мінімальну кількість виконуваних завдань. Крім того, ми побачили, що процесора і пам'яті не вистачало, але наш сервіс був занадто повільним, а іноді навіть мав помилку тайм–ауту.

Проблема як раз була в сторонньому сервісі який ми не контролюємо. Час відповіді сторонніх служб став 500–1000 мс. Але він ніколи не мав проблем із тайм–аутом і міг обробляти більше клієнтів, ніж ми. Така ситуація є досить банальною при розробці власних високо навантажених систем і є досить поширеною.

CPU процесора був низьким, пам'яті вистачало, але ми змогли обробити лише 200 запитів/сек.

Це була проблема з тим як працює підхід на основі Servlet API і існує проблема з тим що одному запиту видається цілий потік для його обробки. Пул потоків за замовчуванням становить 200, тому ми маємо 200 запитів на секунду для часу відповіді 1000 мс. Але нам потрібна була еластична служба: ми повинні обробляти стільки запитів, скільки може базова служба. І час відповіді має бути майже таким же, як і для базової служби. Існує декілька варіантів як вирішити цю проблему:

- Збільшити розмір пулу потоків;
- Використовувати DeferredResult або CompletableFuture з Servlet API (Non–Blocking);
- Spring WebFlux.

Варіант 1 – збільшити кількість потоків. Так, це хороший обхідний шлях, але тільки обхідний шлях. Оскільки не можна встановити це значення на кілька тисяч, тому що це Docker з дуже обмеженою пам'яттю. І кожен потік потребує пам'яті стека. Інша проблема полягає в тому, що якщо якийсь сторонній сервіс мав великий час відповіді, наприклад, 5 секунд, то все ще маємо ту саму проблему. Пропускна здатність дорівнює = розмір пулу потоків/час відповіді. Якщо буде 1000 потоків і затримка 5 секунд, то пропускна здатність становить 200 запитів/сек. ЦП знову низький, і сервіс має достатньо ресурсів для обробки.

Варіант 2 – Використовувати DeferredResult або CompletableFuture з Servlet API **(Non–Blocking)**. Як відомо, Servlet API версії 3.1 підтримує асинхронну обробку. Щоб він запрацював, потрібно просто повернути деяку обіцянку, і Servlet оброблятиме це асинхронно. Було порівняно DeferredResult з CompletableFuture і результат був таким же. Таким чином, візьмемо CompletableFuture.

Варіант 3 – Використання Spring Webflux (Project Reactor). Зараз це найпопулярніша тема. З документації Spring: “неблокуючий веб–стек для обробки паралельності з невеликою кількістю потоків і масштабування з меншою кількістю апаратних ресурсів”. Для подальшого удосконалення системи необхідно визначитись з основними критеріями які доцільно поліпшити щоб отримати результати на порядок вищі за отримані.

Для цілісності проведення експерименту використано однакові умови в усіх випадках. До цього відноситься версія Spring Boot, версія Java, вид AWS машини на якій будуть відбуватися запуск наших мікросервісів. Візуальний опис сценарію представлений на рисунку 2.2. Також обмежимо кількість бібліотек http клієнтів які ми будемо використовувати. Тож сформулюємо наступні обмеження:

- **Spring Boot**:2.5.6 (latest);
- **Java**: 11 OpenJDK;
- **Node**: t2.micro (Amazon Linux);
- **Http Clients:** Java 11 Http Client, Apache Http Client, Spring WebClient.

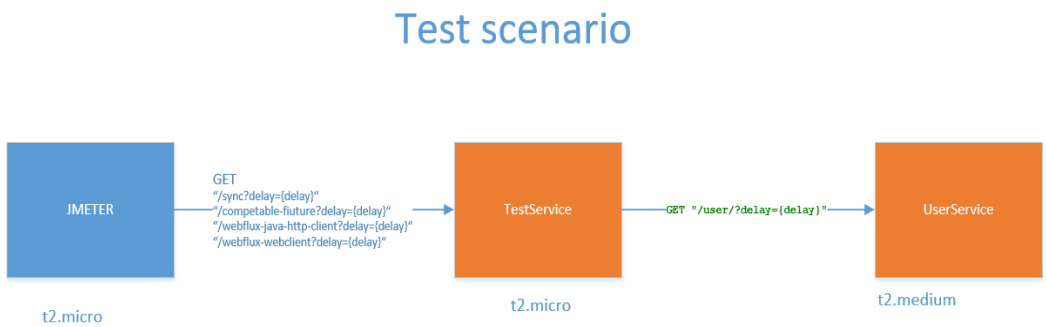


Рисунок 2.2 Тестовий сценарій

Test–Service (наш проксі–сервіс) надає кілька кінцевих точок GET для тестування. Усі кінцеві точки мають параметр затримки (у мс), який використовується для затримки обслуговування сторонніх розробників.

User–Service (сервіс третьої сторони. Або той який вище ми не можемо контролювати) надає єдину кінцеву точку GET «/user?delay={delay}». Параметр Delay(ms) використовується для емуляції затримки. Якщо ми надішлемо /user?delay=10, то час відповіді становитиме 10 мс + затримка мережі (мінімальна в AWS). Ця служба для користувачів є нашою сторонньою службою (сервісом користувача), яка дуже швидка і може обробляти понад 4000 запитів/сек.

Для тесту продуктивності ми будемо використовувати Jmeter. Перевіримо наш сервіс на 100, 200, 400, 800 одночасних запитів із затримкою 10 100 500 мс. Всього 12 тестів для кожної реалізації. У цьому тесті веб додадки використовують систему збірки Maven. Це ніяк не впливає на дослідження. Для подальшого удосконалення системи необхідно визначитись з основними критеріями які доцільно поліпшити щоб отримати результати на порядок вищі за отримані.

2.2 Критеріїї для розробки високо навантажених систем

2.1.1 Надійність системи

Кожен інтуїтивно уявляє, що означає бути надійним чи ненадійним. Від програмного забезпечення зазвичай очікується наступне:

- програма виконує очікувану користувачем функцію;
- воно здатне витримати помилкові дії користувача або застосування програмного забезпечення несподіваним чином;
- його продуктивність є достатньо високою для поточного сценарію використання, при передбачуваному навантаженні та обсязі даних;
- система запобігає будь–якому несанкціонованому доступу та неправильній експлуатації.

Якщо вважати, що все зазначене означає «працювати нормально», то термін «надійність» матиме значення, грубо кажучи, «продовжувати працювати нормально навіть у разі проблем».

Можливі проблеми називаються збоями, а системи, створені для них, називаються стійкими до збоїв. Цей термін здатний ввести в деяку оману: він наводить на думку, що можна зробити систему стійкою до всіх можливих видів збоїв. Однак на практиці це неможливо. Збій (fault) та відмова (failure) – різні речі. Збій зазвичай визначається як відхилення одного з компонентів системи від робочих характеристик, тоді як відмова – це ситуація, коли вся система загалом припиняє надання необхідного сервісу користувачеві. Знизити ймовірність збоїв до нуля неможливо, отже, зазвичай краще проектувати механізми стійкості до збоїв, які б запобігали переходу збоїв у відмови.

Парадоксально, але в подібних стійких до збоїв системах має сенс підвищити частоту збоїв за допомогою їхньої умисної генерації – наприклад, шляхом переривання роботи окремих, вибраних випадковим чином процесів без попередження. Багато критичних помилок фактично відбуваються через недостатню обробку помилок умисне породження збоїв гарантує постійне тестування механізмів забезпечення стійкості до них, що підвищує впевненість у належній обробці збоїв за їхньої «природної» появи. Хоча зазвичай вважається, що стійкість системи до збоїв важливіша за їх запобігання, існують випадки, коли попередження краще лікування (наприклад, коли "лікування" не існує). Це справедливо у разі безпеки: якщо атакуючий скомпрометував систему і отримав доступ до конфіденційних даних, то нічого вдіяти вже не можна.

2.1.2 Апаратні збої

Коли йдеться про причини відмови систем, насамперед на думку приходять апаратні збої. Фатальні збої вінчестерів, поява дефектів ОЗУ, відключення електроживлення, відключення кимось мережевого кабелю. Будь–хто, хто мав

справу з великими центрами обробки та зберігання даних, знає, що подібне відбувається постійно за наявності великої кількості машин.

Вважається, що середній час напрацювання на відмову (mean time to failure, MTTF) вінчестерів становить від 10 до 50 років. Таким чином, у кластері зберігання з 10 тисячами вінчестерів слід очікувати в середньому однієї відмови жорсткого диска на день.

Перша природна реакція на цю інформацію – підвищити надмірність окремих компонентів апаратного забезпечення з метою зменшення частоти відмов системи. Можна створити RAID–масиви з дисків, забезпечити дублювання електроживлення серверів та наявність у них CPU з можливістю гарячої заміни, а також запастися батареями та дизельними генераторами як резервні джерела електроживлення ЦОДів. При відмові одного компонента його місце під час заміни займає резервний компонент. Такий підхід не запобігає повністю відмовим, що виникають через проблеми з обладнанням, але цілком прийнятним і часто здатним підтримувати безперебійну роботу машин протягом багатьох років.

Донедавна надмірність компонентів апаратного забезпечення була достатньою для більшості додатків, роблячи критичну відмову окремої машини явищем цілком рідкісним. За наявності можливості досить швидкого відновлення з резервної копії на новій машині час простою у разі відмови не є катастрофічним для більшості додатків. Отже, багатомашинна надмірність була потрібна лише невеликій частині додатків, для яких була критично важлива висока доступність.

Однак у міру зростання обсягів даних та обчислювальних запитів додатків все більше програм почали використовувати більшу кількість машин, що призвело до пропорційного зростання частоти відмов обладнання. Більше того, на багатьох хмарних платформах, таких як Amazon Web Services (AWS), екземпляри віртуальних машин досить часто стають недоступними без попередження, оскільки платформи віддають перевагу гнучкості та здатності швидко адаптуватися перед надійністю однієї машини.

Тому відбувається зсув у бік систем, здатних перенести втрату цілих машин завдяки застосуванню методів стійкості до збоїв замість надмірності апаратного забезпечення або додатково до неї. Такі системи мають і експлуатаційні переваги: система з одним сервером вимагає планового простою за необхідності перезавантаження машини (наприклад, для встановлення виправлень безпеки), тоді як стійка до апаратних збоїв система допускає встановлення виправлень по вузлу за раз, без вимушеної бездіяльності всієї системи.

Зазвичай вважають так: апаратні збої мають випадковий характер і незалежні один від одного: відмова диска одного комп'ютера не означає, що диск іншого незабаром теж почне збоїти. Звичайно, можливі слабкі кореляції (наприклад, через загальну причину на зразок температури в серверній стійці), але в інших випадках одночасна відмова великої кількості апаратних компонентів є малоймовірною.

2.1.3 Програмні збої

Інший клас збоїв систематична помилка у системі. Подібні збої складніше запобігти, і через їх кореляцію між вузлами вони зазвичай викликають. Набагато більше системних відмов, ніж некорелювані апаратні збої [5]. Розглянемо приклади.

- Програмна помилка, що призводить до фатального збою екземпляра сервера програми за конкретних «поганих» вхідних даних. Наприклад, візьмемо секунду координації 30 червня 2012 року, що викликала одночасне зависання безлічі програм через помилку в ядрі операційної системи Linux.
- Виходить з–під контролю процес, що повністю вичерпав якийсь загальний ресурс: час CPU, оперативну пам'ять, простір на диску або смугу пропускання мережі.
- Сервіс, від якого залежить робота системи, уповільнюється, перестає відповідати на запити або починає повертати зіпсовані відповіді.
- Каскадні збої, при яких крихітний збій в одному компоненті викликає збій в іншому компоненті, а той, своєю чергою, викликає подальші збої.

Помилки, що спричиняють подібні програмні збої, часто довго залишаються неактивними, аж до моменту спрацьовування під впливом незвичайних обставин. При цьому виявляється, що додаток робить якесь припущення щодо свого оточення і хоча зазвичай таке припущення справедливе, зрештою воно стає невірним з якоїсь причини.

Швидкого вирішення проблеми систематичних помилок у програмному забезпеченні не існує. Може виявитися корисним безліч дрібниць, таких як: ретельне обмірковування припущень та взаємодій усередині системи; всебічне тестування; ізоляція процесів; надання процесам можливості перезапуску після фатального збою; оцінка, моніторинг та аналіз поведінки системи під час промислової експлуатації. Якщо система повинна забезпечувати виконання будь–якої умови (наприклад, у черзі повідомлень кількість вхідних повідомлень має бути дорівнює кількості вихідних), то можна організувати постійну самоперевірку під час роботи та видачу попередження у разі виявлення розбіжності.

2.1.4 Надійність

Надійність необхідна не тільки в програмному забезпеченні керування атомними електростанціями та повітряним рухом, надійна робота також очікується від поширених додатків. Помилки в комерційних додатках призводять до втрат продуктивності (і правових ризиків, якщо цифри у звітах неточні), а простої сайтів електронної комерції можуть призвести до колосальних втрат у вигляді втрат прибутку та шкоди репутації.

Творці навіть «некритичних» програм несуть відповідальність перед своїми користувачами. Візьмемо, наприклад, батьків, які зберігають усі фотографії та відео своїх дітей у вашому додатку для фотографій. Як вони почуватимуться якщо база даних несподівано пошкоджена? Чи зможуть вони відновити дані з резервної копії?

Бувають ситуації, коли доводиться пожертвувати надійністю, щоб зменшити витрати на розробку (наприклад, при створенні прототипу продукту для нового

ринку) або експлуатаційні витрати (наприклад, для послуги з дуже низьким прибутком) – але вам потрібно дуже обережно «зрізати кути».

2.1.5 Масштабування

Навіть якщо на сьогоднішній момент система працює надійно, немає гарантій, що вона так само працюватиме в майбутньому. Одна з частих причин зниження ефективності це зростання навантаження: наприклад, система зросла з 10 тис. працюючих одночасно користувачів до 100 тис. або з 1 до 10 млн. Це може бути й обробка значно більших обсягів даних, ніж раніше .

Масштабованість (scalability) – термін, який використовуватиметься для опису здатності системи справлятися зі збільшеним навантаженням. Зазначу, однак, що це не одномірний ярлик, який можна «навісити» на систему: фрази

"X – масштабована" або "Y – немасштабована". Швидше за все, обговорення масштабування означає розгляд наступних питань: «Якими будуть наші варіанти вирішення проблеми, якщо система зросте певним чином?» і «Як ми можемо розширити обчислювальні ресурси для обліку додаткового навантаження?».

2.1.6 Опис продуктивності систем

Після опису навантаження на систему можна з'ясувати, що станеться під час її зростання. Слід звернути увагу до два аспекти.

- Як зміниться продуктивність системи, якщо збільшити параметр навантаження за незмінних ресурсів системи (CPU, оперативна пам'ять, пропускна спроможність мережі тощо)?
- Наскільки потрібно збільшити ресурси зі збільшенням параметра

У системах пакетної обробки даних, таких як Hadoop, зазвичай хвилює пропускна здатність – кількість записів, які можливо обробити в секунду, або загальний час, необхідний виконання завдання на наборі даних певного розміру. В онлайн–системах важливіший, як правило, час відповіді сервісу, тобто час між відправленням запиту клієнтом та отриманням відповіді.

Терміни "час очікування" (latency) і "час відгуку" (response time) часто використовуються як синоніми, хоча це не те саме. Час відгуку це те, що бачить клієнт: окрім фактичного часу обробки запиту (час обслуговування, service time), він включає затримки під час передачі інформації через мережу та затримки повідомлень у черзі. Час очікування – тривалість очікування запитом обробки, тобто час, протягом якого очікується обслуговування [11].

Навіть якщо повторювати один раз той самий запит, час відгуку буде дещо відрізнятися при кожній спробі. На практиці в системі, що обробляє безліч різноманітних запитів, час відгуку здатний істотно відрізнятися. Отже, необхідно розглядати час відгуку не як одне число, бо як розподіл значень, характеристики якого можна визначити.

На рис. 2.3 кожен сірий стовпець відображає запит до сервісу, а його висота показує тривалість виконання цього запиту. Більшість запитів виконуються досить швидко, але трапляються й окремі аномальні запити із значно більшою тривалістю. Можливо повільні запити є більш витратними, наприклад, оскільки пов'язані з обробкою великих обсягів даних. Але навіть за сценарію, коли можна було б думати, що обробка всіх запитів займатиме однаковий час, насправді тривалість відрізняється. Додатковий час очікування може знадобитися через перемикання тексту на фоновий процес, втрати мережного пакета та повторну передачу по протоколу TCP, паузу на складання сміття, збій сторінки, що вимагає читання з диска, механічні вібрації в серверній стійці [12] та з багатьох інших причин.

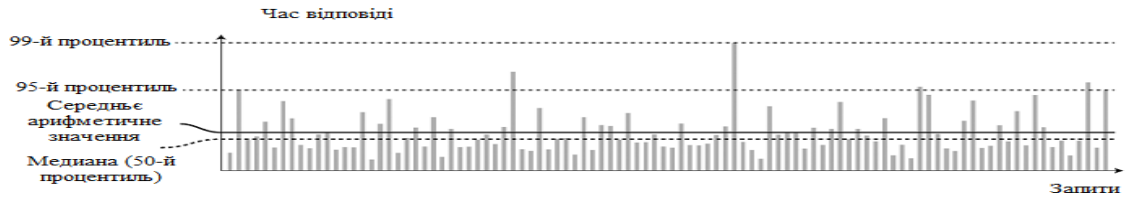


Рисунок 2.3 Приклад ілюстрації середнього значення, процентилю часу відповіді для вибірки із 100 запитів до сервісу

Часто дивляться саме на середній час відгуку. (Строго кажучи, термін «середнє» не має на увазі значення, обчисленого за якоюсь конкретною формулою, але на практиці під ним зазвичай розуміється арифметичне середнє: при n значеннях скласти їх усі і поділити на n.) Проте середнє значення далеко не найкраща метрика для випадків, коли потрібно знати «типовий» час відгуку, оскільки він нічого не говорить про те, яка кількість користувачів фактично мала таку затримку.

Зазвичай зручніше застосовувати відсоток. Якщо відсортувати список часів відгуку за зростанням, то медіана – середня точка: наприклад, медіанний час відгуку дорівнює 200 мс, означає, що відповіді на половину запитів повертаються менш ніж через 200 мс, а половина запитів займає більш тривалий час.

Це робить медіану чудовою метрикою, коли потрібно дізнатися, скільки користувачам зазвичай доводиться чекати: Половина запитів користувача обслуговується за час відгуку менше медіанного, а ті, що залишилися, обслуговуються більш тривалий час. Медіана також називається 50–м відсотком, який іноді позначають p50. Зверніть увагу: медіана належить до окремого запиту; якщо користувач виконує кілька запитів (за час сеансу або тому, що в одну сторінку включено кілька запитів), ймовірність виконання хоча б одного з них повільніше за медіану значно перевищує 50%.

Щоб з'ясувати, наскільки погані аномальні значення, можна звернути увагу на більш високі відсотки: часто застосовуються 95–й, 99–й і 99.9–й (p95, p99 і p999). Це порогові значення часу відгуку, для яких 95 %, 99 % або 99,9 % запитів виконуються швидше за відповідне порогове значення часу. Наприклад, те, що час відгуку для 95–го процентиля дорівнює 1,5 с, означає наступне: 95 зі 100 запитів займають менше 1,5 с, а 5 зі 100 займають 1,5 с або довше.

Верхні відсотки часу відгуку, відомі також під назвою «хвостових» часів очікування, важливі тому, що безпосередньо впливають на досвід взаємодії користувача з сервісом. Наприклад, Amazon визначає вимоги до часу відгуку для внутрішніх сервісів у термінах 99.9х процентилей, хоча це стосується лише 1 з

1000 запитів. Справа в тому, що клієнти з найповільнішими запитами найчастіше саме ті, у кого найбільше даних в облікових записах, оскільки вони зробили багато покупок, тобто є найціннішими клієнтами [13]. Важливо, щоб ці клієнти були задоволені; необхідно забезпечити швидку роботу сайту саме для них: компанія Amazon виявила, що зростання часу відгуку на 100 мс знижує продажі на 1% [14], а за іншими повідомленнями, уповільнення на 1с знижує задоволеність користувачів на 16% [15, 16].

З іншого боку, оптимізація за 99.99–м процентилем (найповільнішим з 10 000 запитів) вважається надто дорогою і не приносить достатньо вигоди з точки зору цілей Amazon. Знижувати час відгуку на дуже високих відсотках – непросте завдання, оскільки на них можуть впливати з незалежних від вас причин різні випадкові події, а вигоди від цього мінімальні.

Наприклад, проценти часто використовуються у вимогах до рівня надання сервісу (service level objectives, SLO) та угод про рівень надання сервісу (service level agreements, SLA) – контракти, що описують очікувані продуктивність та доступність сервісу. У SLA, наприклад, може бути зазначено: сервіс розглядається як функціонуючий нормально, якщо його медіанний час відгуку менше 200 мс, а 99–й процентиль менше 1 с (коли час відгуку більше, це рівносильно непрацюючому сервісу), причому у вимогах може бути зазначено що сервіс повинен працювати нормально не менше 99,9% часу. Завдяки цим метрикам клієнтські програми знають, чого очікувати від сервісу, та забезпечують користувачам можливість вимагати відшкодування у разі недотримання SLA.

За значну частину часу відгуку на верхніх відсотках часто несуть відповідальність затримки повідомлень у черзі. Оскільки сервер може обробляти паралельно лише невелику кількість завдань (обмежену, наприклад, кількістю ядер процесора), навіть невеликої кількості повільних запитів достатньо для затримки наступних запитів – явище, яке іноді називається блокуванням голови черги. Навіть якщо наступні запити обробляються сервером швидко, клієнтська програма все одно спостерігатиме низький загальний час відгуку через час очікування

завершення попереднього запиту. Беручи до уваги це явище, важливо вимірювати час відгуку за клієнта. При штучній генерації навантаження з метою тестування масштабованості системи клієнт, що генерує навантаження, повинен надсилати запити незалежно від часу відгуку. Якщо клієнт чекатиме завершення попереднього запиту перед відправкою наступного, це буде рівносильно штучному скорочення черг при тестуванні проти реальністю, що спотворить отримані результати вимірів [17].

Верхні процентили набувають особливого значення в прикладних сервісах, що викликаються неодноразово при обслуговуванні одного запиту кінцевого користувача. Навіть при виконанні дзвінків паралельно запиту кінцевого користувача все одно доводиться чекати завершення найповільнішого з паралельних дзвінків. Достатньо одного повільного виклику, щоб уповільнити весь запит кінцевого користувача, як показано на рис. 2.4.

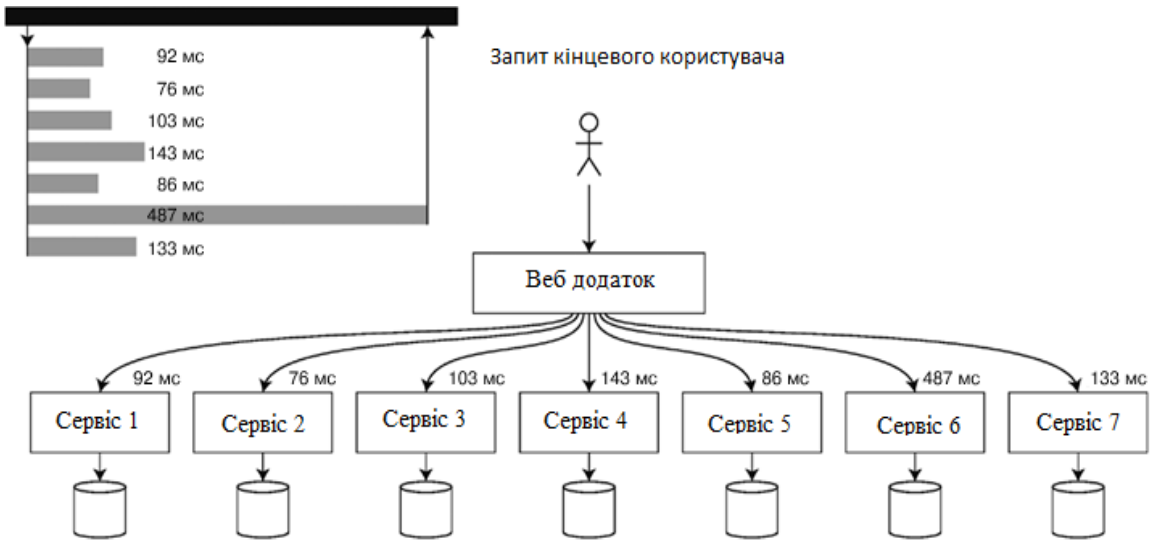


Рисунок 2.4 Приклад системи в якій для опрацювання запиту необхідно здійснити запити в декілька інших

Навіть якщо лише невеликий відсоток прикладних викликів повільні, шанси потрапити на повільний виклик зростають, якщо запит кінцевого користувача потребує їх багато, так що більший відсоток запитів кінцевих користувачів виявляється повільним (це явище відоме під назвою «посилення «хвостового» часу очікування» [18]).

Якщо потрібно додати в моніторингові інструментальні панелі ваших сервісів відсоток часу відгуку, необхідно ефективно організувати їх регулярне обчислення. Наприклад, можна використовувати ковзне вікно часу відгуку запитів за останні 10 хвилин з обчисленням кожну хвилину медіани та різних відсотків за значеннями з даного вікна з побудовою графіка цих метрик.

Як «наївну» реалізацію можна запропонувати список часів відгуку для всіх запитів у тимчасовому вікні з сортуванням цього списку щохвилини. Якщо така операція є недостатньо ефективною, існують алгоритми наближеного обчислення відсотків при мінімальних витратах процесорного часу та оперативної пам'яті, наприклад forward decay [19], t–digest [20] та HdrHistogram [21].Необхідно враховувати, що усереднення відсотків з метою зниження часового дозволу або об'єднання даних з кількох машин математично безглуздо – правильніше агрегувати дані про час відгуку шляхом складання гістограм [22].

Висновки до розділу

У цьому розділі було розглянуто основні принципи та критеції які важливі при розробці високо–навантажених систем. Були розглянуті та описані основні вимоги до таких систем. Було вибрано технології для порівняння та проведення досліду. Було виділено основні моменти які необхідно врахувати та виділено основні критеції оцінки системи.

3. ЕКСПЕРИМЕНТАЛЬНА РОБОТА ПО ДОСЛІДЖЕННЮ ПОВЕДІНКИ ВИСОКОНАВАНТАЖЕНИХ СИСТЕМ

3.1 Розробка та опис початкової системи за допомогою Spring MVC

Для розробки системи написаної за допомогою Spring MVC необхідно розробити наступни сервіси:

- Test Service;
- User–Service.

Test–Service буде надавати кілька кінцевих точок GET для тестування. Усі кінцеві точки повинні мати параметр затримки (у мс), який буде використовуватись для затримки обслуговування та проведення різних сценаріїв тесів навантаження.

User–Service буде надавати єдину кінцеву точку GET запиту «/user?delay={delay}». Також необхідно реалізувати логіку затримки через параметр Delay(ms). Він буде використаний у тестах навантаження для емуляції затримки. Якщо ми надішлемо /user?delay=10, то час відповіді становитиме 10 мс + затримка мережі (мінімальна в AWS). Ця служба для користувачів є нашою сторонньою службою (сервісом користувача), яка дуже швидка і може обробляти понад 4000 запитів/сек. Сервіс буде описаний у наступному розділі бо код сервісу спільний для обох сервісів і буде написаний за допомогою Spring Webflux та Reactor щоб витримувати необхідне навантаження при тестуванні.

Тепер необхідно описати ключовий функціонал мікросервісу – Test–Service. На лістингу 3.1 продемонстровано Rest контроллер який буде приймати запити та перенапвляти їх на інший сервіс який буде моделювати затримку.

Лістинг 3.1 Spring MVC Rest контроллер

```
@RestController
public class Controller {
    private final HttpClient httpClient;
    private final String userServiceHost;
```

```
    @Autowired
    public Controller(
        @Value("${user.service.host}") String userServiceHost) {
        this.userServiceHost = userServiceHost;
        this.httpClient = HttpClient.newBuilder()
.executor(Executors.newSingleThreadExecutor())
.build();
    }
    @GetMapping(value = "/sync")
    public String getUserSync(@RequestParam long delay) {
        return sendRequestWithApacheHttpClient(delay).thenApply(x -> "sync: " +
x).join();
    }


}
```

У цьому конттроллері присутня адреса на яку можливо відправити запит. Він приймає всі необхідні параметри та використовує блокуючий API.

3.2 Розробка високо навантаженої системи за допомогою Spring WebFlux (Reactor)

У цьому розділі описано ключові моменти розробки Test Service та User–Service написаний за допомогою неблокуючого підходу.

Код User–Service буде спільним для обох сервісів і буде написаний за допомогою SpringWebflux щоб витримувати навантаження при тестуванні. Цей сервіс має контроллер який вміє приймати запити та відповідати на них з затримкою яка нам буде потрібна в залежності від ситуації. Так сервіс дуже легкий що дає змогу навантажувати його без страху що він зможе нам зіпсувати результати тесту. На лістинку 3.2 зображено код основного контроллеру.

Лістинг 3.2 Spring WebFlux Rest контроллер

```
@RestController
public class UserController {
    @GetMapping("/user")
    public Mono<String> getUserWithDelay(@RequestParam long delay) {
        return Mono.just("USER").delayElement(Duration.ofMillis(delay));
    }
}
```

Тепер потрібно зробити деякі зміни в Test–Service щоб він відповідав критеріям реактивного сервісу. Спочатку необхідно додати залежність у pom.xml проекту на модуль spring–boot–webflux–starter. Цей модуль включає всі необхідні залежності щоб зробити сервіс реактивним і включає всі інші необхідні залежності:

- spring–boot і spring–boot–starter для базового налаштування програми Spring Boot
- фреймворк spring–webflux
- reactor–core, який нам потрібен для реактивних потоків, а також reactor–netty

Клас Spring RestTemplate за своєю природою блокує. Отже, ми не хочемо використовувати його в реактивній програмі. Для реактивних програм Spring пропонує клас WebClient, який не блокує. Але мі будемо використувати декілька варіантів неблокуючих Http клієнтів та порівняємо результати і зробимо висновкий який наразі є найкращим варінтом та справляється краще із навантаженням і відповідає критеріям для того щоб бути використаним при розробці високо навантаженої системи.

Тепер необхідно описати реактивний контроллер який буде повертати данні від User Service. Цей простий реактивний клас завжди повертає тіло JSON. Він може повертати багато інших речей, включаючи потік елементів із бази даних, потік елементів, згенерованих за допомогою обчислень, тощо. Важливим моментом є те що ми беспосереднь повертаємо об’єкти обготки які вміют працювати с асинхронним середовищем. Прикладом об’єкт Mono, який містить тіло ServerResponse. На лістингу 3.3 представлено код реативного контроллеру User–Service.

Лістинг 3.3 Реактивний контроллер сервісу User–Service

```
@RestController
public class Controller {
    private final WebClient webClient;
    private final HttpClient httpClient;
    private final String userServiceHost;
    private final CloseableHttpAsyncClient apacheClient;

    @Autowired
```

```
    public Controller(@Value("${user.service.host}") String userServiceHost) {
        this.userServiceHost = userServiceHost;
        this.webClient = WebClient.builder().baseUrl(userServiceHost).build();
        this.httpClient =
HttpClient.newBuilder().executor(Executors.newSingleThreadExecutor()).build();
        this.apacheClient =
HttpAsyncClients.custom().setMaxConnPerRoute(2000).setMaxConnTotal(2000).build();
        this.apacheClient.start();
    }


    @GetMapping(value = "/completable-future-java-client")
    public CompletableFuture<String> getUserUsingWithCFAndJavaClient(@RequestParam
long delay) {
        return sendRequestWithJavaHttpClient(delay).thenApply(x -> "completable-
future-java-client: " + x);
    }

    @GetMapping(value = "/completable-future-apache-client")
    public CompletableFuture<String>
getUserUsingWithCFAndApacheClient(@RequestParam long delay) {
        return sendRequestWithApacheHttpClient(delay).thenApply(x -> "completable-
future-apache-client: " + x);
    }

    @GetMapping(value = "/webflux-java-http-client")
    public Mono<String> getUserUsingWebfluxJavaHttpClient(@RequestParam long
delay) {
        CompletableFuture<String> stringCompletableFuture =
sendRequestWithJavaHttpClient(delay).thenApply(x -> "webflux-java-http-client: " +
x);
        return Mono.fromFuture(stringCompletableFuture);
    }

    @GetMapping(value = "/webflux-webclient")
    public Mono<String> getUserUsingWebfluxWebclient(@RequestParam long delay) {
        return webClient.get().uri("/user/?delay={delay}",
delay).retrieve().bodyToMono(String.class).map(x -> "webflux-webclient: " + x);
    }

    @GetMapping(value = "/webflux-apache-client")
    public Mono<String> apache(@RequestParam long delay) {
        return
Mono.fromCompletionStage(sendRequestWithApacheHttpClient(delay).thenApply(x ->
"webflux-apache-client: " + x));
    }
}
```

Важливим моментом є те, що ми маємо декілька адрес які використовують різні бібліотеки щоб виконувати Http запити. Необхідно звернути увагу що кодова база у них різна і результати можуть бути різними але це нам і потрібно щоб обрати найкращий варіант. Одним із них є класс WebClient використовує реактивні функції у вигляді Mono для зберігання вмісту повідомлення (повертається методом

getMessage). Вони використовують API функції, а не імперативний, для ланцюга реактивних операторів.

Може знадобитися час, щоб звикнути до Reactive API, але WebClient має цікаві функції і також може використовуватися в традиційних програмах Spring MVC.

Щоб запустити серіси необхідно створити головний класс і запустити программу через метод main. Код цих методів однаковий для всіх сервісів тому доцільно описати лише один. На лістингу 3.4 представлено головний класс сервісу Test Service.

Лістинг 3.4 Main класс сервісу User–Service

```
@SpringBootApplication
public class Application {

  public static void main(String[] args) {
    ConfigurableApplicationContext context =
SpringApplication.run(Application.class, args);
  }
}
```

@SpringBootApplication – це головна аннотація Spring Boot. Вона дуже зручна анотація, яка додає все з наступного:

- @Configuration: позначає клас як джерело визначень bean для контексту програми.
- @EnableAutoConfiguration: повідомляє Spring Boot про початок додавання bean–компонентів на основі налаштувань шляху до класів, інших компонентів і різних налаштувань властивостей. Наприклад, якщо spring–webmvc знаходиться на шляху до класу, ця анотація позначає програму як веб–програму та активує ключові дії, такі як налаштування DispatcherServlet.
- @ComponentScan: повідомляє Spring шукати інші компоненти, конфігурації та служби в пакеті hello, дозволяючи йому знайти контролери.

Метод main() використовує метод SpringApplication.run() Spring Boot для запуску програми. Ви помітили, що не було жодного рядка XML? Також немає файлу web.xml. Це веб–додаток на 100% чистий Java, і вам не довелося мати справу з налаштуванням будь–якої сантехніки чи інфраструктури.

Ви можете запустити програму з командного рядка за допомогою Gradle або Maven. Ви також можете створити один виконуваний файл JAR, який містить усі необхідні залежності, класи та ресурси, і запустити його. Створення виконуваного файлу jar полегшує відправку, версію та розгортання служби як програми протягом усього життєвого циклу розробки, у різних середовищах тощо.

Тепер можливо запустити наші сервіси. Принцип запуску будет однаковий для обох сервісів через main метод. Щоб запустити Webflux (Project Reactor) версію сервісу необхідно використати команду: mvn clean install –P web–flux .

Для Servlet(Tomcat) версії додатку треба використовувати профіль maven “servlet” і команду mvn clean install –P servlet.

3.3 Аналіз результатів досліду

У тестах навантаження важливими є метрики та постійний моніторинг мікро-сервісів. Існує багато різних метрик на основі яких ми зможемо зробити висновки про те чи працює в даний момент часу сервіс стабільно чи ні, чи сервіс задовольняє вимогам про пропускній здатності і здатність обробляти ту кількість запитів яка необхідна для компанії або чи стаються якісь події які заважають йому нормально функціонувати. Перша метрика це Throughput results(msg/sec) тобто це здатність обробити певну кількість повідомлень в секунду. Що таке пропускна здатність? Пропускна здатність – це кількість продукту чи послуги, яку компанія може виготовити та надати клієнту протягом певного періоду часу. Цей термін часто використовується в контексті швидкості виробництва компанії або швидкості, з якою щось обробляється. Підприємства з високим рівнем пропускної здатності можуть відібрати частку ринку у своїх аналогів з нижчою пропускною здатністю,

оскільки висока пропускна здатність зазвичай свідчить про те, що компанія може виробляти продукт або послугу ефективніше, ніж її конкуренти.

Розуміння пропускної здатності або ідея пропускної здатності, також відома як швидкість потоку, є частиною теорії обмежень в управлінні бізнесом. Керівна ідеологія цієї теорії полягає в тому, що ланцюг настільки сильний, наскільки міцна його найслабша ланка. Мета бізнес – менеджерів – знайти шляхи мінімізації впливу найслабших ланок на продуктивність компанії та максимізації пропускної здатності для кінцевих користувачів продукту. Після того, як пропускна спроможність буде максимізована шляхом усунення неефективності, дозволяючи вхідним і вихідним потокам працювати найідеальнішим чином, компанія може досягти максимізації доходу. У пропускну здатність зараховуються лише продукти, які фактично продані. Рівень виробничих потужностей компанії тісно пов'язаний з пропускною здатністю, і керівництво може зробити кілька типів припущень щодо потужності. Якщо компанія припускає, що виробництво працюватиме безперервно без будь–яких перебоїв, керівництво використовує теоретичні потужності, але цей рівень потужності недосяжний. Жоден виробничий процес не може вічно виробляти максимальну продуктивність, тому що машини потрібно ремонтувати та обслуговувати, а працівники беруть дні відпусток. Для підприємств більш реально використовувати практичні потужності, які враховують ремонт машин, час очікування та відпустки. Пропускна здатність компанії також залежить від того, наскільки добре компанія керує своїм ланцюгом поставок, тобто взаємодією між компанією та її постачальниками. Якщо з будь–якої причини поставки недоступні як вихідні ресурси для виробництва, перебої негативно впливають на пропускну здатність. У багатьох випадках два продукти можуть почати виробництво з використанням одного процесу, що означає, що спільні витрати розподіляються між кожним продуктом. Однак, коли виробництво досягає точки відокремлення, продукція виробляється за допомогою окремих процесів. Така ситуація ускладнює підтримку високого рівня пропускної здатності. Результати про даній метриці наведені у «Додаток А».

Наступною метрикою є CPU Utilization. Використання ЦП відноситься до використання комп'ютером ресурсів обробки або обсягу роботи, яку виконує ЦП. Фактичне використання ЦП залежить від кількості та типу керованих обчислювальних завдань. Деякі завдання вимагають багато часу ЦП, тоді як інші потребують менше через вимоги до ресурсів, не пов’язаних із ЦП. Використання ЦП може використовуватися для оцінки продуктивності системи. Наприклад, велике навантаження лише з кількома запущеними програмами може свідчити про недостатню підтримку живлення ЦП або запущених програм, прихованих системним монітором, – це високий показник вірусів та/або шкідливих програм. Завантаження ЦП не слід плутати із навантаженням ЦП. Результати про даній метриці наведені у «Додаток Б».

Тепер розглянемо стани потоків процессору при навантаженні в обох випадках. Tomcat без завантаження, який використовується у стартовому веб–спеціальному Spring, створює 10 потоків http exec. Під навантаженням він зможе масштабуватися і збільшити кількість свої потоків до 200 потоків (це стандартне значення) його можна збільшити, але є ризик виникнення такої ситуації яка називається – контеншн. Контеншн потоків виникає в випадку коли програма породжує більше потоків ніж кількість потоків процессора і в такому випадку потоки породжені програмою спаречаються між собою і таким чином сповільнюють роботу програми. У моєму прикладі Tomcat було масштабовано приблизно до 170 потоків з одним завантаженням 800 користувачів. Результати про стан потоків Spring MVC програми без навантаження наведені у «Додаток В» а результати про стан потоків з навантаженням наведені у «Додаток Г».

Тапер проведемо аналогічні заміри потоків але вже у випадку системи розробленої за допомогою реактивного підходу з використанням Project Reactor. Без навантаження було створено лише один реактивний потік – «reactive-http-nio-1». Він працює постійно. Під навантаженням «Netty» збільшив кількість потоків до 12. Тести відбувалися на машині на якій встановлений 6-ядерний і він працює в 12 потоках, тому «Netty» збільшив їх до кількісті потоків поточного процесора. Всі ці

потоки працюють і не зупиняються. Аналогічні данні по цій метриці представлені в додатках. Результати про стан потоків Spring Webflux Project Reactor програми без навантаження наведені у «Додаток Д» а результати про стан потоків з навантаженням наведені у «Додаток Е».

Наступними даними буде профілювання мікросервісів при навантаженні. Перевірте затримку 10 мс для базової служби (100 200 400 800 одночасних користувачів). 4 стрибки в CPU – це 4 тести навантаження (100, 200, 400, 800 користувачів). На рисунку 3.1 представлено тестові запуски тесту навантаження для мікросервісу написаного за допомогою Servlet API.

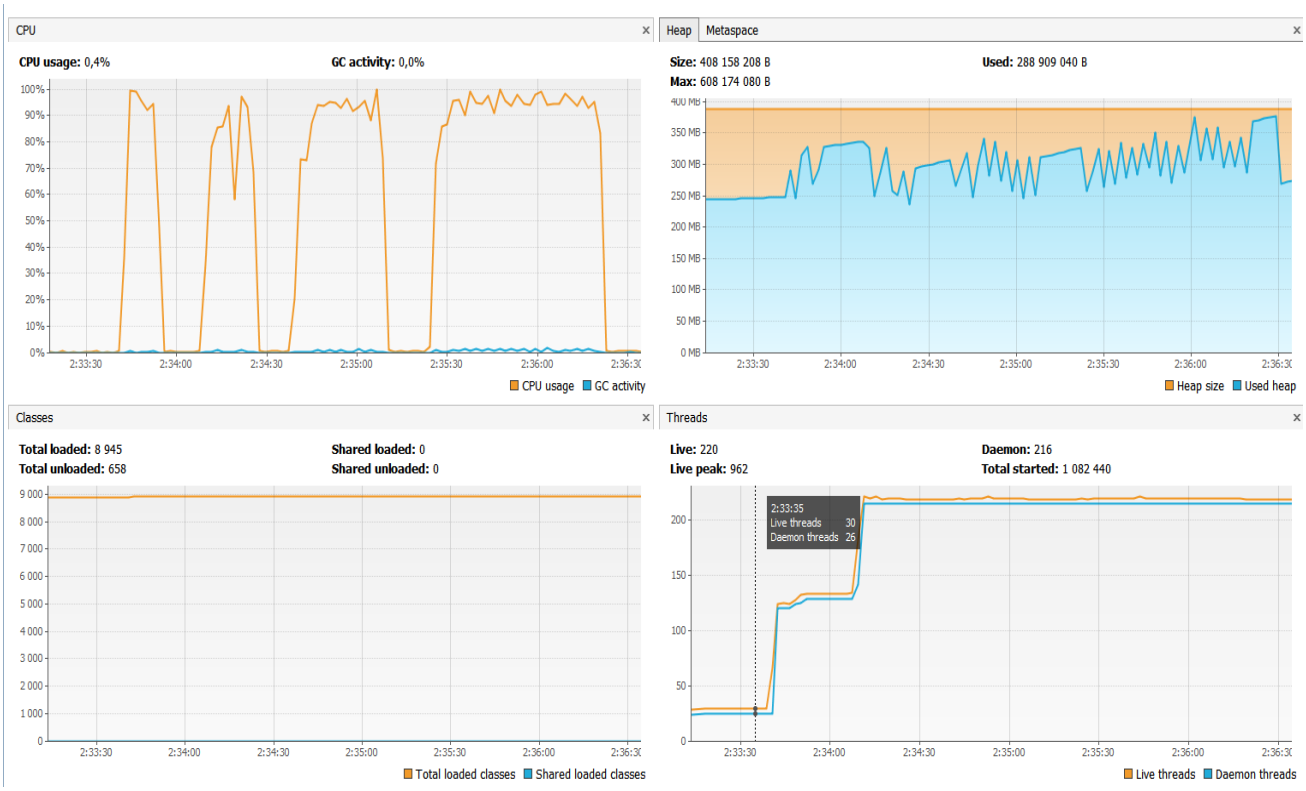


Рисунок 3.1 Профілювання Servlet API мікросервісу за допомогою JMeter

Наступними результатами буде запуск тести навантаження для мікросервісу написаним з використанням CompletableFuture with Java Http Client and Tomcat. CompletableFuture – клас для асинхронної роботи, який дозволяє комбінувати кроки обробки, з'єднуючи їх у ланцюг.Він з'явився в Java8 і слугує для передачі

інформації між паралельними потоками виконання. Фактично це блокуюча черга, здатна передати лише одне посилальне значення. На відміну від звичайної черги, передає також виняток, якщо воно виникло при обчисленні значення, що передається. Клас містить близько 50 методів для виконання, об'єднання і обробки винятків. Розглянемо ми лише деякі з них. На рисунку 3.2 представлено тестові запуски тесту. Цей варінт дуже цікавий бо було використано нативні клієнти які інтегровані напрояму с Java але повертають асинхронну выдповідь. У цьому випадку код написаний таким чином що ми все одно блокуємо потік але комунікація по мережі відбувається асинхронно.

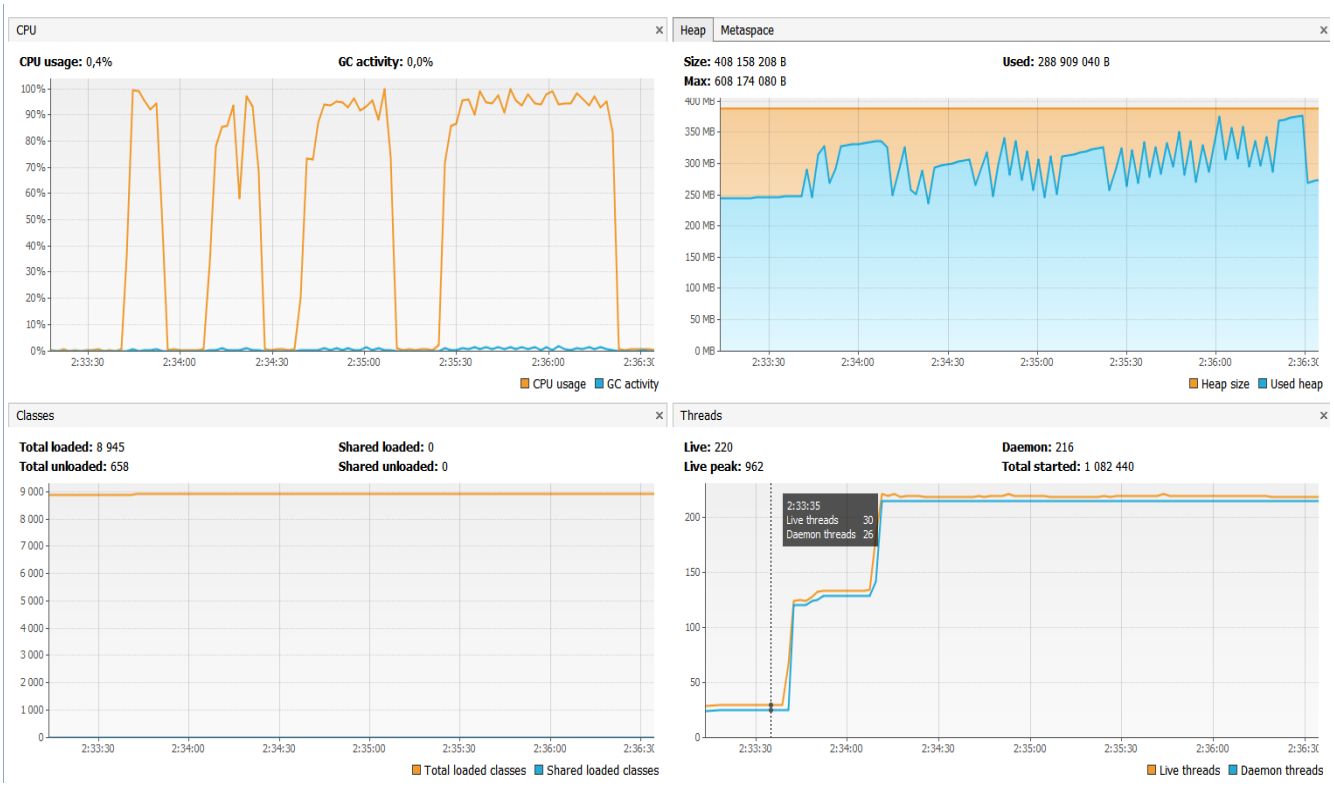


Рисунок 3.2 Профілювання CompletableFuture and Java Http Client мікросервісу за допомогою JMeter

Наступними результатами буде запуск тести навантаження для мікросервісу написаним з WebFlux with WebClient and Apache Client(WebClient and Apache Client have the same memory utilization and thread stuff). Цей варінт дуже цікавий бо було

використано нативні клієнти які інтегровані напрояму с Project Reactor. У цьому випадку код написаний таким чином що не відбувається додатковий перетворень між класами. Також у цьому випадку не створюється ніяких проміжних класів і в теорії використання таких клієнтів має бути най оптимальнішим варіантом. WebClient є потокобезпечним, оскільки він незмінний. WebClient призначений для використання в реактивному середовищі, де нічого не прив'язано до певного потоку. Spring WebClient – це неблокуючий реактивний клієнт для виконання HTTP–запитів, частина фреймворку Spring Webflux. Цей варіант показав найкращі резуальтати за весь час досліду. На рисунку 3.3 представлено тестові запуски тесту.

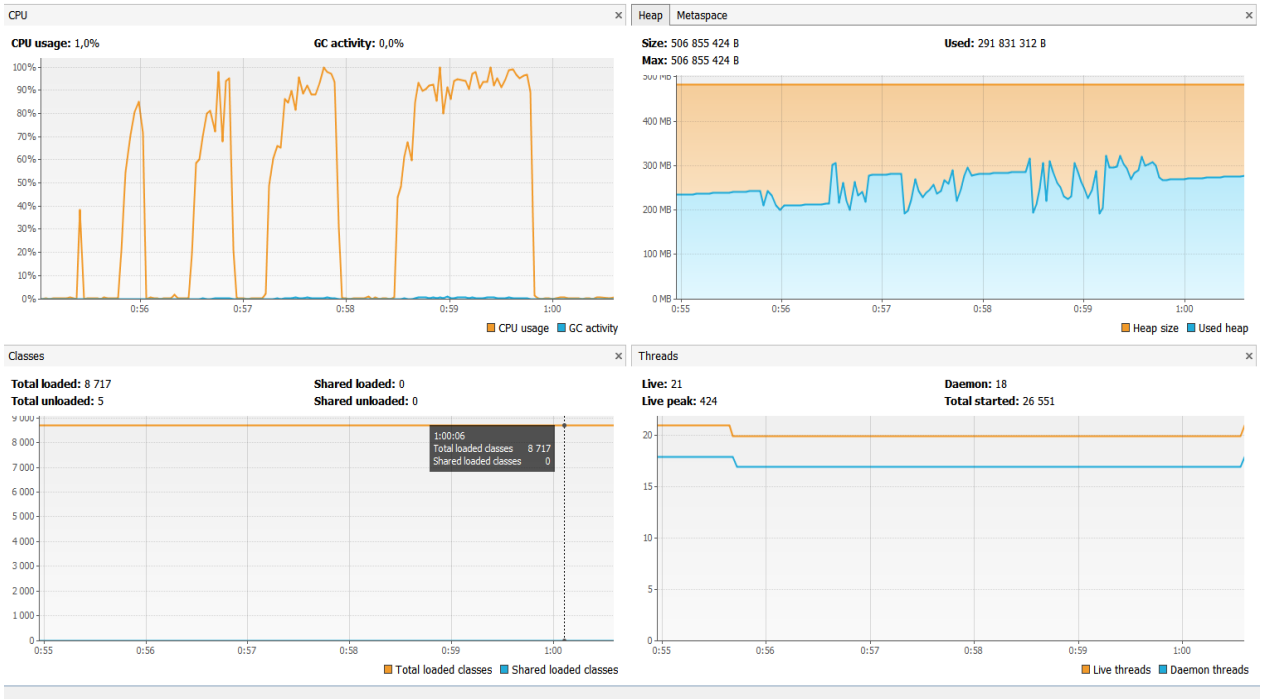


Рисунок 3.3 Профілювання WebFlux with WebClient and Apache Client мікросервісу за допомогою JMeter.

Наступними результатами буде запуск тести навантаження для мікросервісу написаним з WebFlux and Java Http Client. Оскільки в реактивному програмванні всі операціїї слід писати в необлокуючому стилі тож було злоблено обгортку навколо http клієнту щоб він зміг комунікувати з реативними типами. На рисунку

3.4 представлено тестові запуски тесту для 100 користувачів: Всі запуски відбувалися послідновно.

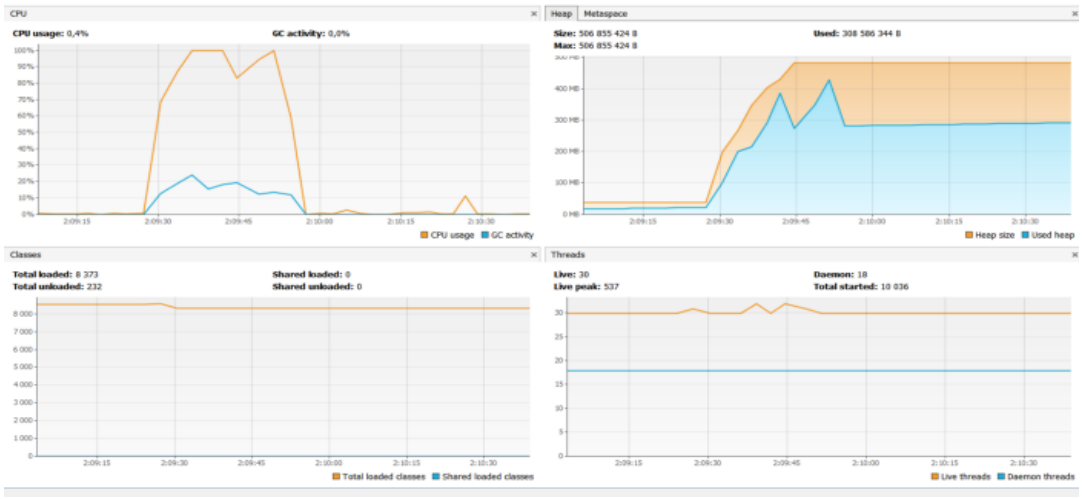


Рисунок 3.4 Профілювання WebFlux and Java Http Client для 100 користувачів мікросервісу за допомогою JMeter.

На рисунку 3.5 представлено тестові запуски тесту для 200 користувачів:

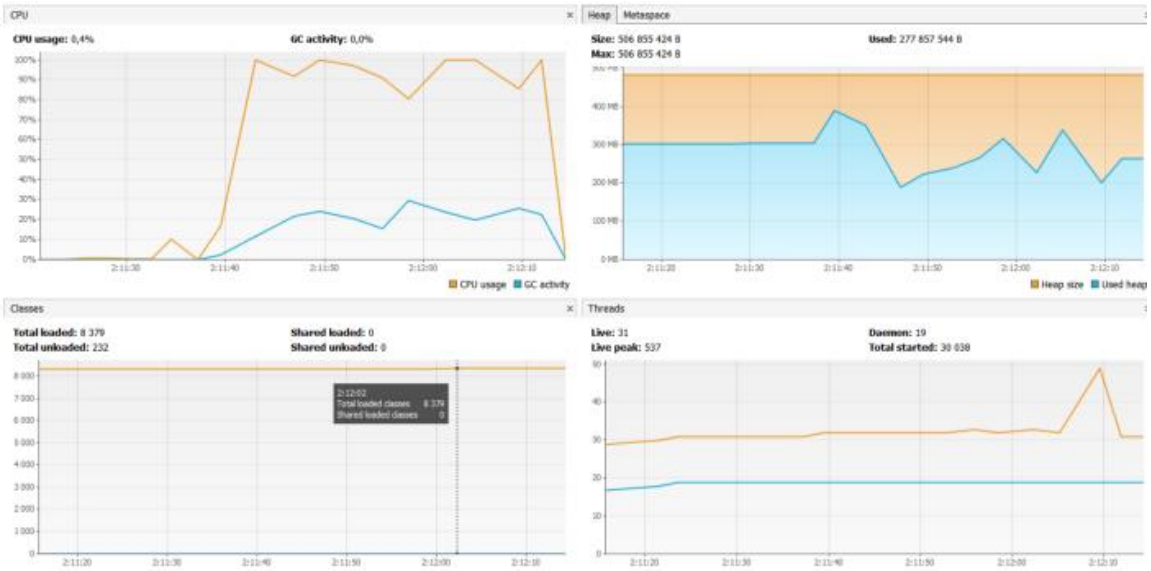


Рисунок 3.5 Профілювання WebFlux and Java Http Client для 200 користувачів мікросервісу за допомогою JMeter.

На рисунку 3.6 представлено тестові запуски тесту для 400 користувачів.

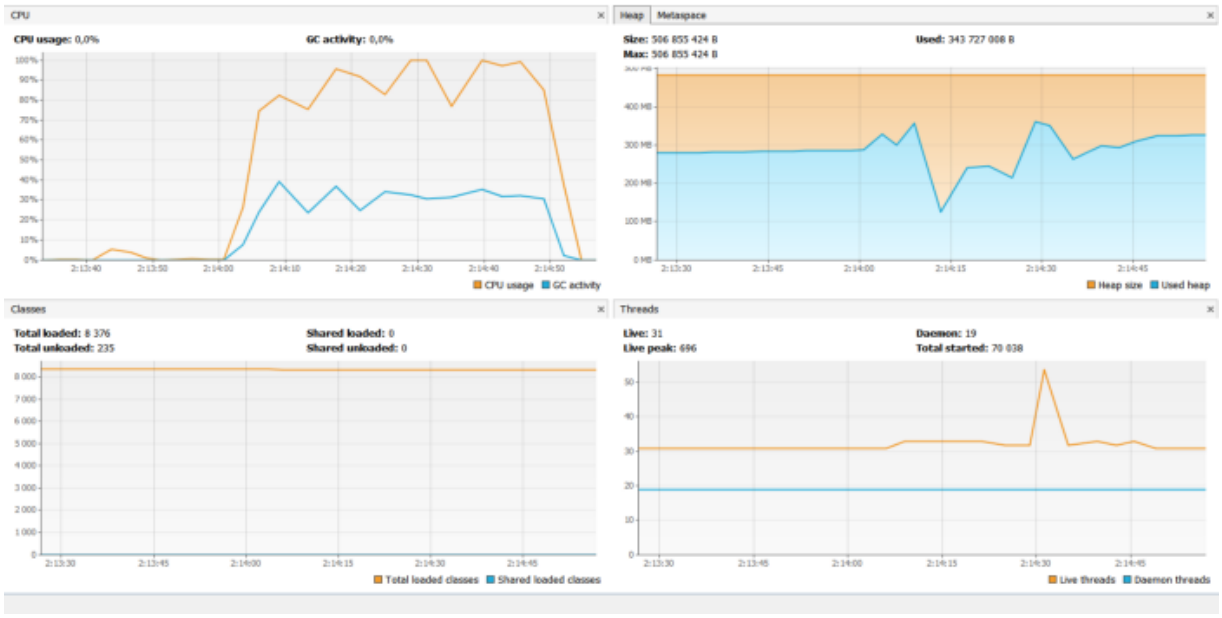


Рисунок 3.6 Профілювання WebFlux and Java Http Client для 400 користувачів мікросервісу за допомогою JMeter.

На рисунку 3.7 представлено тестові запуски тесту для 800 користувачів:

Рисунок 3.7 Профілювання WebFlux and Java Http Client для 800 користувачів мікросервісу за допомогою JMeter.

Тепер збільшимо затримку до 500 мілі–секунд для 100, 200, 400, 800 паралельних користувачів. На рисунку 3.8 представлено тестові запуски тесту для 100 користувачів мікросервісу на Servlet API:

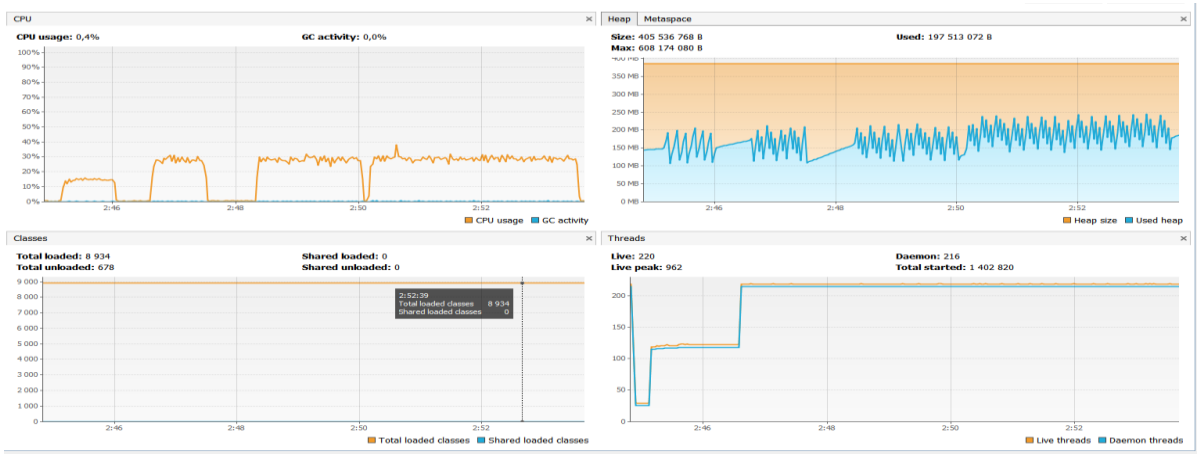


Рисунок 3.8 Профілювання Servlet API з 500 мс затримкою мікросервісу за допомогою JMeter

На рисунку 3.9 представлено тестові запуски тесту для 100, 200, 400, 800 користувачів мікросервісу на CompletableFuture with Java Http Client and Tomcat.

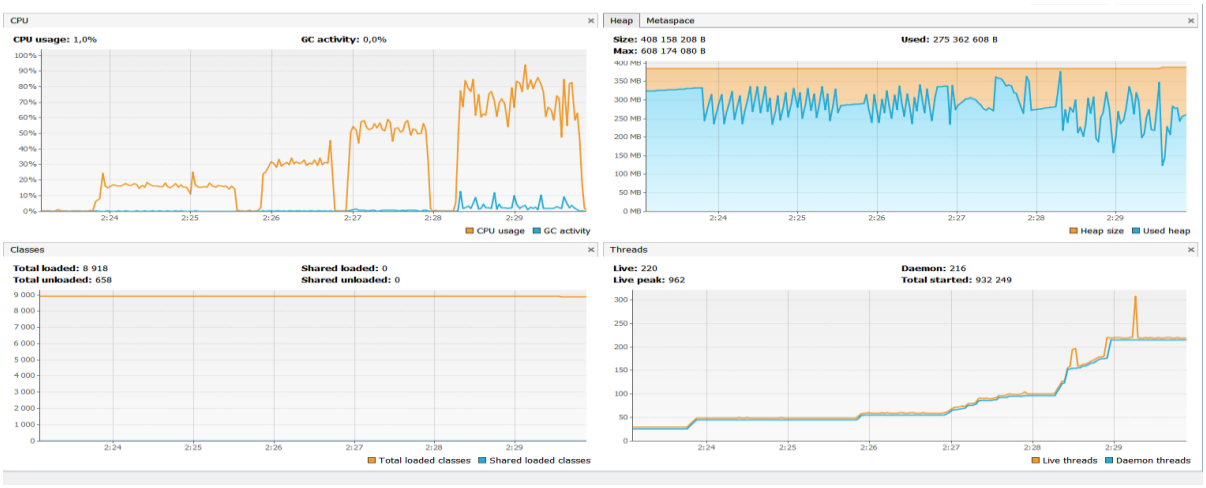


Рисунок 3.9 Профілювання CompletableFuture with Java Http Client and Tomcat з 500 мс затримкою мікросервісу за допомогою JMeter

На рисунку 3.10 представлено тестові запуски тесту для 100, 200, 400, 800 користувачів мікросервісу на CompletableFuture with Apache Http Client.

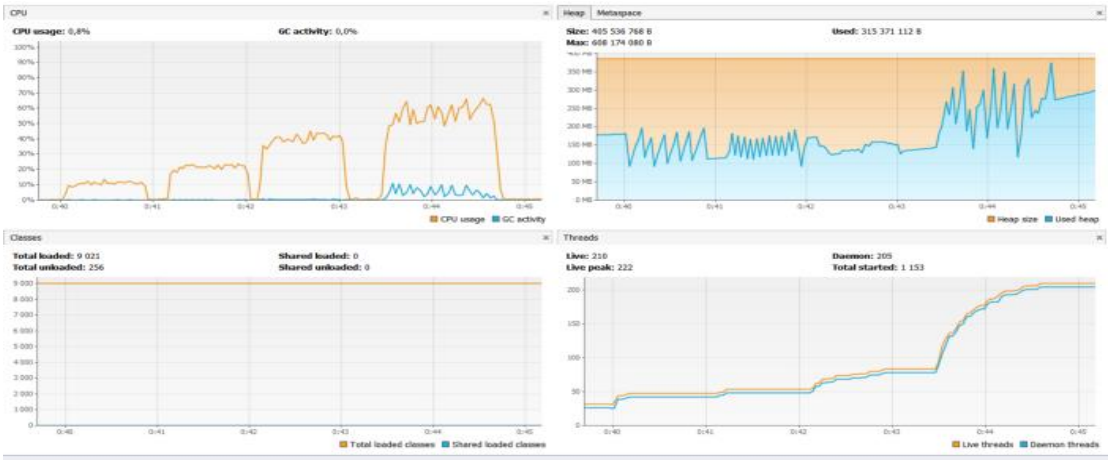


Рисунок 3.10 Профілювання на CompletableFuture with Apache Http Client and Tomcat з 500 мс затримкою мікросервісу за допомогою JMeter

На рисунку 3.11 представлено тестові запуски тесту для 100, 200, 400, 800 користувачів мікросервісу WebFlux with WebClient and Apache Client (WebClient and Apache Client have the same memory utilization and thread stuff).

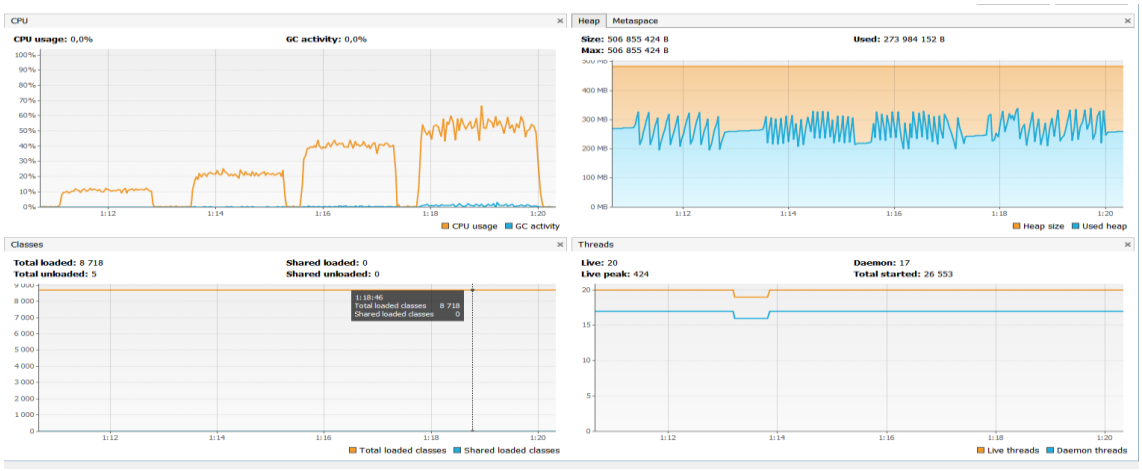


Рисунок 3.11 Профілювання WebFlux with WebClient and Apache Client з 500 мс затримкою мікросервісу за допомогою JMeter

На рисунках 3.12, 3.13, 3.14, 3.15 представлено тестові запуски тесту для 100, 200, 400, 800 користувачів мікросервісу WebFlux and Java Http Client.

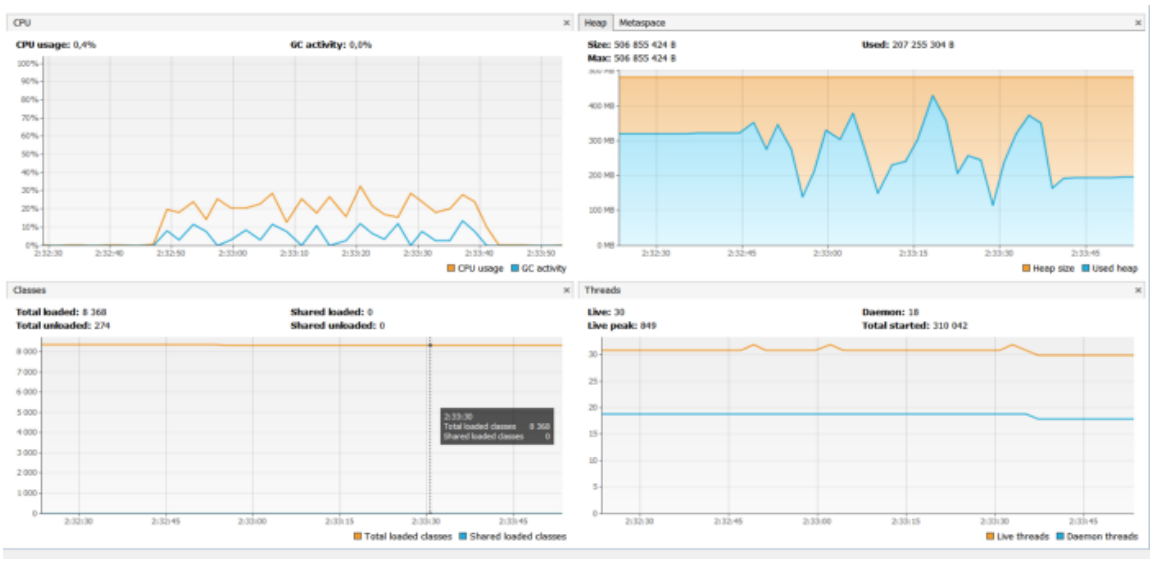


Рисунок 3.12 Профілювання на WebFlux and Java Http Client з 500 мс затримкою мікросервісу та 100 паралельними користувачами за допомогою JMeter

Рисунок 3.13 Профілювання на WebFlux and Java Http Client з 500 мс затримкою мікросервісу та 200 паралельними користувачами за допомогою JMeter

Рисунок 3.14 Профілювання на WebFlux and Java Http Client з 500 мс затримкою мікросервісу та 400 паралельними користувачами за допомогою JMeter

Рисунок 3.15 Профілювання на WebFlux and Java Http Client з 500 мс затримкою мікросервісу та 800 паралельними користувачами за допомогою JMeter

Підведемо підсумки. Мікросервіс написаний за допомогою Servlet API добре працює лише у випадку, коли базова служба швидка (10 мс). Неблокувальний підхід за допомогою Servlet є досить хорошим рішенням і для випадку, коли базова служба повільна (500 мс). Він не витримує конкуренції з Webflux(Project Reactor) лише у разі великої кількості запитів. Spring Webflux з клієнтами WebClient і Apache виграє у всіх випадках. Найбільш істотна різниця (в 4 рази швидше, ніж блокування сервлетів), коли базова служба повільна (500 мс). Він на 15–20% швидше, ніж неблокуючий сервлет із CompetableFuture. Крім того, він не створює багато потоків у порівнянні з Servlet (20 проти 220). На жаль, ми не можемо використовувати WebFlux скрізь, тому що для нього потрібні асинхронні драйвери/клієнти. В іншому випадку ми повинні створити власні пули потоків/обгортки.

Клієнт Java 11 Http повільніше, ніж клієнт Apache Http (зниження продуктивності ~30%) для одноядерного сервера з 1 Гб RAM Spring WebClient має таку ж продуктивність, як і Apache Http Client для одноядерного екземпляра сервера RAM 1 ГБ. Комбінація моделей часу виконання WebFlux і Java 11 Http Client не працює добре, якщо у вас лише одне ядро і мало оперативної пам’яті.

Необхідно також сказати що данний тест є синтетичним та він лише моделює певний кейс з реального життя. Щоб вибрати підходящий варіант розробки прогорамного забезпечення слід врахувати безліч факторів і не завжди реактивний підхід буде працювати гарантовано краще за будь який інший.

Висновки до розділу

У цьому розділі було описано розробку високонавантажених систем їх порівняльний аналіз на основі замірів та результатів метрик при тестах навантаження. У результаті було побудовано отримано безліз графіків які показують ефективність розроблених систем та їх відповідніть до терміну

hightload. Було зроблено висновки щодо того яка система є найбільш ефективною та підходить для розробки високо навантаженої системи. Тож підсумуємо:

- WebFlux Project Reactor – це не про швидкість опрацювання запитів до сервісів, він не пришвидшує роботу конкретної операції або щось інше, він дає змогу використовувати ресурси машини, а саме потоки більш ефективно не блокуючи їх і дає змогу збільшити одночасну обробку великої кількості запитів;
- потрібно заздалегіть правильно продумати архітектуру програми, також важливо невикористовувати блокуючі з'єднання. Система повинна відповідати вимогам Реактивного маніфесту;
- для того щоб позбутися блокування бази даних або інтеграції з повільними сервісами, необхідно використовувати тільки реактивні драйвери та бібліотеки;
- WebFlux найкраще підходить дл систем де необхідно підтримувати велику кількості одночасних запитів (системи з високим навантаженням);
- якщо в системі існує велика кількість блокуючих з’єднань, або неможливо реалізувати драйвер або бібліотеку за реактивним маніфестов, а також якщо перед системою не стоїть завдання обробляти велику кількість одночасних запитів, це наприклад може бути якась внутрішня система для ведення бухгалтерського обліку якою будуть користуватися лише бухгялтери компанії, то такій системі не потрібно враховувати цей фактор і буде краще звернути увагу на стандартні реалізації MVC на tomcat.

ВИСНОВКИ

В ході виконання магістерської дипломної роботи було розглянуто проблему розробки високонавантажених систем та основних проблем при їх розробці на Java. Сучасний стан інформатизації суспільства вимагає від розробників програмного забезпеченння технологій, що могли опрацьовувати велику кільсть баз даних швидко і надійно. Для пришвидшення роботи системи, збільшення рівню надійності, а також витримування більших об’ємів даних через систему та їх обробки необхідно розробити підходи, які будуть описувати проблеми та способи їх вирішення через аналіз та моніторинг системи при роботі з системами. Саме в ході виконання магістерської дипломної роботи і було розглянуто проблему розробки високонавантажених систем та основних проблем при їх розробці на Java.

У процесі виконання роботи отримали наступні результати:

1. Аналіз, який був проведений у сфері високонавантажених систем дав змогу розробити системи та покрити основні ситуаціїї, які можуть статися при розробці таких систем. Також були обрані підходи до розробки, збору метрик та графіків з результатами роботи системи при різному навантаженні.

2. Сформовано вимоги та критерії до системи та проаналізовано існуючі рішення на предмет відповідності. На підставі всього сказаного були висунуті вимоги до програмного забезпечення та успішно реалізовані.

3. Розроблено дві системи, на основі яких формується результат роботи таких систем у різних ситуаціях. Розроблено тести навантаження, які дозволяють відстежувати поведінку або регрессію, та робити висновки щодо поведінки системи при навантеженню та вирішувати такі проблеми досить швидко.

Результати даної роботи можна використовувати при розробці реальних високонавантажених систем, доповнюючи їх своїми метриками, по яким можна буде сказати що система є надійною та може працювати при високому навантаженню. Відслідковувати та вести постійний монітронг критичних функції системи є дуже важливим при розробці програмного забезпечення у сучасному світі.

ПЕРЕЛІК ПОСИЛАНЬ

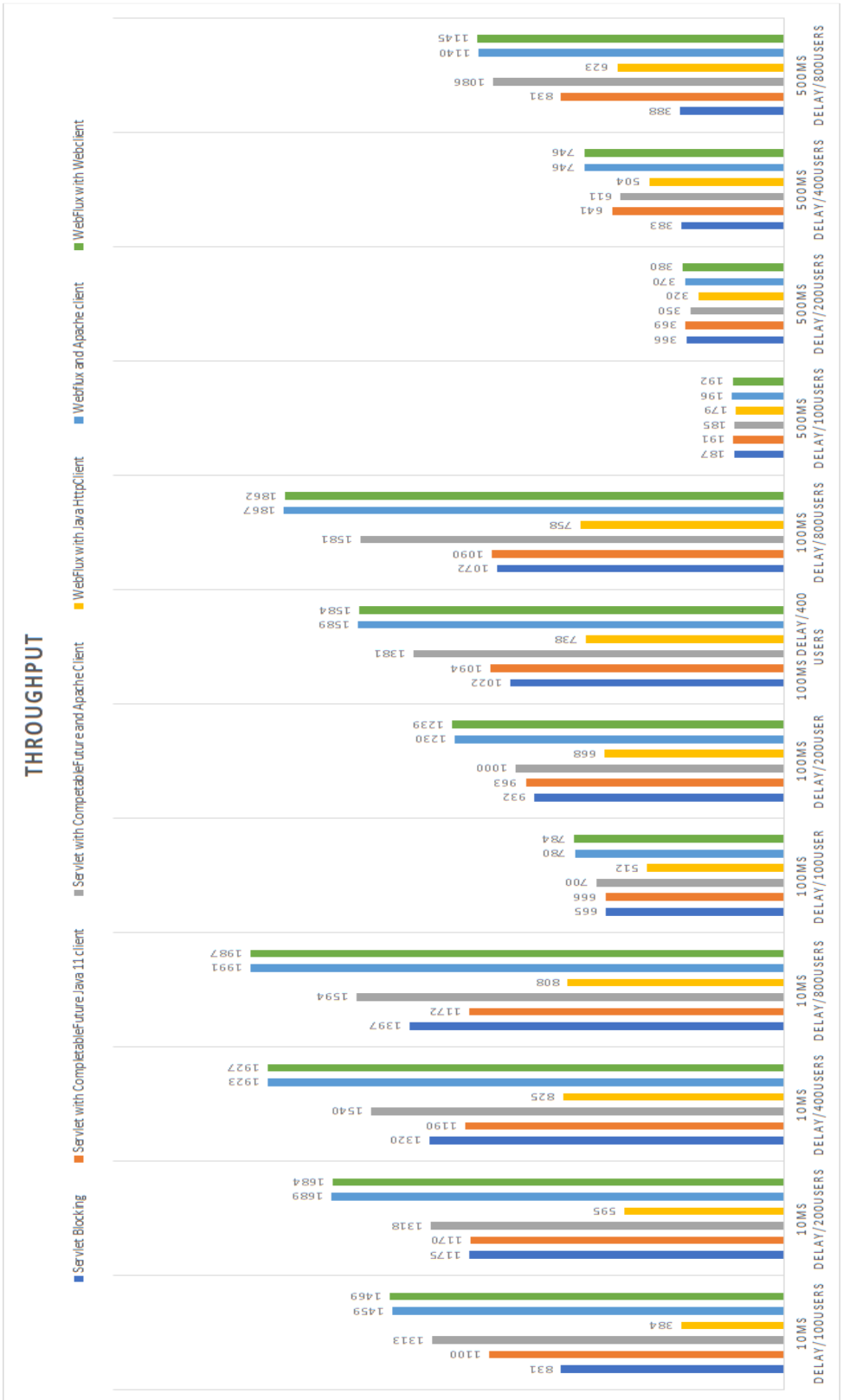
1. *Stonebraker M., Çetintemel U.* ‘One Size Fits All’: An Idea Whose Time Has Come and Gone // 21st International Conference on Data Engineering (ICDE), April 2005 [Електронний ресурс]. http://citeseerx.ist.psu.edu/ viewdoc/download?doi=10.1.1.68.9136&rep=rep1&type=pdf.
2. *Foote B., Yoder J.* Big Ball of Mud // 4th Conference on Pattern Languages of Programs (PLoP), September 1997 [Електронний ресурс]. – http://www.laputan.org/pub/foote/mud.pdf.
3. *Hamilton J.* On Designing and Deploying Internet–Scale Services // 21st Large Installation System Administration Conference (LISA), November 2007 [Електронний ресурс]. –: https://www.usenix.org/ legacy/events/lisa07/tech/full_papers/hamilton/hamilton.pdf.
4. *Kreps J.* Getting Real About Distributed System Reliability. March 19, 2012 [Електронний ресурс]. –: http://blog.empathybox.com/post/19574936361/ getting–real–about–distributed–system–reliability.
5. *Ford D., Labelle F., Popovici F. I., et al.* Availability in Globally Distributed Storage Systems // 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI), October 2010 [Електронний ресурс].– https://static.googleusercontent.com/media/research.google.com/ ru//pubs/ archive/36737.pdf.
6. *Beach B.* Hard Drive Reliability Update Sep 2014. – September 23, 2014 [Електронний ресурс]. –https://www.backblaze.com/blog/hard– drive– reliability–update–september–2014/.
7. *Voss L.* AWS: The Good, the Bad and the Ugly. December 18, 2012 [Електронний ресурс]–:https://web.archive.org/web/20160406004621/http:// blog.awe.sm/ 2012/12/18/aws–the–good–the–bad–and–the–ugly/.
8. *Gunawi H. S., Hao M., Leesatapornwongsa T., et al.* What Bugs Live in the Cloud? // 5th ACM Symposium on Cloud Computing (SoCC), November 2014 [Електронний ресурс]. –http://ucare.cs.uchicago.edu/ pdf/socc14–cbs.pdf.

9. *Moseley B., Marks P.* Out of the Tar Pit // BCS Software Practice Advancement (SPA), 2006 [Електронний ресурс]. –: http://citeseerx.ist.psu. edu/viewdoc/summary?doi=10.1.1.93.8928.
10. *Breivold H. P., Crnkovic I., Eriksson P. J.* Analyzing Software Evolvability // 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC), July 2008 [Електронний ресурс]. –http:// www.mrtc.mdh.se/publications/1478.pdf.
11. *Cook R. I.* How Complex Systems Fail // Cognitive Technologies Laboratory, April 2000 [Електронний ресурс]. –http://web.mit.edu/2.75/ resources/random/How%20Complex%20Systems%20Fail.pdf.
12. *Sommers K.* After all that run around, what caused 500ms disk latency even when we replaced physical server? November 13, 2014 [Електронний ресурс]. –: https://twitter.com/kellabyte/status/532930540777635840.
13. *DeCandia G., Hastorun D., Jampani M., et al.* Dynamo: Amazon's Highly Availa– ble Key–Value Store // 21st ACM Symposium on Operating Systems Princip– les (SOSP), October 2007 [Електронний ресурс]. – http:// www.allthingsdistributed.com/files/amazon–dynamo–sosp2007.pdf.
14. *Linden G.* Make Data Useful // slides from presentation at Stanford University Data Mining class (CS345), December 2006 [Електронний ресурс]. – http://glinden.blogspot.com.by/2006/12/slides–from–my–talk–at–stanford.html.
15. *Everts T.* The Real Cost of Slow Time vs Downtime. November 12, 2014 [Електронний ресурс]. – https://blog.radware.com/applicationdelivery/ wpo/2014/11/real–cost–slow–time–vs–downtime–slides/.
16. *Brutlag J.* Speed Matters for Google Web Search. June 22, 2009 [Електронний ресурс]. – https://research.googleblog.com/2009/06/speed– matters.html.
17. *Treat T.* Everything You Know About Latency Is Wrong. December 12, 2015 [Електронний ресурс]. – http://bravenewgeek.com/everything– you–know– about–latency–is–wrong/.

18. *Dean J., Barroso L. A.* The Tail at Scale // Communications of the ACM, volume 56, number 2, pages 74–80, February 2013 [Електронний ресурс]. – https://cacm.acm.org/magazines/2013/2/160173–the–tail–at–scale/abstract
19. *Cormode G., Shkapenyuk V., Srivastava D., Xu B.* Forward Decay: A Practical Time Decay Model for Streaming Systems // 25th IEEE International Conference on Data Engineering (ICDE), March 2009 [Електронний ресурс]. – http://dimacs.rutgers.edu/~graham/pubs/papers/fwddecay.pdf.
20. *Dunning T., Ertl O.* Computing Extremely Accurate Quantiles Using t–Digests. March 2014 [Електронний ресурс]. – Режим доступа: https://github.com/tdunning/t–digest.
21. *Tene G.* HdrHistogram [Електронний ресурс]. – Режим доступа: http://www.hdrhistogram.org/.
22. *Schwartz B.* Why Percentiles Don’t Work the Way You Think. December 7, 2015 Електронний ресурс]. – Режим доступа: https://www.vividcortex.com/blog/why–percentiles–dont–work–the–way–you–think.

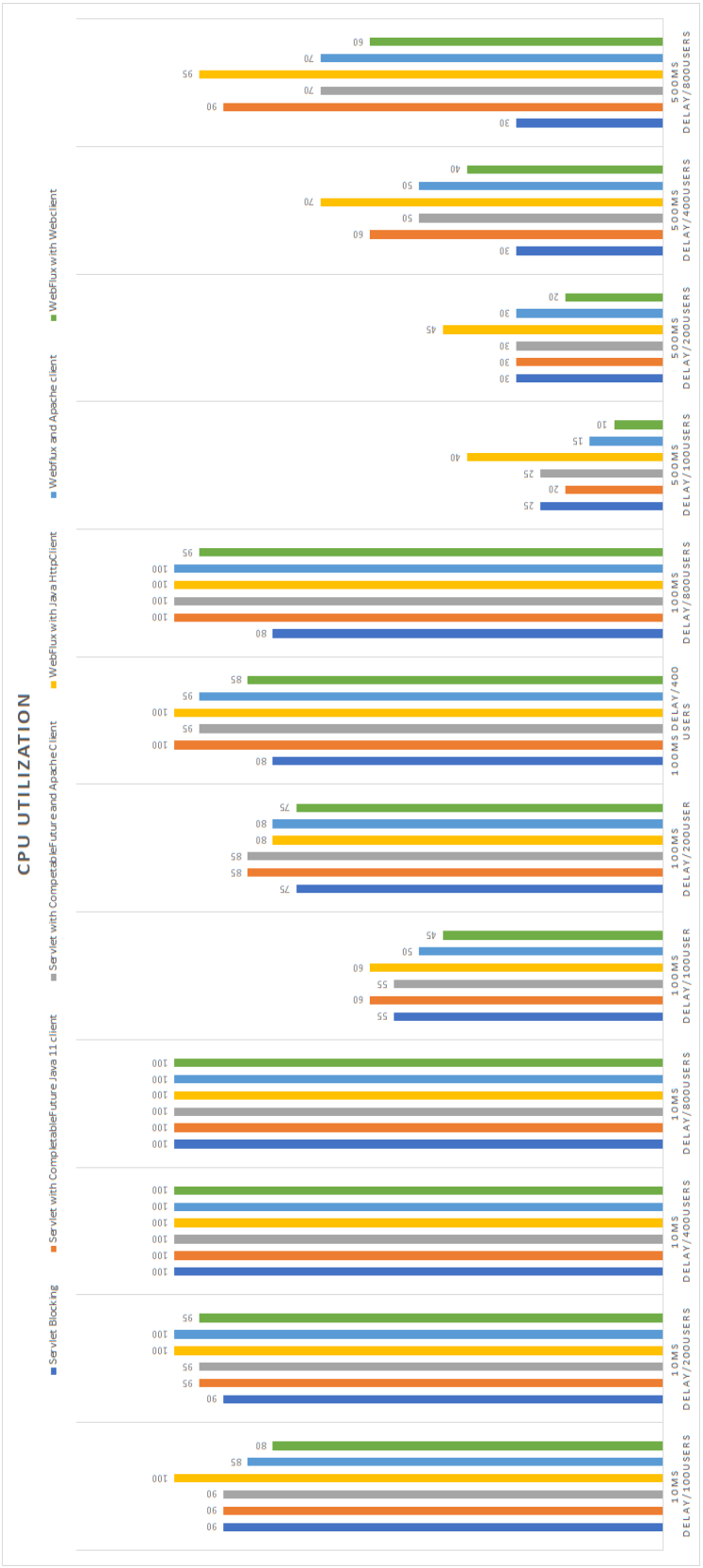
ДОДАТОК А

Результат тестів навантаження по метриці Throughput



ДОДАТОК Б

Результати тестів навантаження для метрики CPU Uutilization



ДОДАТОК В

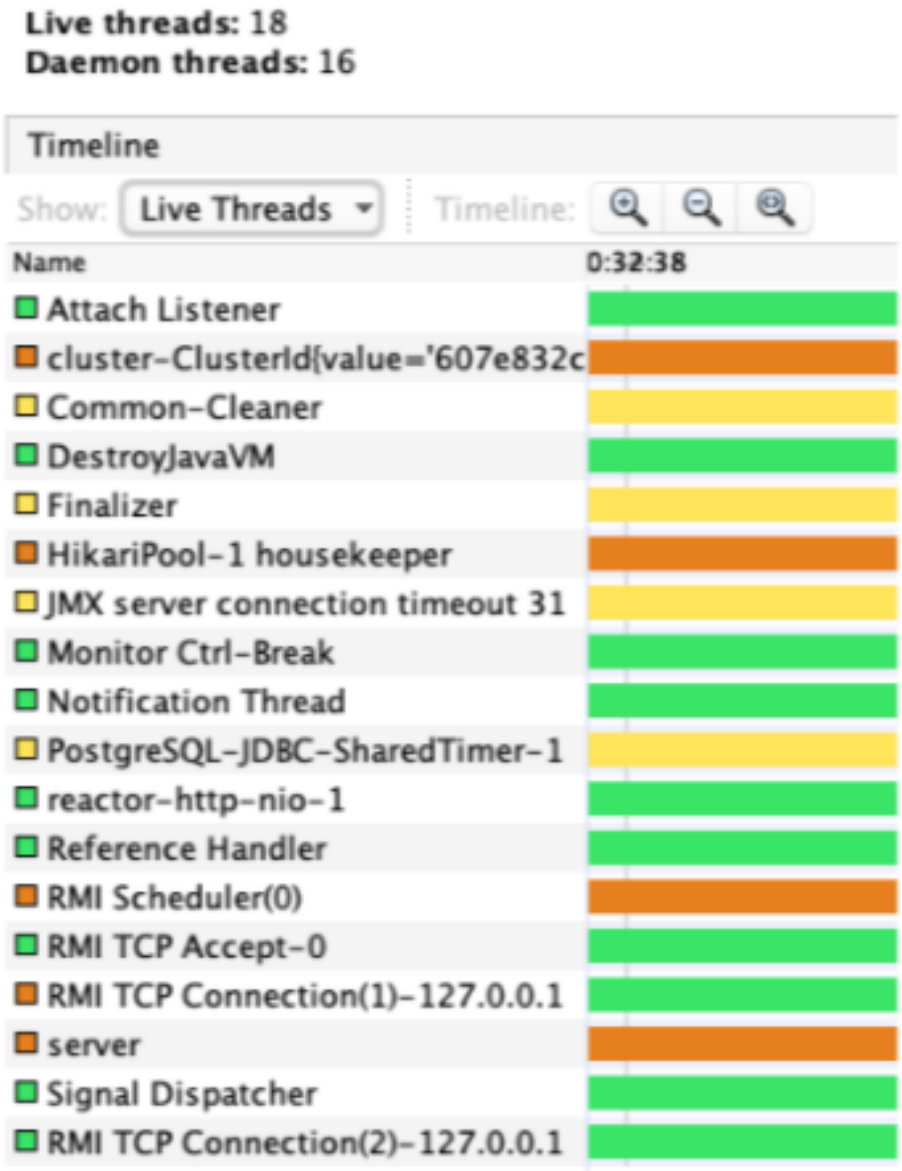
Результати стану потоків Spring MVC програми без навантаження

ДОДАТОК Г

Результати стану потоків Spring MVC програми під навантаженням

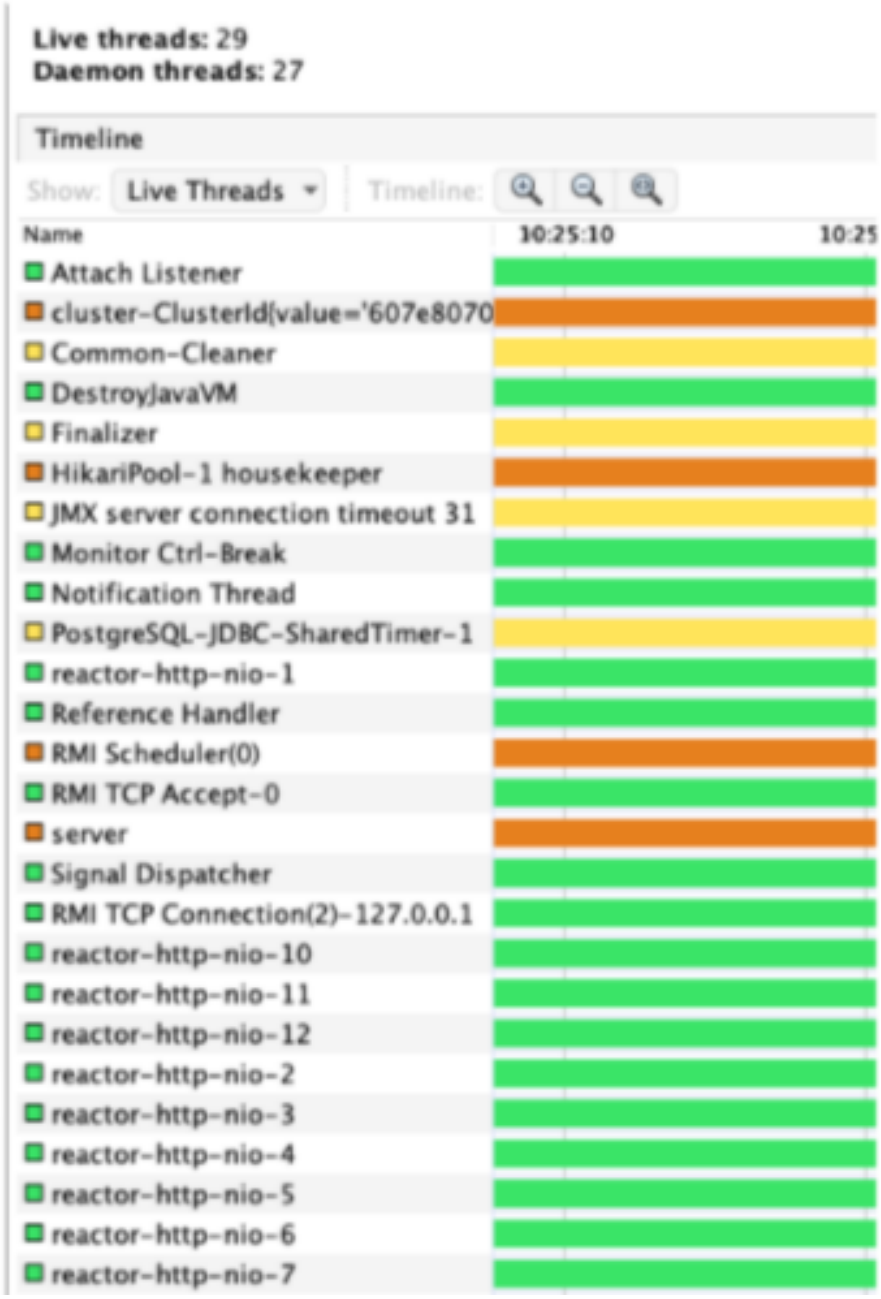
ДОДАТОК Д

Результати стану потоків реактивної програми без навантаження



ДОДАТОК Е

Результати стану потоків реактивної програми під навантаженням



ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ (Презентація)

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

НАВЧАЛЬНО–НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра інженерії програмного забезпечення

Магістерська робота

ЗАСТОСУВАННЯ МЕТОДОЛОГІЇ РЕАКТИВНОГО ПІДХОДУ ПРИ РОЗРОБЦІ ВИСОКОНАВАНТАЖЕНИХ СИСТЕМ ІЗ ЗАСТОСУВАННЯМ PROJECT REACTOR JAVA

Виконав: студент групи ПДМ – 61, Деревець Артем Валерійович
Керівник: : к.п.н., доц. кафедри ІПЗ Шевченко С.М

Київ - 2021

МЕТА, ОБ'ЄКТА ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи: Удосконалення методології реактивного підходу на базі Project Reactor Java.

Об'єкт дослідження: Функціювання високонавантажених систем на Java

Предмет дослідження: Методи та засоби розробки високонавантежних систем на базі Project Reactor Java

ПОСТАНОВКА ЗАДАЧІ

- Дослідити та скласти порівняльний аналіз систем, написаних з використанням блокуючого Sevlet API, і систем, які використовують асинхронний неблокуючий підхід на базі Event Loop.
- Описати основні підходи для розробки високонавантажених систем
- Визначити недоліки та особливості обох підходів.

3

ІСНУЮЧІ ПІДХОДИ ПРИ РОЗРОБЦІ ВИСОКО НАВАНТАЖЕНИХ СИСТЕМ НА JAVA

- Servlet Model (Servlet based systems) - Spring MVC;
- Reactive Model (Event Loop Based systems) - Spring Webflux with Project Reactor, RxJava, Vertx;

4

ВХІДНІ ДАННІ

Оточення:

- Spring Boot:2.5.6 (latest);
- Java: 11 OpenJDK;
- Node: t2.micro (Amazon Linux);
- Http Clients: Java 11 Http Client, Apache Http Client, Spring WebClient.

Тестові додатки:

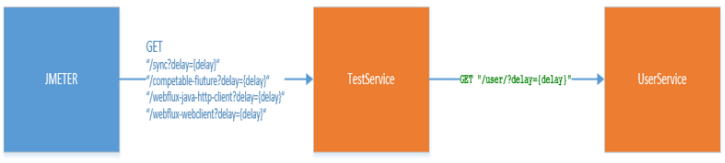
- Spring MVC
- Spring Webflux (Project Reactor)

Типи тестових замірів:

- 100 користувачів (з затримкою та без)
- 200 користувачів (з затримкою та без)
- 400 користувачів (з затримкою та без)
- 800 користувачів (з затримкою та без)

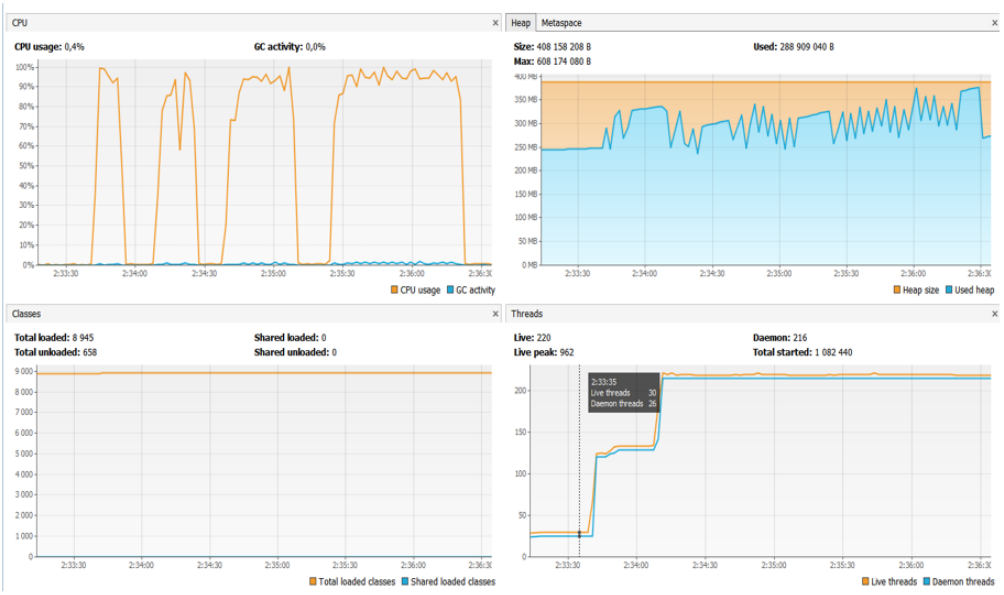
5

ТЕСТОВИЙ СЦЕНАРІЙ



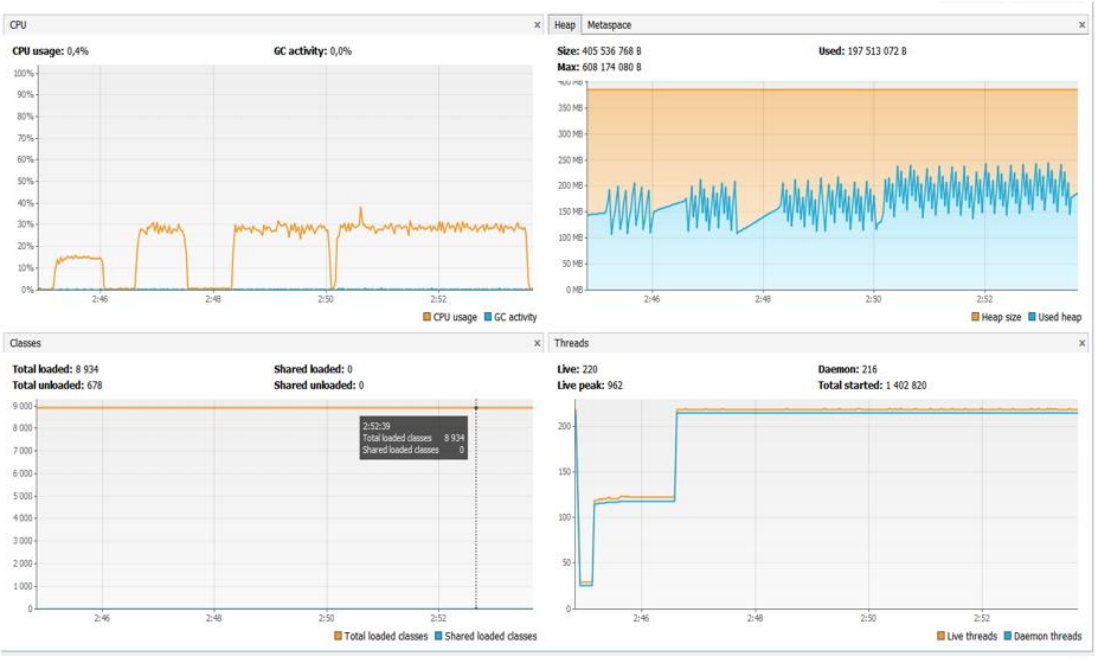
6

РЕЗУЛЬТАТИ SPRING MVC У ТЕСТІ БЕЗ ЗАТРИМКИ



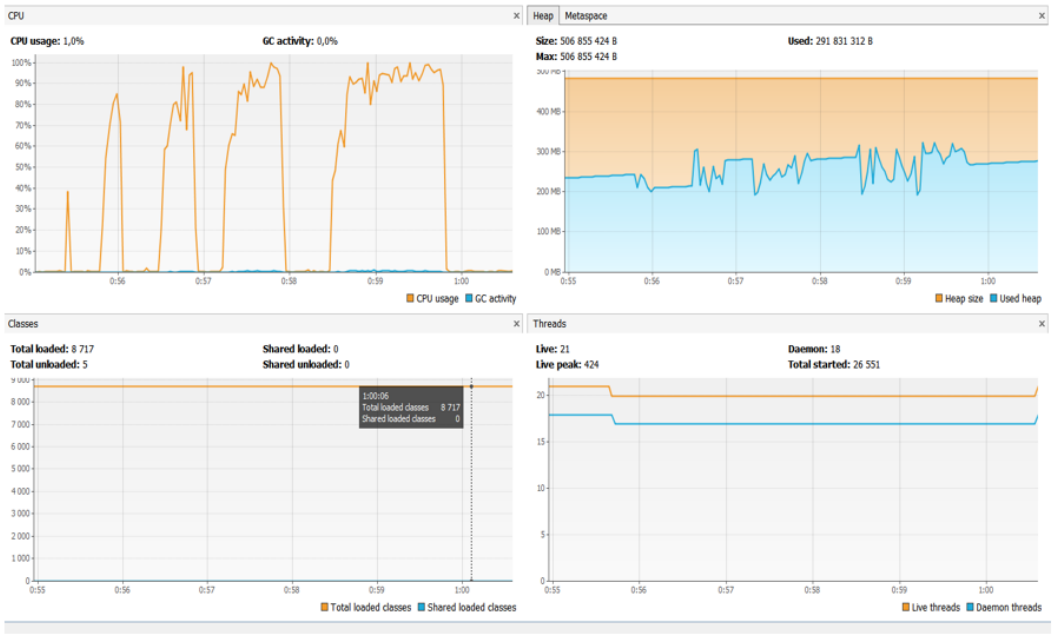
7

РЕЗУЛЬТАТИ SPRING MVC У ТЕСТІ З ЗАТРИМКОЮ 500 МС



8

РЕЗУЛЬТАТИ SPRING WEBFLUX (PROJECT REACTOR)



9

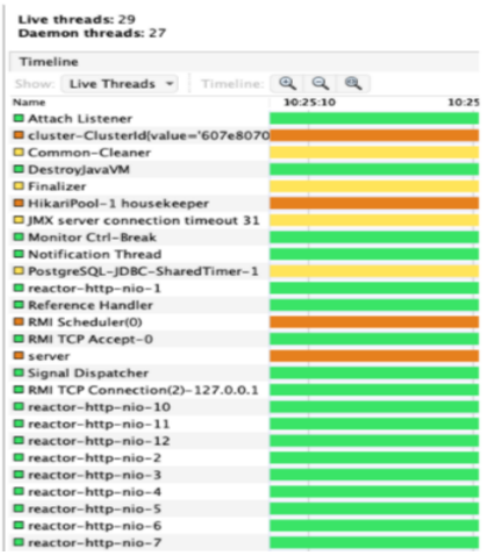
РЕЗУЛЬТАТИ SPRING WEBFLUX (PROJECT REACTOR) З ЗАТРИМКОЮ 500MS

10

СТАН ПОТОКІВ ПРИ НАВАНТАЖЕННІ

Spring MVC

Spring WebFlux



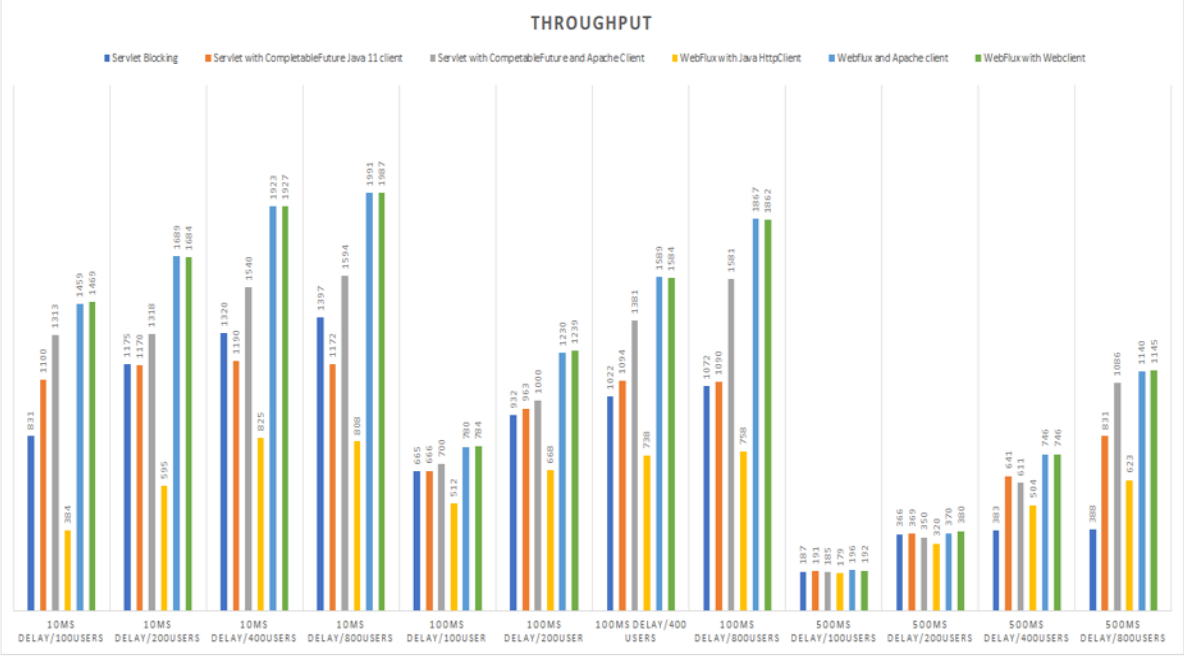
– Заблокований (Park)

– Працюючий (Running)

– Очікуючий (Wait)

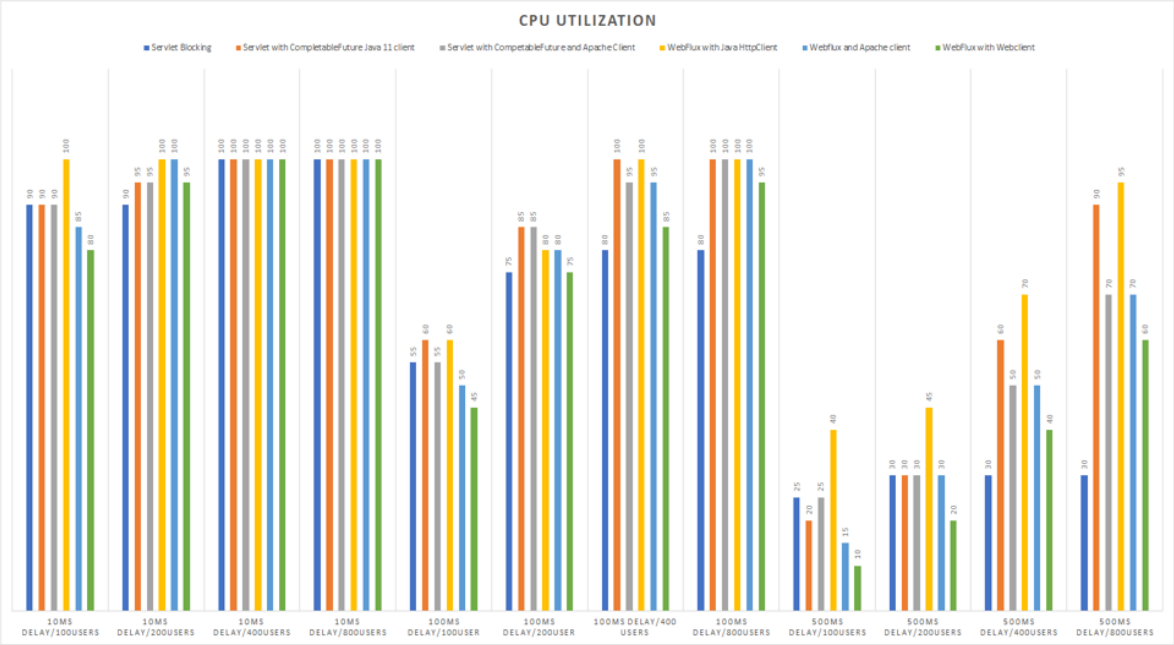
11

РЕЗУЛЬТАТИ ПРОПУСКНОЇ ЗДАТНОСТІ



12

РЕЗУЛЬТАТИ ЗАВАНТАЖЕННЯ ПРОЦЕСОРА



13

ВИСНОВКИ

У процесі виконання роботи отримали наступні результати:

- Аналіз, який був проведений у сфері високонавантажених систем дав змогу розробити системи та покрити основні ситуаційї, які можуть статися при розробці таких систем. Також були обрані підходи до розробки, збору метрик та графіків з результатами роботи системи при різному навантаженні.
- Сформовано вимоги та критерії до системи та проаналізовано існуючі рішення на предмет відповідності. На підставі всього сказаного були висунуті вимоги до програмного забезпечення та успішно реалізовані.
- Розроблено дві системи, на основі яких формується результат роботи таких систем у різних ситуаціях. Розроблено тести навантаження, які дозволяють відстежувати поведінку або регресію, та робити висновки щодо поведінки системи при навантеженні та вирішувати такі проблеми досить швидко.

14

Апробація результатів

1. Деревець А. Методологія реактивного підходу при розробці високонавантажених систем із застосуванням Project Reactor Java // Конференція «Світ телекомунікації та інформатизації», Збірник матеріалів XIII Міжнародної науково-технічної конференції студентства та молоді, 21.10.21, Київ, ДУТ.

2. Деревець А. Високонавантажені системи: суть, вимоги, критерії до розробки // А. Деревець, С. Шевченко / Зв'язок, ДУТ.

15

ДЯКУЮ ЗА УВАГУ!

16