ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНИХ НАУК

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Vert.x та Spring WebFlux у реактивному програмуванні на Java.»

на здобуття освітнього ступеня магістра
зі спеціальності 122 Комп'ютерні науки
(код, найменування спеціальності)
освітньо-професійної програми Комп'ютерні науки
(назва)

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Максим ПРИХОДЬКО
(підпис) *(Ім'я, ПРІЗВИЩЕ здобувача)*

Виконав:
здобувач вищої освіти Максим ПРИХОДЬКО
група КНДМ-63

Керівник: Сергій ІЩЕРЯКОВ
науковий ступінь, вчене звання к.т.н., доцент

Рецензент:
науковий ступінь, вчене звання
_____
(Ім'я, ПРІЗВИЩЕ)

Київ 2024

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Комп'ютерних наук

Ступінь вищої освіти Магістр

Спеціальність 122 Комп'ютерні науки

Освітньо-професійна програма Комп'ютерні науки

ЗАТВЕРДЖУЮ
Завідувач кафедрою Комп'ютерних наук

_____ Віктор ВИШНІВСЬКИЙ
«_____» _____ 2024 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Приходько Максим Юрійович
(прізвище, ім'я, по батькові здобувача)

1. Тема кваліфікаційної роботи: «Vert.x та Spring WebFlux у реактивному програмуванні на Java.»

керівник кваліфікаційної роботи Сергій ІЩЕРЯКОВ к.т.н., доцент,
(Ім'я, ПРІЗВИЩЕ науковий ступінь, вчене звання)
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «15» жовтня 2024р. №320

2. Строк подання кваліфікаційної роботи «15» грудня 2024р.

3. Вихідні дані до кваліфікаційної роботи: науково-технічна література, інструменти для розробки реактивних систем, методи розробки програмного забезпечення.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
   4.1. Дослідження концепції реактивних систем
   4.2. Дослідження технологій і архітектури реактивних фреймворків
   4.3. Проведення тестування фреймворків і порівняння за різним критеріями

5. Перелік графічного матеріалу: презентаці
   5.1. Дослідження концепції реактивних систем
   5.2 Концепція реактивних систем.
   5.3 Архітектура реактивних фреймворків.
   5.4 Результати порівняння реактивних фреймворків.

6. Дата видачі завдання «16» вересня 2024 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз наявної науково-технічної літератури	16.09-27.09.24	Виконав
2	Аналіз реактивних систем	30.09-11.10.24	Виконав
3	Підбір технологій реалізації	14.10-25.10.24	Виконав
4	Порівняння продуктивності реактивних фрейм-ворків	28.10-08.11.24	Виконав
5	Порівняння зручності і охоплення реактивних фреймворків	11.11-22.12.24	Виконав
6	Результати порівняння	25.11-29.11.24	Виконав
7	Оформлення роботи: вступ, висновки, реферат	02.12-06.12.24	Виконав
8	Розробка демонстраційних матеріалів	09.12-13.12.24	Виконав

Здобувач вищої освіти _____ Максим ПРИХОДЬКО
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

Керівник
кваліфікаційної роботи _____ Сергій ІЩЕРЯКОВ
(підпис) *(Ім'я, ПРІЗВИЩЕ*

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня магістра: 113 стор., 3 табл., 48 рис., 11 джерел.

Наукове завдання – порівняння реактивних фреймворків.

Мета роботи – виявити переваги і недоліки того чи іншого фреймворка і зрозуміти зону їх застосування.

Об'єкт дослідження – процес порівняння фреймворків та їх роботи.

Предмет дослідження – порівняння реактивних фреймворків та аналіз результатів.

Методи дослідження – розбір архітектури фреймворків, порівняння за різними ккритеріями.

Короткий зміст роботи: Оглянуто концепцію реактивних систем. Проведено порівняльний аналіз реактивних фреймворків.

Оглянуто проблеми багатопоточних сервісів. Аналіз проблем що вирішує реактивний маніфест. Розібрано архітектуру та як працюють реактивні фреймворки. Порівняв реактивні фреймворки та визначився з зоною їх використання.

Ключові слова: реактивна концепція, дослідження даних, порівняння, реактивні фреймворки

ABSTRACT

Text part of the master's qualification work: 113 pages, 48 pictures, 3 table, 11 sources.

Scientific task is to comparison of reactive frameworks.

Goal of the work is to identify the advantages and disadvantages of each framework and understand their application areas.

Object of research – process of comparing frameworks and their functionality.

Subject of research – comparison of reactive frameworks and analysis of results.

Research methods – analysis of framework architectures and comparison based on various criteria.

Summary of the work: The concept of reactive systems has been reviewed. A comparative analysis of reactive frameworks has been conducted. Problems of multithreaded services have been examined. Issues addressed by the Reactive Manifesto have been analyzed. The architecture and functionality of reactive frameworks have been explored. Reactive frameworks were compared, and their application areas were identified.

Keywords: reactive concept, data research, comparison, reactive framework

ЗМІСТ

ВСТУП

У сучасному світі програмування зростає потреба в ефективних рішеннях для обробки високих навантажень, асинхронних запитів і масштабованих систем. Реактивне програмування стало однією з найпопулярніших концепцій для розробки таких систем, адже воно дозволяє будувати програмні рішення, які здатні ефективно працювати з великими обсягами даних, одночасно знижуючи навантаження на ресурси серверів. У цьому контексті фреймворки, що підтримують реактивні моделі програмування, як-от Vert.x і Spring WebFlux, набули широкого застосування.

Ця робота присвячена порівняльному аналізу двох провідних технологій для реалізації реактивного програмування на платформі Java - Vert.x та Spring WebFlux. Vert.x є високо масштабованою бібліотекою, що орієнтована на подієву модель програмування, а Spring WebFlux є частиною відомого фреймворка Spring, який додає підтримку реактивного програмування для розробки веб-застосунків. Обидва фреймворки пропонують різні підходи до вирішення завдань асинхронності та паралельної обробки запитів, але мають різні архітектурні особливості, переваги та обмеження.

Метою цієї дипломної роботи є проведення порівняльного аналізу цих двох технологій, вивчення їх основних характеристик, а також визначення їхніх сильних і слабких сторін. Особлива увага буде приділена таким аспектам, як обробка запитів, інтеграція з іншими компонентами екосистеми Java, а також ефективність використання машинних ресурсів в умовах масштабованих систем. Це дозволить зробити висновки про доцільність вибору того чи іншого фреймворку для вирішення конкретних завдань у контексті реактивного програмування.

У результаті дослідження передбачається створення рекомендацій щодо застосування Vert.x і Spring WebFlux, зокрема в контексті їх використання в реальних проектних рішеннях.

1 ПРОБЛЕМИ ПРИ РОЗРОБЦІ СУЧАСНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ТА ШЛЯХИ ЇХ ВИРІШЕННЯ

1.1 Концепція багатопотоковості

1.1.1 Проблема одноядерних мікропроцесорів

15 листопада 1971 року компанія Intel випустила перший мікропроцесор. Цей пристрій зробив процесори комерційно доступними, а також змінив підхід до комп’ютерів, суттєво їх здешевивши, а також перетворив з спеціалізованих на універсальні пристрої.

Цей чип міг обробляти дані у дуже маленьких порціях - лише по 4 біти за раз. Та мав у своєму складі 2300 транзисторів. Для порівняння сучасні мікропроцесори мають сотні мільярдів транзисторів.

Мікропроцесор міг виконувати 60 тис. у середньому, та максимум до 93 тис. інструкцій у секунду. Для порівняння, один з перших повністю електронних комп'ютерів - американський ENIAC - виконував тільки 5000 тис. інструкцій у секунду, займав площу в 279 кв. м та важив 30 тонн.

Такий процесор був заточений під невелику кількість задач, але з часом навантаження, кількість і складність задач збільшувалась. Процесори покращували покращувались дуже швидко, за рахунок покращення внутрішньої архітектури і збільшенню кількості транзисторів, що давало приріст в тактовій частоті (кількість операцій, які процесор може виконати за одну секунду).

На жаль, на початку 2000-х розробники зіштовхнулись з обмеженнями при збільшенні тактової частоти, що ускладнило та уповільнило розвиток процесорів:

- Електронні витоки: Зменшення розмірів транзисторів призводить до збільшення витоків струму, що знижує енергоефективність і підвищує тепловиділення.
- Тепловиділення: З підвищенням тактової частоти процесор споживає більше енергії і генерує більше тепла. Це вимагає більш складних систем охолодження, а перевищення допустимого рівня може призвести до по-

шкодження процесора або зниження його ефективності через термічне тротлінг.

- Енергоспоживання: Споживання енергії процесором збільшується приблизно в квадраті від підвищення тактової частоти. Це значно підвищує витрати енергії, що є неефективним як для мобільних пристроїв, так і для стаціонарних комп’ютерів.
- Фізичні обмеження транзисторів: Зі збільшенням тактової частоти транзистори всередині процесора повинні перемикатися швидше. Однак є фізичні межі того, наскільки швидко можуть перемикатися транзистори без втрат електричної стабільності.

Саме через це інженери переглянули “філософію” розвитку процесорів і змінили напрям. Вони помітили, що деякі інструкції ніяк не впливають на виконання інших інструкцій на певних проміжках часу. Отже деякі інструкції процесору можна було виконувати паралельно. Саме тому з часом прийшли до ідеї додати ще одне ядро поряд таким чином дозволивши виконувати певні операції паралельно. Таким чином розробка процесорів розділилась на вирішення двох проблем: збільшення тактової частоти і збільшення кількості ядер на одному процесорі.

1.1.2 Паралелізм та багатопотоковість

Багатоядерні мікропроцесори дали змогу виконувати задачі паралельно. В розробці програмного забезпечення з’явилось нове поняття - паралелізм. Паралелізм - це здатність виконувати незалежні завдання програми в один і той же момент часу.

При виконані великої і складної задачі виникає проблема неоптимального використання можливостей центрального процесора. Зазвичай такі ситуації виникають, коли процесор очікує виконання операцій введення-виведення, транзакцій бази даних або запуску зовнішньої програми рис. 1.1.

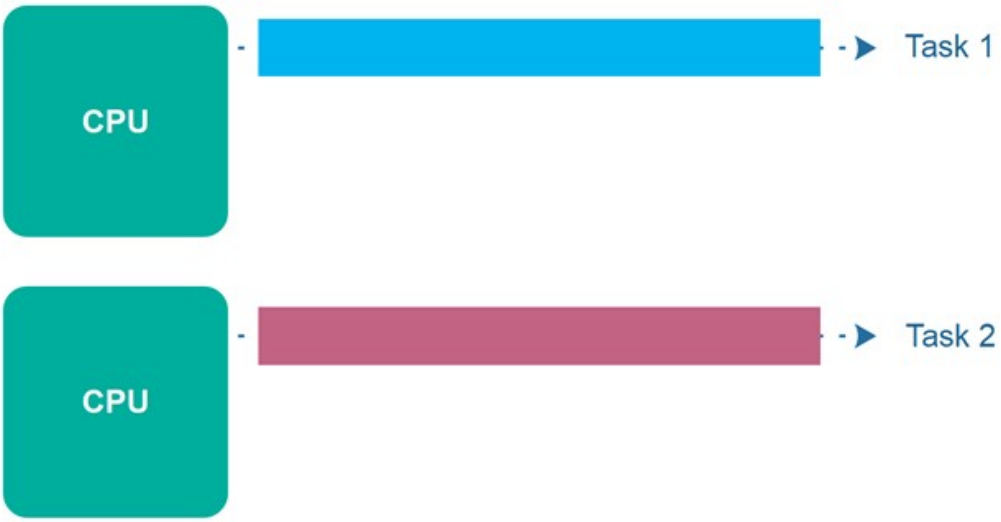


Рисунок 1.1 Приклад виконання задач паралельно

Саме тому була розроблена концепція багатопотоковості, основна мета якої - максимізувати ЦП шляхом мінімізації часу його простою рис. 1.2.

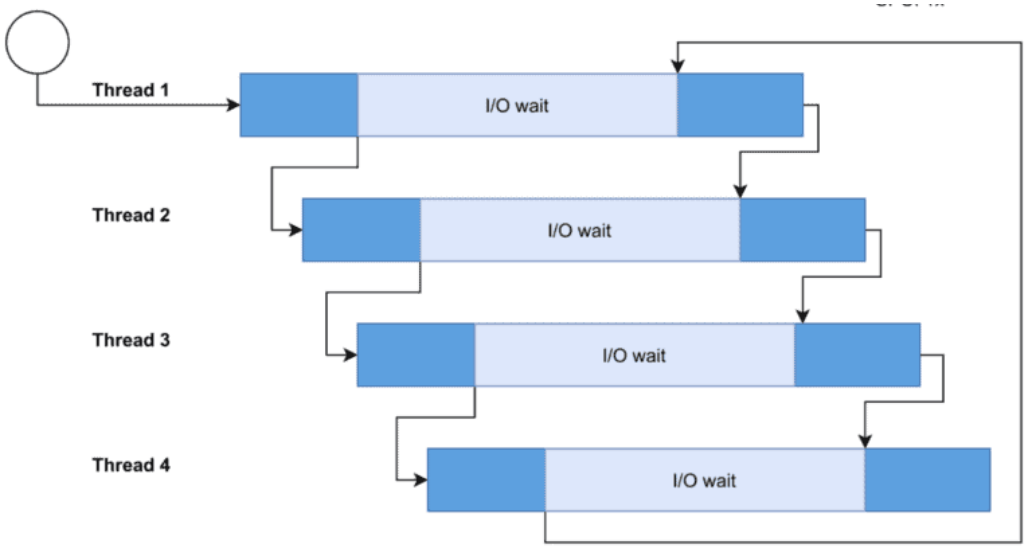


Рисунок 1.2 Приклад багатопотоковості

Потік (thread) – це одна з дій усередині процесу, що є найменшою одиницею обробки, виконання якої може бути призначено планувальником операційної системи.

Потік не є чимось фізичним, це є абстракція на рівні операційної системи. Одне ядро процесора може один процес, в одному процесі може виконуватись декілька потоків. Багатопотоковість фактично означає, що кілька завдань можуть виконуватися протягом періоду часу, що накладається. Одне із завдань може початися раніше, ніж буде виконано попереднє; однак вони не працюватимуть одночасно. Центральний процесор регулюватиме часові проміжки для кожного

завдання та відповідним чином перемикатиме контексти. Тому цю концепцію досить складно реалізувати і особливо налагодити.

Давайте розглянемо, як працюють багатопотоковість і паралелізм, на прикладі нижче. Як бачимо, є два ядра і два завдання. У паралельному підході кожне ядро виконує обидва завдання, перемикаючись між ними з часом. Навпаки, паралельний підхід не перемикається між завданнями, а виконує їх паралельно з часом рис. 1.3:

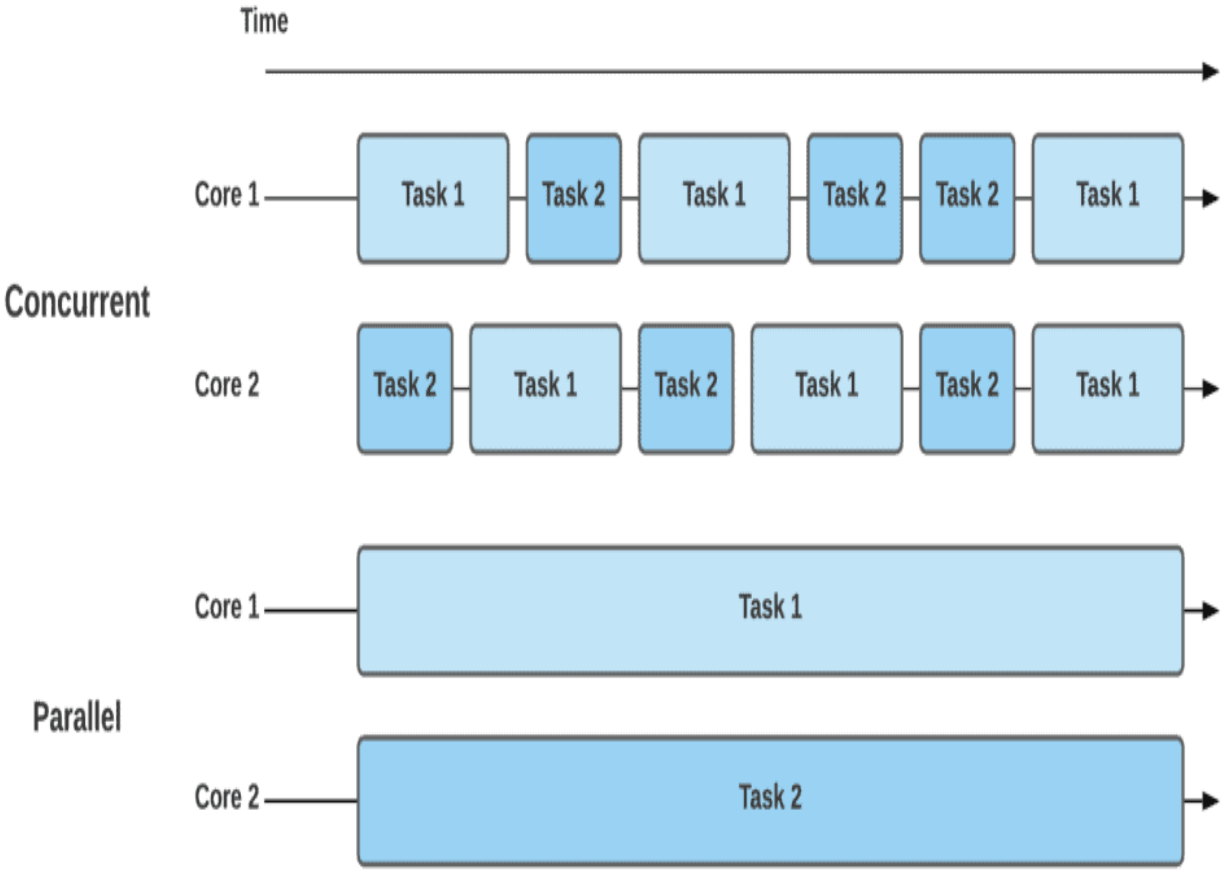


Рисунок 1.3 Приклад виконання задач конкурентно і паралельно

Цим простим прикладом одночасної обробки може бути будь-яка інтерактивна програма, наприклад текстовий редактор. У такій програмі можуть бути деякі операції вводу-виводу, які витрачають цикли ЦП. Коли ми зберігаємо файл або друкуємо його, користувач може одночасно вводити текст. Головний потік одночасно запускає багато потоків для введення, збереження та подібних дій. Вони можуть працювати в один і той же період часу; однак насправді вони не працюють паралельно.

1.2 Синхронність та асинхронність

1.2.1 Синхронність

У минулій частині я розібрав навіщо взагалі створювати декілька потоків і що при виконанні певних операцій потоки можуть простоювати чекаючи відповіді. Операції при яких потоки простоюють і чекають результат називаються блокуючі.

При розробці сучасного забезпечення ми постійно намагаємось придумати як найбільш оптимально отримати, обробити та відправити дані. На жаль, ми постійно стикаємось з блокуючими операціями на різних рівнях абстракції. На найнижчих рівнях це операції запису у файл, робота з мережевими сокетами, виконання системних команд. На рівні трохи вище ми стикаємось з такими проблемами, як запис і читання з бази даних, обробка великих масивів даних за допомогою інших потоків. Але якщо на рівні сервісу ці часто бувають не дуже помітні, то на архітектурному рівні блокуючі операціїї можуть коштувати значно дорожче. Наприклад в нас є система, де один потік відповідає одному вхідному запиту:

1) Сервіс А отримує запит.
2) Сервіс А обробляє запит.
3) В ході обробки запиту Сервіс А потребує додаткової обробки даних в Сервісі В, тому робить запит на Сервіс В і чекає відповідь.
4) Сервіс В оброблює дані протягом 10 хв і повертає на Сервіс А.
5) Сервіс А отримує відповідь і закінчує обробку даних.

А уявімо що такий запит не один, а сотні тисяч за хвилину рис. 1.4.

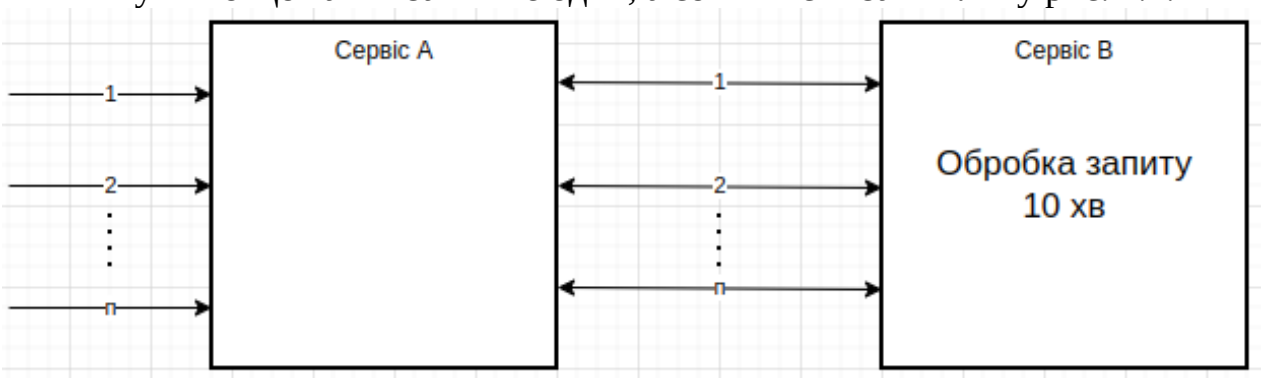


Рисунок 1.4 Приклад синхронної роботи

Тобто Сервісу А потрібно створити сотні тисяч потоків, що будуть простоювати по 10 хв і чекати на відповідь від сервіса В. А якщо таких сервісів у нас декілька? Очевидно що така система буде просто недієздатною.

1.2.2 Асинхронне рішення

На щастя, є рішення проблеми синхронності - коли один потік заблокований і чекає на відповідь. Розробники подумавши, вирішили, що було б добре якби потік в Сервісі А замість того, щоб чекати відповідь від Сервісу Б знав, що йому потрібно прийти і отримати відповідь через 10 хв, а цей час він би оброблював інші запити. Таким чином нам би не довелось би створювати сотні тисяч потоків. Такий підхід називається асинхронним.

Отже асинхронність це коли виконавець отримавши задачу оброблює її до того часу поки вона не буде заблокованою. Коли операція є блокуючою, виконавець не очікуючи відповіді починає робити інші задачі, а коли відповідь сформована виконавець повертається за нею і оброблює далі, поки та знову не буде заблокована.

Повернемось до нашої системи і переглянемо алгоритм, додавши асинхронність рис. 1.5:

1. Сервіс А отримує Запит 1.
2. Потік в Сервісі А обробляє Запит 1.
3. В ході обробки запиту Сервіс А потребує додаткової обробки даних в Сервісі В, тому робить запит на Сервіс В.
4. Потік не чекаючи відповіді оброблює Запит 2.
5. Обробивши Запит 2, Потік йде за результатом з Сервісу В для Запиту 1.
6. Потік закінчує обробку Запиту 1.
7. Потік йде за результатом з Сервісу В для Запиту 2.
8. Потік закінчує обробку Запиту 2.

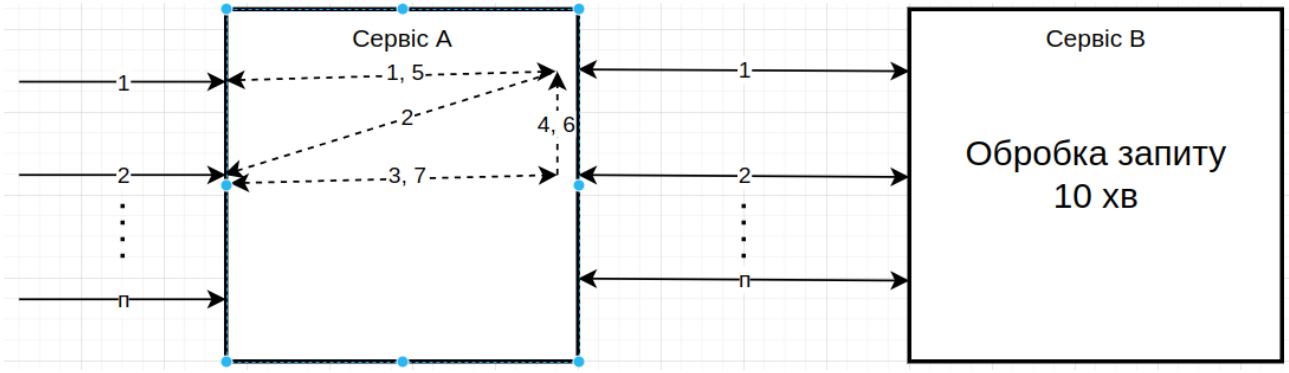


Рисунок 1.5 Приклад асинхронної роботи

Насправді незважаючи на певні переваги асинхронного програмування, очевидно, що воно має і певні недоліки перед синхронним.

Основними недоліками асинхронного програмування є:

- Підвищена складність коду : його може бути складніше написати та зрозуміти.
- Важче налагодити : проблеми можуть бути складнішими для виявлення.
- Більш складна обробка помилок : вимагає особливої обережності для належного керування помилками.
- Проблеми синхронізації : кілька операцій можуть конфліктувати під час доступу до спільних ресурсів.
- Додаткові запити: сервіс спочатку відправляє запит, а потім запитує про результат.

Перевагами синхронності є:

- Простота: синхронне програмування набагато легше ніж асинхронне, адже нам не потрібно думати про колбеки, одночасний доступ до ресурсів.
- Немає зайвих запитів.

Зрештою, вибір зводиться до операційних залежностей. Ви хочете, щоб початок операції залежав від завершення іншої операції, чи ви хочете, щоб вона запускалася незалежно?

Асинхронна архітектура не блокує, тому виконання одного завдання не залежить від іншого. Завдання можуть виконуватися одночасно.

Синхронна архітектура блокує, тому виконання кожної операції залежить від виконання попередньої. Кожне завдання вимагає відповіді перед переходом до наступної ітерації.

1.2.3 Варіанти використання асинхронності і синхронності

Програмування змушує наш цифровий світ працювати, але без правильного поєднання програм і операцій виникне хаос і поганий досвід користувачів. Важливо розуміти, коли використовувати кожен тип програмування.

Асинхронне програмування має вирішальне значення для програмування незалежних завдань. Наприклад, асинхронні програми ідеально підходять для проектів розробки з великою кількістю ітерацій. Асинхронне програмування забезпечить просування розробки, тому що кроки не повинні слідувати фіксованій послідовності.

Чуйний інтерфейс користувача є чудовим варіантом використання для асинхронного планування. Візьмемо, наприклад, додаток для покупок. Коли користувач відкриває своє замовлення, розмір шрифту має збільшуватися. Замість того, щоб спочатку чекати завантаження історії та оновлення розміру шрифту, асинхронне програмування може змусити обидві дії виконуватися одночасно.

Асинхронне програмування відносно складне. Це може надто ускладнити речі та ускладнити читання коду. З іншого боку, синхронне програмування є досить простим; його код легше писати і не вимагає відстеження та вимірювання потоків процесів (як це робить асинхронний).

Оскільки завдання залежать одне від одного, ви повинні знати, чи можуть вони виконуватися незалежно, не заважаючи одне одному.

Синхронне програмування також може підійти для програми для покупок, орієнтованої на клієнтів . Користувачі хочуть купувати всі свої товари разом, а не окремо під час оформлення замовлення онлайн. Замість виконання замовлення кожного разу, коли користувач додає щось у свій кошик, синхронне програмування гарантує, що спосіб оплати та пункт призначення для всіх товарів вибираються одночасно.

1.2.4 Моделі паралелізму

Знаючи що таке синхронне і асинхронне, я б ще хотів зачепити тему паралелізму, а саме моделі паралелізму. Варто сказати що як і потоки так і процеси можуть мати спільний стан рис. 1.6 або окремий рис. 1.7.

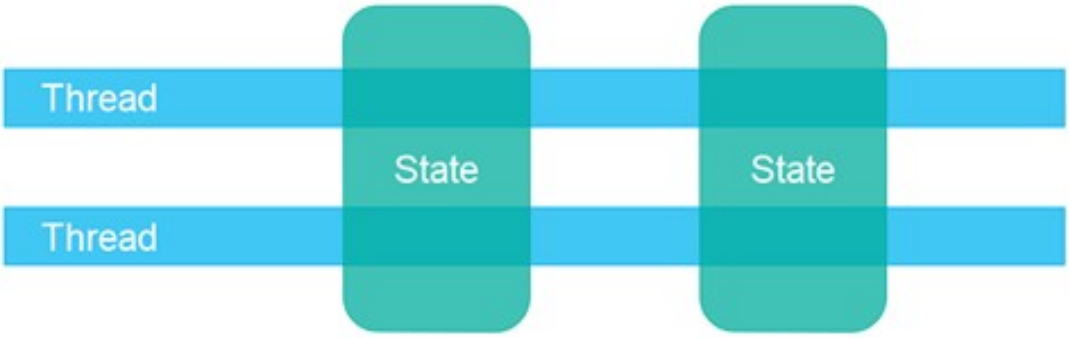


Рисунок 1.6 Приклад спільного стану

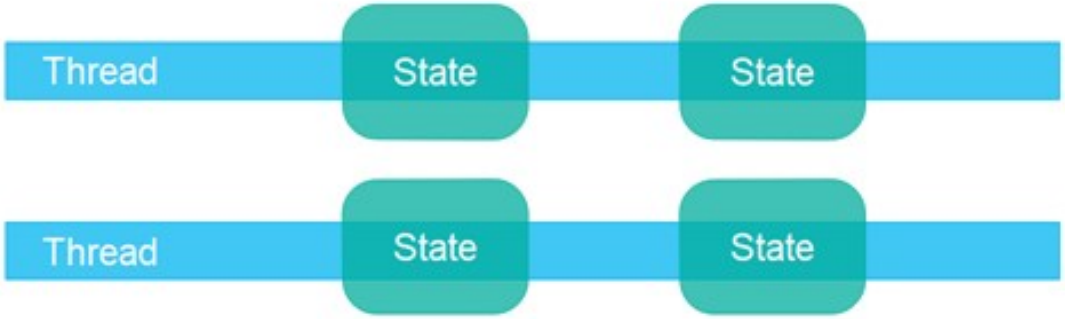


Рисунок 1.7 Приклад окремого стану на потік

Коли стан спільний, то часто можуть виникати проблеми з доступом, оновленням, видаленням даних. Тому варто при проектуванні систем по можливості варто уникати спільного стану, але інколи це не можливо. Це буде одною з важливих характеристик при описі моделей паралельності. Отже, я б виділив дві основні моделі паралелізму.

Перша модель, яку ще називають модель паралельних робітників, розподіляє завдання серед потоків рис. 1.8.

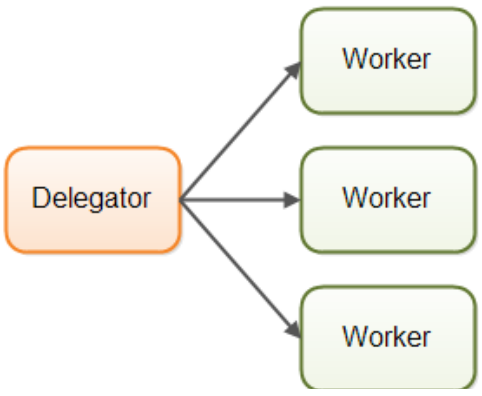


Рисунок 1.8 Приклад моделі паралельних робітників

Ця модель є простішою у розумінні і дуже популярною при розробці, напевно в кожній сучасній системі, в якій є хоча б мінімальні потреби до

продуктивності використовується ця модель. Також важливою перевагою є масштабування, якщо нам треба зробити роботу швидше, то ми можемо додати ще одного воркера.

Але ця модель має і недоліки:

- Спільний стан: основною проблемою цієї моделі є спільний стан. В реальних системах, рано чи пізно ми стикаємось зі спільним станом. Для того, щоб підтримувати роботу з консистентними даними, потоки або процеси часто блокуються. Наприклад, коли ми додаємо нові дані в якусь таблицю бд одним потоком - ця таблиця може бути заблокованою для інших потоків, до тих пір поки запис не закінчиться.
- Збільшення робітників не означає пропорційному зростанню продуктивності. Це пов’язано з фізичними обмеженнями. Чим більше у нас розпаралелена задача, тим частіше процесору потрібно перемикатись між потоками. Перемикання між потоками не є безкоштовними, бо займають певний час і ресурси. Також залежить від проценту послідовних обчислень в сервісі. Відповідно до закону Амдала, прискорення виконання програми за рахунок розпаралелювання її інструкцій на безлічі обчислювачів обмежено часом, необхідним виконання її послідовних інструкцій
- Недетермінованість порядку виконання. Це означає, що порядок виконання розпаралелених задач не є чітко визначеним.

Друга модель, яку ще називають конвеєрною моделлю, змушує процеси передавати дані послідовно - у вигляді ланцюга рис. 1.9.

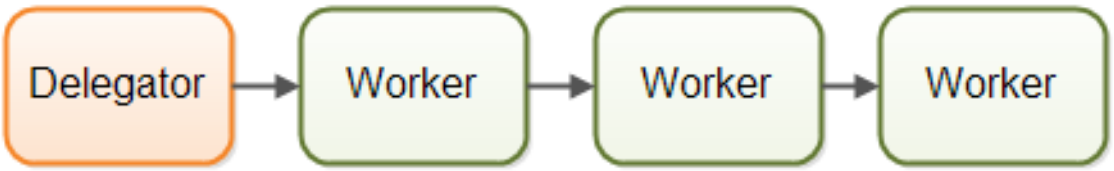


Рисунок 1.9 Приклад конвеєрної моделі

Модель працює таким чином, що робітник виконавши завдання передає дані іншому робітнику, а сам оброблює інші дані в цей час. Отже у процесів немає спільного стану і немає необхідності чекати заблоковані ресурси. Також вирішується проблема недетермінованості. Недоліками є те що така модель є складнішою

в реалізації і в розумінні. Забігаючи наперед, на такій моделі будується реактивне програмування.

Насправді в реальних системах використовують комбінований підхід з конвеєрної моделі і паралельних робітників рис. 1.10.

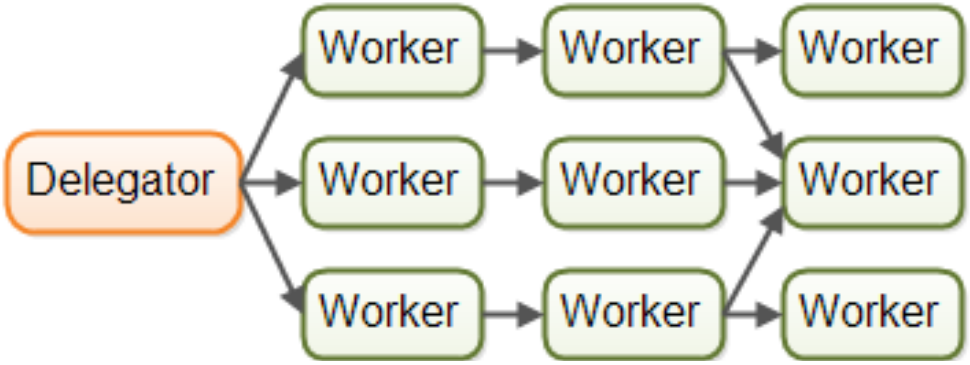


Рисунок 1.10 Приклад комбінованої моделі

1.3 Концепція реактивності

1.3.1 Що таке реактивність?

Сучасні користувачі систем або системи які використовуть сторонні сервіси звикли що система має завжди бути доступною, швидко відповідати і справлятись з високим навантаженням. На жаль, цього не так просто досягнути, адже існує безліч проблем з якими зіштовхуються розробники при проектуванні і розробки системи: нестабільність мережі, велика кількість корнер кейсів, різне навантаження в різні проміжки часу, тощо. Отже, сучасні системи потребують сучасного переосмислення і підходів.

При розробці хочемо, щоб системи були чуйними, стійкими, еластичними та керованими повідомленнями. Системи що відповідають таким принципам з часом назвали реактивними системами.

Системи, побудовані як реактивні системи, є більш гнучкими, слабко пов’язаними та масштабованими. Це полегшує їх розвиток і піддається змінам. Вони значно терпиміші до невдач Реактивні системи дуже швидко реагують, надаючи користувачам ефективний інтерактивний зворотний зв’язок.

Але що значить таке реактивний? Як на мене це реактивний - це дуже перевантажене слово, саме тому в мене виникло кпа проблем коли я намагався розібратись з цією темою на концептуальному рівні. Якщо в пошукову систему вбити

слово реактивний, то видає купу різних варіантів: реактивні системи, реактивне програмування, реактивні розширення, реактивний обмін повідомленнями, тощо. Одразу здається що ці результати взагалі не мають нічого спільного рис. 1.11.

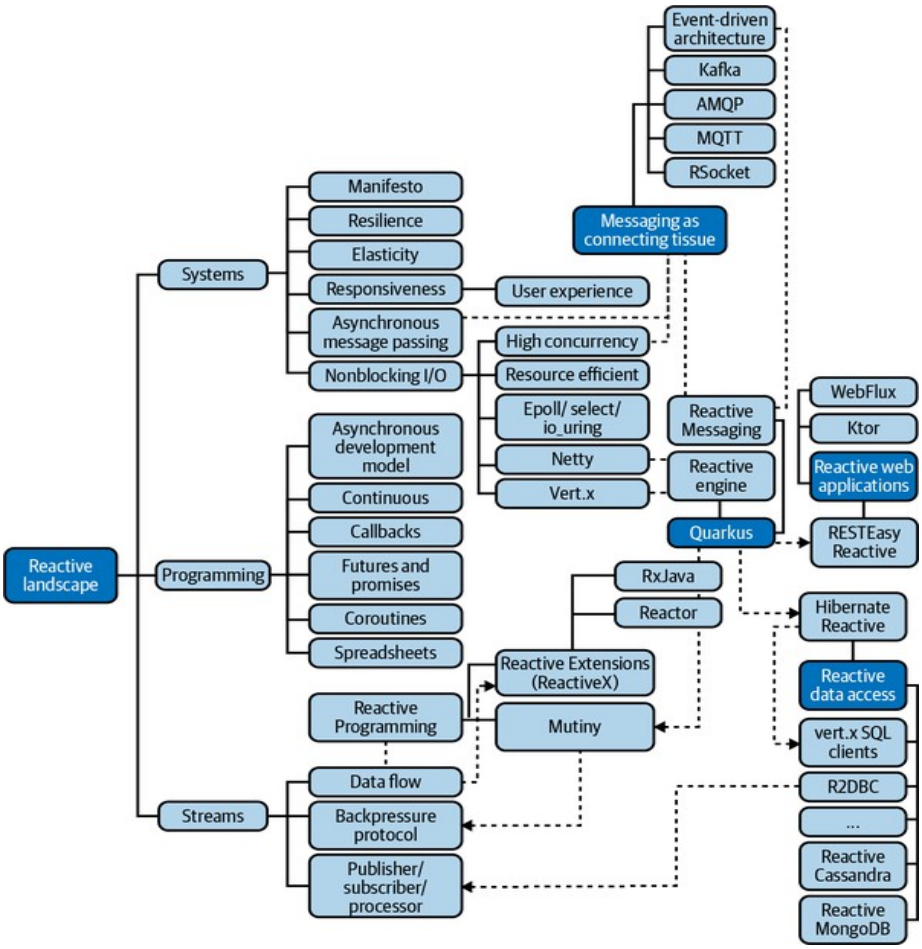


Рисунок 1.11 Реактивні технології

Для того щоб розібратись в загальному що таке реактивний, я скористався Оксфордським словником англійської мови, результат:

1. Показ реакції на подразник.
2. Діяти у відповідь на ситуацію, а не створювати або контролювати її.
3. Схильність до хімічної реакції.
4. (Фізіологія) Виявлення імунної відповіді на специфічний антиген.
5. (Про хворобу або хворобу) Викликаний реакцією на щось.
6. (Фізика) Що стосується реактивного опору.

Серед цих визначень два доречні в нашому контексті. Перше визначення, що показує реакцію на подразник , відноситься до певного роду реакції. Бути реактивним означає реагувати на подразники, якими б вони не були. Це в свою чергу говорить, що бути реактивним також означає зіткнутися з несподіваними та

неконтрольованими ситуаціями. Хоча ці визначення цікаві, вони не стосуються програмного забезпечення. Але ми можемо взяти до уваги ці визначення, щоб створити нове, специфічне для програмного забезпечення. Реактивна програма - це програма що реагує на подразники, такі як події користувача, запити та збої.

1.3.2 Реактивний маніфест

Реактивність - це концепція. Реактивність немає чітко визначених рамок і контрактів. Це лише набір правил, паттернів і принципів, якщо дотримуючись яких ми досягнемо успіху при розробці програмного забезпечення. Все це було описано в одному документі, яке називається “Реактивний маніфест”. Друга і найактуальніша версія була опублікована 16 вересня 2014 р і досі доповнюється. Це є опенсорс рішенням, тобто кожний бажаючий може подати запит на редагування. Маніфест можна знайти за посиланням: www.reactivemanifesto.org.

Реактивна система керується подіями, така система називається Event-Driven. Система повинна мати можливість реагувати на зміну робочого навантаження - Scalable, отже має бути масштабована. Крім того, система також повинна мати можливість реагувати на збої - Resilient. І нарешті, і, мабуть, найважливіше, система повинна мати можливість реагувати на своїх користувачів, має бути чуйною - Responsive. Бути керованим подіями – це властивість, яка забезпечує властивості масштабованості та стійкості, і всі три властивості разом дозволяють системам реагувати на користувачів рис. 1.12.

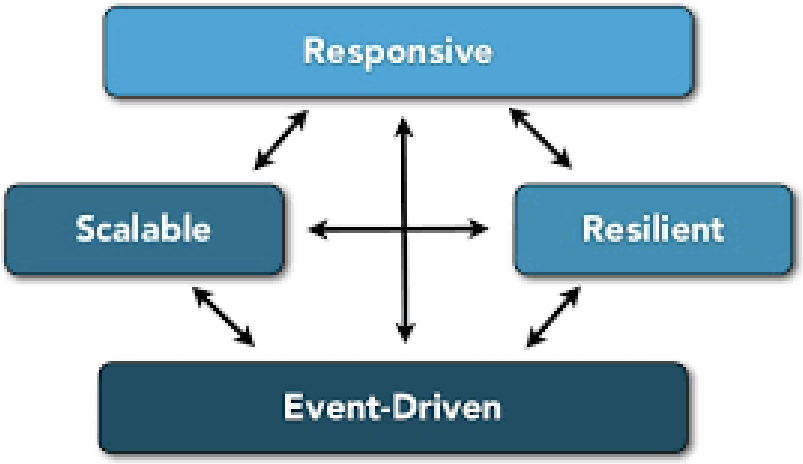


Рисунок 1.12 Концепція реактивності

Керування подіями. Традиційно паралельні програмні системи складалися з кількох потоків, які спілкувалися в спільному та синхронізованому стані, що призводило до сильної зв’язності. Керовані подіями системи, навпаки, складаються з слабо обробників подій. Такі системи працюють асинхронно, не спричиняючи жодного блокування. Оскільки блокування немає, це означає, що ресурси можна використовувати набагато ефективніше.

Масштабованість. Програма є масштабованою, якщо її можна розширювати відповідно до навантаження, що надходить до неї. Зазвичай ми розрізняємо два напрямки масштабування: горизонтальне і вертикальне. Вертикальне масштабування відбувається за рахунок збільшення ресурсів на одній обчислювальній машині. Горизонтальне масштабування відбувається за рахунок додавання обчислювальних машин. Важливою характеристикою для масштабованості є мінімізація стану. Тобто компонент не має зберігати якусь інформацію у себе в пам’яті машини, а відправляти, наприклад, в бд, щоб інші обчислювальні машини де запущена та сама програма також мали доступ до цієї фінформації. Крім того, важливою для масштабування є властивість прозорості розташування. Це означає, що не має значення, де розташоване місце розташування, це може бути на тому самому комп’ютері, що й клієнт, або на іншому комп’ютері в мережі.

Відмовостійкість: програма є стійкою, якщо вона може швидко відновлюватися після збоїв. Такими збоями можуть бути збої програмного забезпечення, як-от створення винятку чи збій апаратного забезпечення: вихід з ладу комп’ютера, збої з’єднання. Інтернет-з’єднання, як правило, не можна додати, існує думка, що це має бути частиною дизайну з самого початку. Стійкість досягається шляхом реплікації, слабкими зв’язками, інкапсуляцією стану та делегуванням.

Чутливість: програма є чутливою, якщо вона забезпечує взаємодію в реальному часі зі своїми користувачами навіть під навантаженням і за наявності збоїв. Чутливі програми можуть бути створені, якщо дотримуватись керованості подіями, масштабованості та відмовостійкост. Але все одно потрібно приділяти особливу увагу дизайну алгоритмів системи і іншим важливим деталям.

Важливим ньоансом є те що система може бути побудована з багатьох компонентів і різних абстрактних рівней. Саме тому важливо що б концепція маніфесту дотримувалась на всіх рівнях системи і всіма компонентаами.

1.3.3 Принципи реактивної системи

Які критерії ми використовуємо, щоб відрізнити «реактивну програму» від тієї, яка не є реактивною? Reactive вирішує проблеми безпосередньо у своїх абстракціях, моделях програмування, протоколах, схемах взаємодії, обробці помилок та інших подібних областях дизайну та архітектури. У той час як багато розподілених систем, на жаль, розглядають проблеми як запізнілу думку - як операційні проблеми або проблеми з інфраструктурою, які вирішуються за допомогою дедалі складніших технологічних стеків, оболонок, обхідних шляхів і невдалих абстракцій із витоком. Реактивний маніфест описує базові принципи реалізації розподілених систем, які роблять програму реактивною.

Першою концепцією є чутливість або вміння системи відповідати за визначений час. Зазвичай між сервісом і клієнтами є певні контракти, які ще мають назву sla(service-level agreement), де зазвичай описується і час за який наша система має відповісти, якщо система не відповідає у визначений час клієнт закриває конекшин. Більш наглядний приклад є з користувачем який відкрив якийсь сайт почекав деякий час і закрив, тому що не отримав відповіді. Для користувачів не має значення, що система правильна, якщо вона не може забезпечити свою функціональність у межах їх часових обмежень.

Бути чутливим означає не лише низьку затримку та швидкий час відповіді, а й керування змінами. Реакція на зміни має бути повідомлена користувачам компонента, будь то люди чи програми, щоб відповіді на запити можна було інтерпретувати в правильному контексті. Реактивна програма, якщо не може обробити запит або не встигає цього зробити, має відправити повідомлення, яке чітко опише контекст проблеми, якщо така є.

Наступним принципом є прийняття невизначенності. Сучасні системи є досить складними, саме тому дані інколи можуть приходити в зовсім неочікуваному форматі або не приходити вазагалі. Отже, метою при побудові системи має бути

максимальна узгодженність. Використовуйте стандартизовані протоколи, контракти, устні узгодженості, тощо, щоб зменшити кількість виняткових ситуацій.

Прийняття невдач. Реактивні програми розглядають збій як очікувану умову, яка зрештою відбудеться. Таким чином, збій повинен бути явно представлений і оброблений на певному рівні, наприклад, в інфраструктурі, компонентом супервізора або в самому компоненті (за допомогою внутрішнього резервування). На запити слід відповідати, коли це можливо, навіть у випадку збою, навіть якщо автономність компонента вже гарантує, що збій залишається у якомога меншій області функції програми. Відокремлення в просторі також дозволяє зберігати відмову в межах визначених зон відмов, тоді як відокремлення в часі дозволяє іншим компонентам надійно виявляти та обробляти несправності, навіть якщо про них неможливо явно повідомити.

Автономія є ще одним принципом. Варто намгатися проектувати систему, щоб компоненти якомога менше залежали один від одного. Це потрібно для того щоб один компонент міг продовжувати роботу, навіть тоджі коли інший компонент не працює.

Принцип консистентності каже про те що ми маємо гарантувати правильність даних і зв’язків між цими даними. Важливим моментом є те що не всі дані мають бути консистентними, а тільки ті що дійсно потрібно. Це спричинено тим що для підтримки консистентності даних витрачаються ресурси.

Принцип розділення за часом говорить про те, що варто уникати блокуючих операції і надавати перевагу асинхронним діям. Потокам не потрібно витрачати час простоюючи і чекаючи відповіді від інших сервісів, варто зайнятись чимось іншим.

Якщо принцип розділення за часом допомагає покращити продуктивність програми запущеної на одній обчислювальній машині, то принцип розділення простору говорить про те що варто мати декілька екземлярів запущеної програми на декількох обчислювальних машинах. Це покращує відмовостійкість та розпаралелює роботу системи.

Останнім описаним принципом у реактивному маніфесті є контроль динамі-

чності. Цей принцип говорить про те що система має завжди буди готова до будь-яких змін, наприклад збільшення або зменшення навантаження. Реактивні системи мають бути пластичні.

1.3.4 Паттерни для побудови реактивних систем

Разом з принципами в маніфесті описані і паттерни засновані на цих принципах, використовуючи можна досягнути реактивності системи. Паттерни не є чимось фіксованим і точно не потрібно використати всі паттерни в своїй системи. Це лише інструменти, які допомагають вирішити проблеми, які зазвичай виникають при розробці реактивних систем.

Першим описаним паттерном є розділення стану. Деякі задачі можна розпаралелити, тобто виконувати на ядрах або взагалі на різних машинах. Якщо наш компонент має певну кількість реплік, то варто щоб компонент не мав стану. Тобто не зберігав якусь інформацію у пам’яті комп’ютера, а записував, наприклад до бази даних або передавав в брокер повідомлень. Цей підхід значно полегшить розпаралелювання. Адже якщо сервіс не має стану, тоді немає різниці яка з реплік обробить запит, результат буде однаковим. Для того щоб компоненти не мали стану, варто розділити систему на невеликі за розміром і функціоналом компоненти.

Наступним паттерном є передача фактів. Ми не можемо передавати стан між розподіленими системами, тому що стан то не є щось стале. Один потік може передати стан, а інший цей стан змінити. Тому варто передавати або реагувати не нп стан а на подію, тобто коли дія вже виконалась, або інакше це називають фактом. Факт є незмінним і представляє те, що вже сталося колись у минулому, те, що не можна змінити або відмінити. Це стабільна цінність, про яку можна міркувати та довіряти нескінченно довго. Зрештою, ми не можемо змінити минуле, навіть якщо іноді хотілося б. Знання є кумулятивним і виникає або шляхом отримання нових фактів, або шляхом виведення нових фактів з існуючих фактів. Анулювання існуючих знань здійснюється шляхом додавання в систему нових фактів, які спростовують існуючі факти. Факти ніколи не видаляються, лише робляться нерелевантними для поточних знань.

Ще одним паттерном є ізоляція змін. Він говорить про те що деякі частини не можна відділити від стану. Тому треба спочатку максимально відділити змінюване від незмінюваного. Ізоляція логіки, яка змінює стан (наприклад, оновлює базу даних або змінює внутрішній стан), від логіки, яка тільки читає дані чи виконує інші дії. Це робиться для підвищення стійкості та передбачуваності реактивних систем.

Наступний паттерн це організація потоків даних. Цей паттерн пояснює переваги використання потоків. Реактивні системи, як я вже писав раніше, мають бути подієво наслідковими, тобто реагувати на події. В такому випадку зручно мати поток з подіями і можливість підписуватись на ці потоки. Підписники можуть самі забирати дані і таким чином контролювати навантаження.

Ще одним паттерном є локалізація стану. Цей паттерн зосереджується на ідеї, що кожен компонент у системі повинен зберігати свій стан локально, щоб мінімізувати залежності між компонентами. Це допомагає зменшити складність і зробити систему більш стійкою та масштабованою. Замість того, щоб зберігати спільний стан у центральному сховищі, кожен компонент має зберігати дані, необхідні лише для нього. Локалізований стан полегшує розуміння та управління компонентами. Компоненти чітко взаємодіють через обмін повідомленнями чи потоками даних, а не безпосередньо змінюють стан один одного.

Останній паттерн звучить, як спостереження за комунікацією. Це про те що розробник має в будь-який момент, розуміти, що відбувається з його системою і яка динаміка. Це допомагає зробити аналітику або виявляти проблеми на ранніх етапах і не допускати критичних ситуацій. Загалом спостережуваність складається з показників додатків, мережевих показників, звітів про працездатність, журналів і трасувань.

1.4 Стандарт Reactive Streams

На початку 2000-х коли почали будуватись розподілені системи, почало ставати актуальним питання оптимальної передачі даних від сервера до клієнта і назад.

На початку трафіку було досить мало і тому певні недоліки передачі даних не дуже були помітні. Наприклад, з самого початку був часто використовуваний підхід, коли дані запрошувались клієнтом у сервера, сервер ітерувався по набору даних і відправляв дані по одному. Такий підхід мав певні переваги: гарно контрольований трафік, якщо клієнт не міг обробити певні запити, то він міг в будь-який момент повідомити сервер. Але очевидно такий підхід виглядає дуже невдалим, адже клієнт може мати блокуючу архітектуру і тому на кожен реквест створювався окремий потік. Але найголовніше те, що передача елементів по мережі має певні затримки. Якщо мережа буде заслабкою, а кількість даних велика, то час виконання певної роботи буде суттєвим.

На зміну цього підходу вирішили відправляти дані не поодному, а одразу цілу колекцію. Таким чином проблема з мережевою затримкою стане менш значною. Але з’являється інша проблема — маленька пропускна здатність мережі. Наприклад мережа може пропустити 20 елементів, а в нас є 30. Очевидним рішенням є розбиття колекції на декілька менших. Якщо таких колекцій буде багато, то ми повернемось до попередньої проблеми.

Наступним лімітом є пам’ять комп’ютера. Що буде якщо її недостатньо для обробки одразу всієї колекції даних? Якщо сервер відправить на наш клієнт завеликий об’єм даних, то ми отримаємо ексепшин - out of memory.

Очевидним рішенням є навчити наш клієнт казати скільки даних він може прийняти. Такий процес називається батчинг, коли клієнт опрацьовує дані порціями.

Тепер наш підхід виглядає більш вдали, але не все ще має недоліки. Наприклад, що якщо кількість клієнтів буде значно більшою? Вони будуть постійно запитувати сервер, щоб отримати дані. Виглядає так, що потрібен кардинально інший підхід.

Придумали інший підхід, а що якщо замість того, щоб клієнт запитував сервер. Сервер сам буде передавати дані. Отже вирішили змінити підхід з пул на пуш. А для роботи багатьох клієнтів з сервером використовують паттерн спостерігач, де застосована модель паблішера і отримувача. Власне, якщо є новий сервіс, потрібно просто підписати його на розсилку повідомлень від сервера.

Такий підхід є асинхронний і вже більш схожий на принципи, що описанні в реактивному маніфесті. Тому такий підхід стали називати реактивним.

Отже тепер ми можемо встановити конекшин від сервера до клієнта. Сервер буде пушити дані по одному елементу в одному з'єднанні а клієнт почне їх асинхронно оброблювати. Коли всі елементи передані, сервер закриває конекшин. Таким чином відпадає проблема з пропускною здатністю мережі і непотрібних запитів від клієнта до сервера.

Також при використанні паттерна спостерігач і пуш підходу постає нова проблема. Сервер зазвичай набагато потужніший за клієнта. Тому може виникнути ситуація, коли клієнт відправляє занадто багато даних які клієнт не в змозі обробити. Або просто може статись ситуація, коли клієнт має збій. Тому клієнт потребує механізму відправки сповіщення до сервера про проблеми, щоб сервер припинив відправляти дані.

Отже, вирішили об'єднати переваги першого підходу, який реалізовував, щось схоже на паттерн ітератор, де всі елементи відправлялись по одному і клієнт мав змогу гарного контролю над відправкою елементів і відправки попередження до клієнта, якщо стався збій. І паттерна спостерігач, який дозволяє асинхронне спілкування, але в якому немає можливості для контролю відправки елементів. В результаті цього ми отримали підхід, який назвали реактивні стріми. Підхід включає в себе: підтримку колбеків — можливість клієнта повідомити сервер скільки даних він здатний обробити, асинхронне спілкування, можливість підписки на сервер нових клієнтів, повідомлення сервера про проблеми на клієнті, повідомлення сервера про закінчення набору даних.

Наступною проблемою стало те, що на різних мовах програмування це було розроблено по різному і це небуло зручно. Розробники потребували стандартизації підходу реактивних стрімів і загальний API. Отже, з'явився стандат реактивних стрімів, який є опенсорсним(відкритим) рішенням і який включає набір правил для розробки сервісів за допомогою реактивних стрімів і загальний API для деяких основних мов програмування. Напевно, найкраще цей підхід прижився в Java на основі, якого було розробленно внутрішні бібліотеки Java і додаткові інструменти - реактивні фреймворки рис. 1.13.

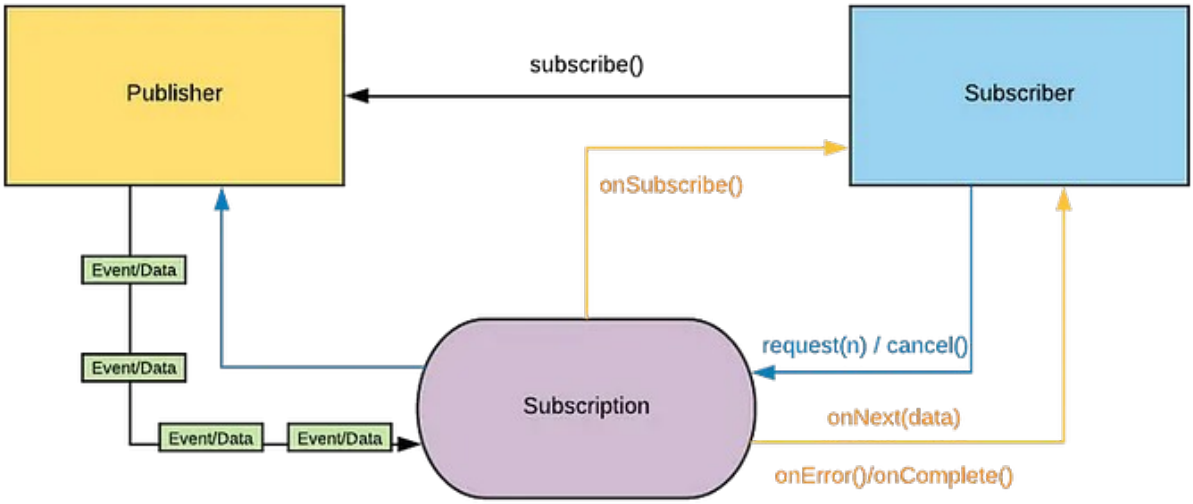


Рисунок 1.13 Приклад передачі і обробки даних в реактивних стрімах

API складається з таких компонентів, які повинні надаватися реалізаціями Reactive Stream:

- Видавець
- Підписник
- Підписка
- Процесор

Видавець є постачальником потенційно необмеженої кількості елементів, публікуючи їх відповідно до запиту, отриманого від підписників.

У відповідь на виклик Publisher.subscribe(Subscriber) можливі послідовності викликів для методів Subscriber надаються таким протоколом:

onSubscribe onNext* (onError | onComplete)?

Це означає, що onSubscribe завжди сигналізується, після чого йде, можливо, необмежена кількість onNext сигналів (за запитом Subscriber), за якими слідує onError сигнал, якщо сталася помилка, або сигнал, onComplete коли більше немає доступних елементів - і все це доти, доки Subscription не скасовано.

Reactive Streams пропонує неблокуючий та асинхронний підхід для обробки потоків даних. Вибір між асинхронною чи синхронною обробкою залежить від потреб конкретної реалізації, і обидва підходи можуть співіснувати в межах одного потоку.

Асинхронність часто вводиться шляхом використання черг або планувальників (schedulers). Реактивний потік може мати кілька асинхронних меж, де обробка переходить від одного ресурсу до іншого рис. 1.14.

```
nioSelectorThreadOrigin | map(f) | filter(p) | consumeTo(toNioSelectorOutput)
--------- R1 ---------- | - R2 - | -- R3 --- | ---------- R4 ---------------
```

Рисунок 1.14 Приклад використання операторів реактивних стрімів

Тут кожна операція (map(), filter(), consumeTo()) виконується на окремому ресурсі (R1, R2, R3, R4), а черги забезпечують передачу даних між ними.

Приклади реалізацій:

- Повністю асинхронна обробка: Кожен етап виконується на окремому ресурсі, використовуючи черги для передачі даних.
- Частково синхронна обробка: Деякі операції виконуються в тому ж потоці, де дані генеруються. Наприклад: map() і filter() виконуються синхронно в одному потоці (R1), а `consumeTo` — асинхронно.
- Злиття обробки до одного ресурсу: усі операції виконуються асинхронно, але на єдиному ресурсі.

Можна об’єднати важливі аспекти реактивних стрімів:

- Backpreassure: Видавець повинен дотримуватись обмежень запиту, навіть якщо джерело даних є нерегульованим. У таких випадках видавець може: буферизувати дані або відкидати надлишкові елементи.
- Ефективність: Підписник може сигналізувати про більший попит (`request(n)`), щоб уникнути частих підтверджень. Це знижує накладні витрати.
- Гнучкість: Підписник може сигналізувати про запит на нові елементи в будь-який час, підтримуючи буфер заповненим і мінімізуючи затримки.

Реактивні стріми надають механізми для балансування між асинхронністю, продуктивністю та ефективним використанням ресурсів.

1.5 Модель Actors

У сучасному світі все відбувається асинхронно. Власне тому, чому б розробникам також не відштовхуватись від цього? Можна будувати розробку програмного забезпечення на основі того, що всі від найбільшого до найменшого елемента будуть працювати асинхронно. Саме з цієї ідеї і виникла модель акторів.

Модель актора в інформатиці - це математична модель паралельних обчислень, яка розглядає актора як базовий будівельний блок паралельних обчислень. Модель складається з:

- Повідомлення - це дані які передаються від актора до актора.
- Поштові скриньки - це адреса і буфер, де збираються повідомлення перш ніж актор на них відреагує.
- Актори - елементи, що реагують на повідомлення.

Різниця між використанням моделі акторів і простим використанням акторів полягає в тому, як ви ставитеся до цих акторів. Якщо ви використовуєте цих акторів як будівельні блоки верхнього рівня, так що весь код у вашій системі знаходиться в системі акторів, ви використовуєте модель актора. З іншого боку, якщо ви будуєте свою систему таким чином, що у вас є актори, що знаходяться всередині неакторів, тоді ви не використовуєте модель актора.

Також важливо розуміти, що програмування в моделі актора не стосується якогось одного інструменту чи технології. Навколо ідеї акторської моделі напи-сані цілі мови. Таким чином, це більше схоже на парадигму програмування, ніж набір інструментів. Вивчення цього подібне до вивчення об’єктно-орієнтованого програмування або функціонального програмування. І як ці дві методології, модель актора передує Akka, і є багато інших її реалізацій.

Щоб реалізувати модель актора, слід дотримуватися кількох основних правил рис. 1.15:

- Усі обчислення виконуються в акторі
- Актори можуть спілкуватися лише за допомогою повідомлень, стан актора може бути змінений тільки через повідомлення.
- Актори можуть створювати інших акторів

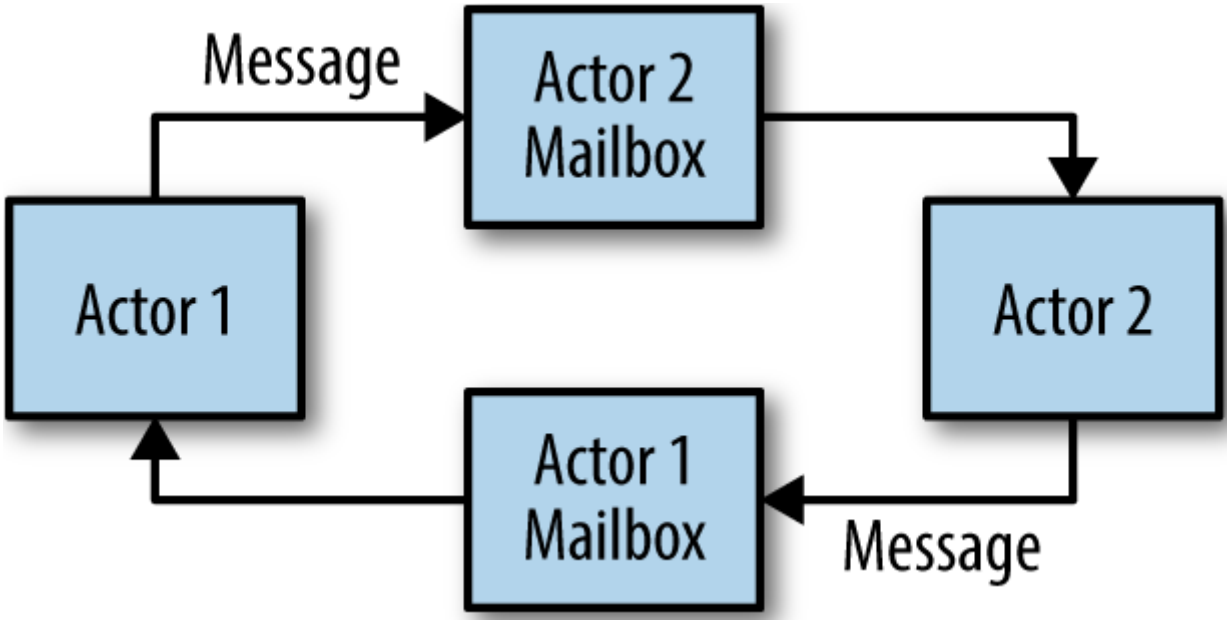


Рисунок 1.15 Спілкування акторів між собою

Отже спробую більш детально описати деталі моделі з точки зору практичного підходу і розроки програмного забезпечення.

Першим питанням, яке в мене виникло: якщо тільки актори можуть робити обчислювальні операції, то де межа поділу акторів, адже задачу можна розділити на величезну кількість рівнів абстракції, тобто під задач. На мою думку, ділити варто до рівня сервісу, а з цією задачею може допомогти підхід DDD. Насправді цей підхід є досить об’ємною темою, тому якщо коротко це про поділ великого монолітного сервісу на мікросервіси шляхом виділення доменних областей. Домен - це предметна область.

Друге питання яке в мене виникло, що дає нам спілкування акторів за допомогою повідомлень і скриньок? Спілкування акторів за допомогою повідомлень і скриньок забезпечує низку ключових переваг, які роблять цю модель ефективною для побудови масштабованих, безпечних та реактивних систем. Ось основні переваги, які мені вдалось виділити для себе:

- Не блокує потоки: відправлення повідомлення від одного актора до іншого відбувається асинхронно. Відправник не чекає завершення обробки, що забезпечує високу продуктивність.
- Ізоляція стану: спілкування через повідомлення гарантує, що стан не буде змінено неконтрольовано іншими потоками. Це усуває необхідність викори-

стання складних механізмів синхронізації, таких як блокування або семафори.

- Безпека даних: актори обробляють повідомлення послідовно, одне за іншим. Це забезпечує консистентність стану.
- Масштабованість: додавання нових акторів або перенесення існуючих на інші вузли не вимагає змін у логіці роботи.
- Контроль навантаження: поштова скринька актора дозволяє регулювати навантаження: актор обробляє лише стільки повідомлень, скільки може. У випадку перевантаження повідомлення можуть бути відкинуті або збережені для обробки пізніше.

Наступним питанням виникло, хто ж створює нових акторів? У моделі акторів зазвичай це завдання самих акторів. Тобто актори породжують акторів. Це створює певну ієрархію, що надає свої переваги. Наприклад, коли виникає якась помилка на якомусь з рівнів, то актор може її обробити або передати вище рис. 1.16.

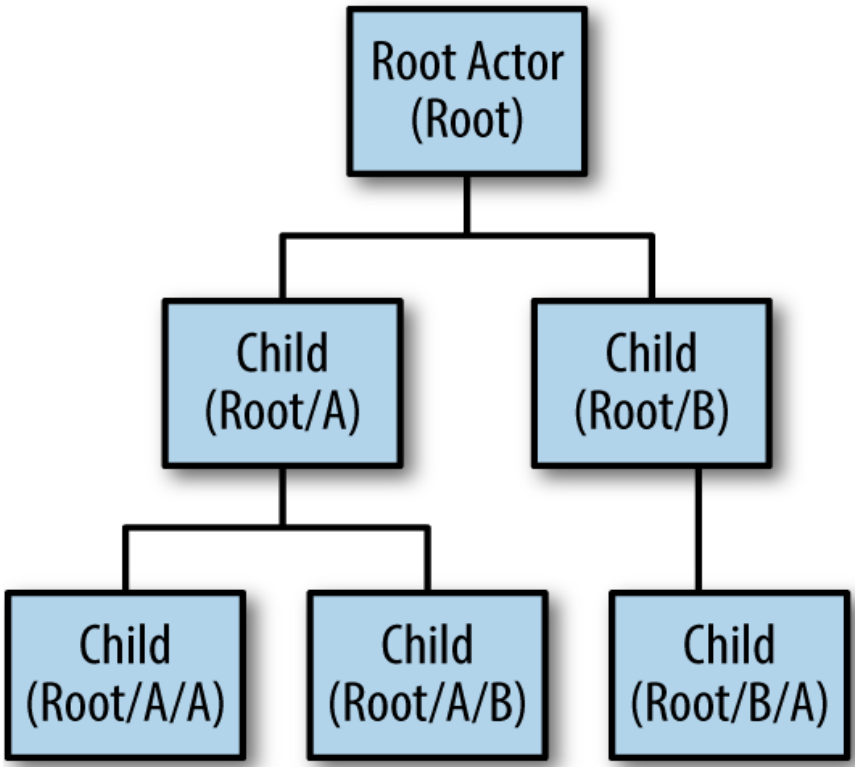


Рисунок 1.16 Ієрархія акторів

Коли ми повинні використовувати автономні актори, а коли ми повинні створити повноцінну систему акторів? Звичайно, неможливо сказати, що ви

завжди повинні використовувати те чи інше в певній ситуації. Кожна ситуація унікальна. Тим не менш, є деякі вказівки, які ви можете взяти до уваги, приймаючи рішення.

Очевидний кандидат для використання акторів - це коли у вас високий ступінь паралельності і вам потрібно підтримувати певний стан. Підтримка паралельного стану - це хліб з маслом моделі актора.

Більш тонкий випадок, який може підійти акторам, це коли вам потрібен високий ступінь паралельності, але вам потрібно бути обережним у тому, як керувати цією паралельністю. Наприклад, вам може знадобитися переконатися, що певний набір операцій може виконуватися одночасно з іншими операціями у вашій системі, але також, що вони не можуть працювати одночасно одна з одною.

Отже, підводячи підсумки варто сказати, що модель акторів є дуже важливою темою для кращого розуміння розробки програмного забезпечення. На основі цієї моделі написані мови програмування і навіть деякі аспекти були взяті до уваги, коли була розроблена концепція ООП. В рамках розробки реактивних системи, модель акторів також є надзвичайно важливою, адже добре інтегрується з принципами реактивного маніфесту. Саме тому на основі моделі акторів побудовані деякі реактивні фреймворки.

2 АРХІТЕКТУРА РЕАКТИВНИХ ФРЕЙМВОРКІВ

2.1 Багатопотоковість та реактивність в Java

2.1.1 Чому Java?

Насправді, як я вже прояснив, реактивне програмування - це є концепцією, що покладається на реактивний маніфест. Тобто, ми можемо писати реактивні системи на багатьох мовах програмування. Але інші питання в тому, наскільки добре це підтримується зі сторони ком’юніті? Як багато готових рішень? Наскільки швидкою є розробка?

Отже, якщо ми вирішили, що наша система має бути реактивною, то скоріше за все ми будемо працювати з системою з великим навантаженням, де потрібна гарна відмовостійкість. З власного досвіду я бачив вдалу реалізацію таких систем на Java і на C++, також часто реалізовують на пайтоні і подібних мов до Java: Scala, Groovy, Go. Одразу варто підмітити, що система, яку я бачив на C++ хоч і працювала довгі роки в продакшині, але дуже важко підтримувалась. З іншої сторони система на Java підтримувалась набагато легше і меншими ресурсами. Власне тому я б і хотів звернути увагу на Java і оглянути її переваги для реактивного програмування.

Java це кросплатформенна мова. Тобто через особливості реалізації цієї мови вона може виконуватись однаково на будь-якій обчислювальній машині.

Java добре підтримувана мова. В цій мові постійно виходять оновлення — нові версії. Також гарна підтримка ком’юніті. Є купа ресурсів, де знайти рішення для своєї проблеми та просто прочитати про різні функції в самій мові або інтеграцію з різними інструментами.

Java має декілька крутих фреймворків, які значно пришвидшують розробку програмного забезпечення. Основні з них це Spring та Quarkus. Spring є більш популярним через більший набір інструментів і кращу підтримку. Натомість, Quarkus більш легший в плані виділення пам’яті. Надалі буду використовувати Spring, адже маю більше практичного досвіду при роботі з ним.

Java підтримує більшість протоколів та моделей для мережного спілкування, таких як: tcp, udp, websocket, grpc, http, rsocket та інші.

Якщо говорити про Java в контексті реактивного програмування, то ця мова відмінно підходить під цю задачу. Адже, в Java добре підтримується багатопоточність (в нових версіях навіть присутні віртуальні потоки) та асинхронність.

Java має велику кількість інструментів, які підтримують реактивний підхід:

- Project Reactor: Це ядро Spring WebFlux, яке забезпечує асинхронне програмування з підтримкою реактивного потоку (Reactive Streams).
- RxJava: Поширена бібліотека для створення реактивних потоків із високим рівнем абстракції.
- Vert.x: Полегшений та високопродуктивний фреймворк для асинхронного програмування, який чудово підходить для масштабованих додатків.
- Akka: Інструмент для обробки повідомлень та реактивного підходу, заснований на акторній моделі.

Отже, я планую будувати майбутню систему на Java варто розібратись як працює багатопоточність в середині цієї мови, як виконується доступ до ресурсів, тощо.

2.1.2 JMM

Кожна строчка коду написана розробником може розглядатись, як інструкціє обчислювальній машині. Але насправді ці інструкції можуть віконуватись не в тому порядку в якому написаний код. Це відбувається через різні оптимізації на різних рівнях: компілятор байткоду, компілятор на рівні машинного коду, наприклад, серед компіляторів поширена така оптимізація як Instruction scheduling, різні процесорні оптимізації. Всі ці оптимізації робляться з метою підвищення продуктивності програм:

- Переупорядкування необхідно для того, щоб знайти найоптимальніший шлях до виконання коду з огляду на вартість виконання процесорних інструкцій. Наприклад, процесор може започаткувати завантаження значення з пам'яті заздалегідь, навіть якщо в порядку програми це читання йде пізніше.

- Операції читання з пам'яті коштують дорого, тому ця оптимізація дозволяє максимально ефективно утилізувати процесор, уникнувши простоювання, коли це читання справді знадобиться. Читання з регістру та кешу коштує набагато дешевше, ніж читання з пам'яті. Більш того, локальний кеш необхідний для того, щоб ядра не простоювали в очікуванні доступу до спільного кешу, а могли працювати з кешем незалежно один від одного.

  На щастя, для розробників:

- Java дає гарантію as-if-serial виконання коду - незалежно від використовуваної JDK підсумковий результат виконання буде не відрізнятиметься від такого порядку, як якщо б дії виконувались дійсно послідовно згідно порядку в коді
- Процесори теж роблять лише такі переупорядкування, які не змінять підсумкового результату виконання інструкцій
- Процесори мають Cache Coherence механізм, який гарантує консистентність даних серед локальних кешів: як тільки значення потрапляє в локальний кеш одного ядра, воно буде видно всім іншим ядрам.

  На жаль:

- Java дає as-if-serial гарантію лише для єдиного треду в ізоляції. Це означає, що у багатопотоковій програмі при роботі з shared даними ми можемо не побачити записи там, де покладаємось на порядок виконання дій у коді іншого треду. Іншими словами, для першого треду в ізоляції валідно переупорядковувати інструкції місцями, якщо це не вплине на результат виконання, але переупорядкування може вплинути на інші треди
- Процесор також дає гарантію лише для єдиного ядра в ізоляції.
- Cache Coherence справді гарантує читання актуальних значень, але пропагація запису відбувається не миттєво, а з деякою затримкою.

Також не менш важливим нюансом при роботі з пам’ятю є проблема запуску коду на різному апаратному забезпеченні. Наприклад, ARM архітектура має набагато слабші гарантії порівняно з x86: ми можемо виявити набагато більше багів у програмі, якщо одного разу стабільно працюючу на x86 програму запустимо на ARM.

То як тоді програми написанні на джава з використання багатопоточності взагалі працюють? Чи можна покладатись що від запуску до запуску результат буде однаковим? Java вважається кросплатформенна мова програмування, як вирішується проблеми роботи з пам’яттю? Я думаю, що вже сталот очевидно, щол для цього всього потрібне рішення на рівні специфікації мови програмування. І справді, в Java є JMM - Java Memory Model.

JMM допомагає зробити мову кросплатформенною та багатопоточною. За замовчуванням JMM дозволяє будь-які переупорядкування та не гарантує видимості змін. Однак при виконанні певних умов гарантується порядок дій, консистентний з порядком у коді, а також видимість всіх змін. Таким чином, JMM дозволяє нам писати програми, які повністю коректно працюватимуть серед безлічі різних імплементацій JDK і мікроархітектур процесорів, водночас зберігаючи переваги оптимізації. Отже, для кращого розуміння багатопоточності в Java, варто розібратись як Java і потоки працюють з пам’яттю.

Кожен потік, запущений у віртуальній машині Java, має власний стек потоків. Стек потоку містить інформацію про те, які методи викликав потік для досягнення поточної точки виконання.

Стек потоків також містить усі локальні змінні для кожного методу, який виконується (усі методи в стеку викликів). Потік може отримати доступ лише до власного стеку потоків. Локальні змінні, створені потоком, невидимі для всіх інших потоків, крім потоку, який їх створив.

Навіть якщо два потоки виконують той самий код, вони все одно створюватимуть локальні змінні цього коду у кожному своєму стеку потоків. Таким чином, кожен потік має свою власну версію кожної локальної змінної. Усі локальні змінні примітивних типів (boolean, byte, short, char, int, long, float, double) повністю зберігаються в стеку потоків і тому невидимі для інших потоків. Один потік може передавати копію примітивної змінної в інший потік, але він не може спільно використовувати примітивну локальну змінну.

Купа містить усі об’єкти, створені у вашій програмі Java, незалежно від того, який потік створив об’єкт. Немає значення, чи був об’єкт створений і призначений локальній змінній, чи створений як змінна-член іншого об’єкта, об’єкт

все ще зберігається в купі. До об’єктів у купі можуть отримати доступ усі потоки, які мають посилання на об’єкт. Коли потік має доступ до об’єкта, він також може отримати доступ до змінних-членів цього об’єкта. Якщо два потоки викликають метод одного об’єкта одночасно, вони обидва матимуть доступ до змінних-членів об’єкта, але кожен потік матиме власну копію локальних змінних.

Ось діаграма, що ілюструє стек викликів і локальні змінні, що зберігаються в стеках потоків, а також об’єкти, що зберігаються в купі рис. 2.1:

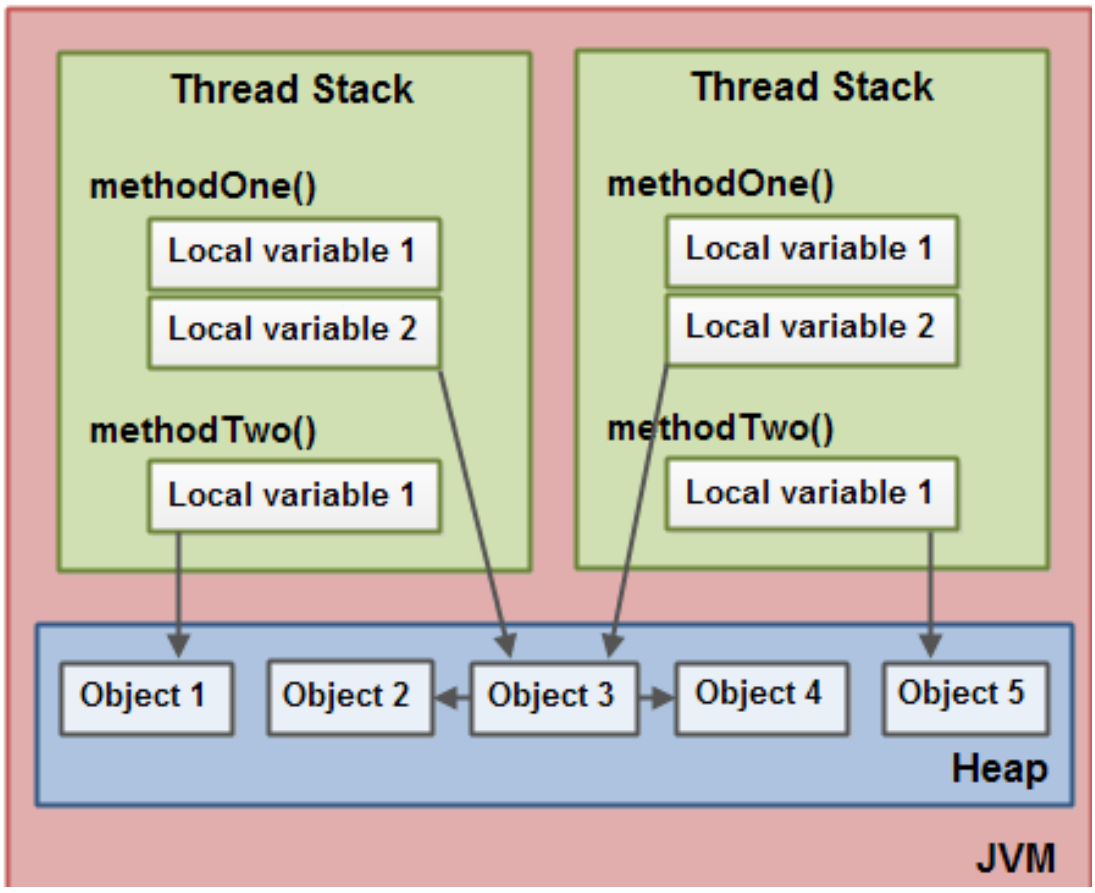


Рисунок 2.1 Java Memory Model

Також варто розуміти що апаратна архітектура відрізняється від абстракцій Java. Сучасні комп’ютери мають різні області збереження пам’яті. Окрім основної є ще різні рівні кешів. Апаратна архітектура пам'яті не розрізняє стеки потоків і купу. На апаратному забезпеченні як стек потоків, так і купа розташовані в основній пам’яті. Частини стеків потоків і купи іноді можуть бути присутніми в кеш-пам'яті ЦП і внутрішніх регістрах ЦП. Це показано на цій діаграмі рис. 2.2:

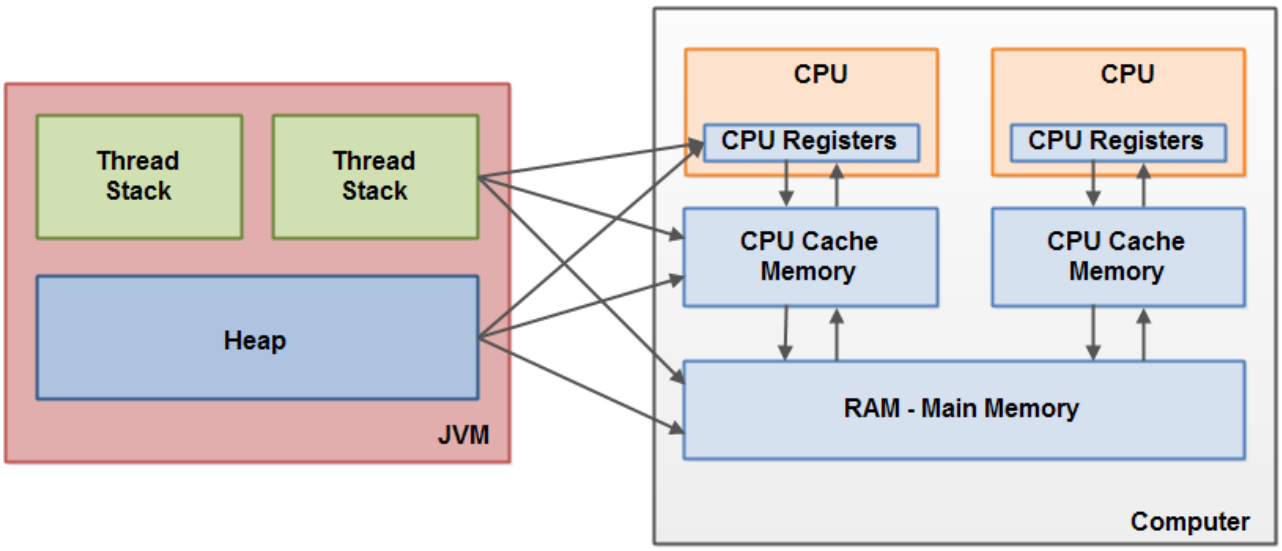


Рисунок 2.2 Робота Java застосунку з апаратною частиною

Коли об’єкти та змінні можуть зберігатися в різних областях пам’яті комп’ютера, можуть виникнути певні проблеми. Дві основні проблеми:

- Видимість оновлень потоків (записів) у спільні змінні.
- Умови перегонів під час читання, перевірки та запису спільних змінних.

Інколи бувають ситуації коли два потоки працюють з одним значенням з хіпа. Тоді виникає ситуація, що при змінні значення, інший потік не побачить зміни. Це через те що потік читає значення не з основної пам’яті, а з кеша, що передує основній пам’яті. Цю проблему можна побороти, використовуючи ключове слово volotile разом зі змінною в Java. Слово volotile гарантує, що змінна буде зчитуватись не з кеша, а з основної пам’яті.

Інша проблема - це перегони потоків. Коли два або більше потоків конкурують за виконання певної області коду. Якщо нам потрібно, щоб тільки один потік міг захопити і виконувати роботу в певній області, ми можемо використовувати ключове слово synchronized.

Ми можемо позначати блок коду synchronized або цілий метод. Важливо розуміти, що в джава у кожного об’єкта є монітор, що реалізує низькорівневий м’ютекс. Монітор каже чи захопив потік об’єкт, тобто чи виконується синхронізований блок чи метод. Це важливо розуміти, бо з цього випливає, що якщо обидва методи позначені як synchronized, два потоки не зможуть виконувати їх одночасно на тому самому об'єкті. Це відбувається через механізм монітора, який належить об'єкту. Коли потік виконує будь-який synchronized метод об'єкта, він

захоплює монітор об'єкта, блокуючи доступ до інших synchronized методів цього об'єкта для інших потоків.

2.1.3 Багатопоточність в Java

Коли запускається Java додаток, виконується метод main(). Метод виконується за допомогою головного потоку. Цей потік відповідає за виконання всіх інших інструкцій, зокрема за створення і запуск нових потоків. Всі потоки в Java є екземплярами класу java.lang.Thread або екземплярами підкласів цього класу. Можна створити і запустити цей клас таким чином рис. 2.3:

```
public class Main {
    public static void main(String[] args) {
        //Створення потоку
        Thread thread = new Thread();
        // Запуск потоку
        thread.start();
    }
}
```

Рисунок 2.3 Створення потоку

Звичайно такий потік нічого не буде робити, для того щоб потік виконував якусь логіку, потрібно перевизначити метод ран у класу Thread або інтерфейсу Runnable. Зазвичай використовують інтерфейс замість класу. Причиною є обмеження перевизначення класів — клас може мати тільки один батьківський клас і безліч інтерфейсів рис. 2.4.

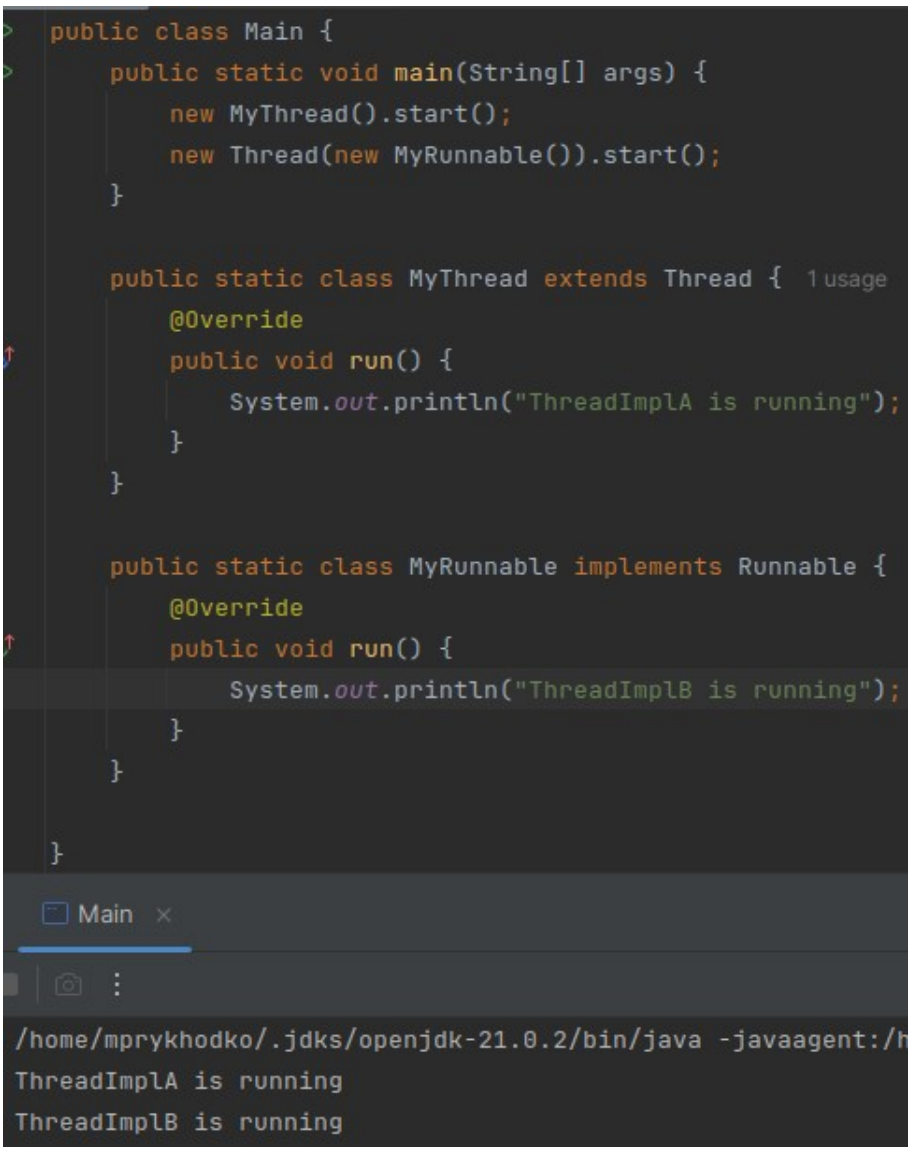


Рисунок 2.4 Приклад перевизначення run()

Під час створення та запуску потоку типовою помилкою є виклик run() методу Thread замість start(). Спочатку ви можете нічого не помітити, оскільки метод Runnable run() виконується так, як ви очікували. Однак він не виконується новим потоком, який ви щойно створили. Замість цього run() метод виконується потоком, який створив потік. Щоб run() метод екземпляра MyRunnable викликався новоствореним потоком, ви повинні start() метод.

Також Thread має декілька корисних методів для зручності керування потоками, декілька основних методів:

1. Назва потоку. Можна дати потоку назву:Thread thread1 = new Thread("потік1"). Це інколи може бути зручно для відстеження роботи декількох потоків. Для отримання назви просто викличте метод thread.getName().
2. Призупинення потоку за допомогою статичного метода Thread.sleep(100)

3. Потоки демона - це потоки, які припиняють свою роботу якщо головний потік припиняє роботу. Зазвичай такі використовують для фонових задач, які не потрібно робити якщо основна задача не виконується. Щоб зробити потік демона потрібно вказати значення true для метода thread.setDaemon(true);
4. Пріоритетність потоків можна визначити викликавши thread.setPriority(1). Чим вище значення пріоритету потоку, тим більше буде виділятись процесорного часу на нього.
5. Отримати статус потоку можна за допомогою методу thread.state(). Потік може мати декілька станів: new(потік створений, але не стартував), runnable(потік, що виконується), blocked(тред, що чекає отримання монітору на об’єкт, який заблокований іншим потоком), waiting(тред, що чекає виконання від іншого потоку, зазвичай така ситуація через виклик join() метода), timed waiting(тред, що заснув після виклику sleep()), terminated(потік, що виконав роботу).

В Java свого часу існувала проблема, що під час нової задачі створювався новий потік. Якщо задач було занадто багато було і багато потоків. Для більш оптимального використання робили пул потоків. Тобто сховище де зберігались вже створені потоки і коли надходила нова задачка брався вільний потік з цього сховища. Але це було важко кожного разу реалізовувати такі пули потоків, до того ж розробники потребували додаткових варіантів керування потоками. Саме тому з певної версії в Java додали інтерфейс java.util.concurrent.ExecutorService і декілька його реалізацій. ExecutorService обробляє створення, керування та повторне використання потоків , полегшуючи виконання одночасних завдань у багатопоточних програмах рис. 2.5.

Рисунок 2.5 Реалізація ExecutorService

ExecutorService має власні реалізації для різних ситуацій:

- FixedThreadPool - створює пул з фіксованою кількістю потоків. Використовується коли відома постійна кількість потоків і потрібне жорстке обмеження ресурсів.
- CachedThreadPool - це пул де створюються динамічно, якщо потік вільний то він перевикористовується, якщо потік не використовується певний час то він видаляється. Час життя потоків задається параметром. Використовується, коли навантаження не відоме спочатку або це навантаження не постійне.
- SingleThreadExecutor — це пул, що містить лише один потік. Використовується коли потрібно послідовне виконання завдань.
- ScheduledThreadPoolExecutor — виконує завдання з певною періодичністю. Можна використати, наприклад, для перевірки стану іншого сервісу.
- ForkJoinPool - це пул потоків, створений для виконання завдань у стилі "розділяй і володарюй". Він працює з задачами, які можна розділити на підзадачі, а потім об'єднати результати. Реалізує механізм "крадіжки роботи" - якщо один потік виконує всі свої завдання, він "краде" завдання у інших потоків. Добре підходить для обчислювальних задач, де потрібно максимально ефективно обробити великі об’єми даних.

2.1.4 Асинхронність в Java

На прикладі ExecutorService вже можна побачити певну асинхронність. Головний потік отримує задачі і передає іншим потокам, які в свою чергу виконують поставлену задачу. Виникає питання, якщо потрібно щоб потік в ExecutorService виконав задачку і повернув результат, як це зробити? На це питання є відповідь — інтерфейси, Future і Callable.

Future - це інтерфейс, що представляє результат асинхронної обробки.

Callable - це інтерфейс по типу Runnable, але якщо Runnable нічого не повертає, то Callable повертає значення рис. 2.6.

```
public class Main {
    public static void main(String[] args) throws ExecutionException, InterruptedException {
        ExecutorService  executorService  = Executors.newSingleThreadExecutor();
        Future<Integer> future = executorService.submit( new  Callable<Integer>() {
            @Override
            public Integer call () {
                return  42 ;
            }
        } );

        Integer  result  = future.get();
        System.out.println(result);
        executorService.shutdown();
    }

}
```

```
Main
/home/mprykhodko/.jdks/openjdk-21.0.2/bin/java -javaagent:/home/mprykhodko/.local/share/JetBrains
42

Process finished with exit code 0
```

Рисунок 2.6 Приклад коду з Callable

Додавання інтерфейсу Future в Java вирішило деякі проблеми, але цей інтерфейс мав і деякі недоліки. Основні з них це блокуючий метод get() і маленький набір методів.

Власне тому у Java 8 додали клас CompletableFuture, що імлементує Future. Ось деякі переваги:

- Можна використовувати як Future, а також дозволяє нам приєднувати зворотні виклики за допомогою таких методів, як thenApply , thenAccept і thenRun . Ці методи дозволяють нам виконувати додаткові дії після завершення вихідного завдання без блокування. CompletableFuture також можна завершити вручну.
- Future не має механізму обробки винятків. CompletableFuture має, наприклад, методи handle і exceptionally.
- Future механізму об’єднання результату. CompletableFuture має, наприклад, методи thenCompose, thenCombine та allOf рис. 2.7.

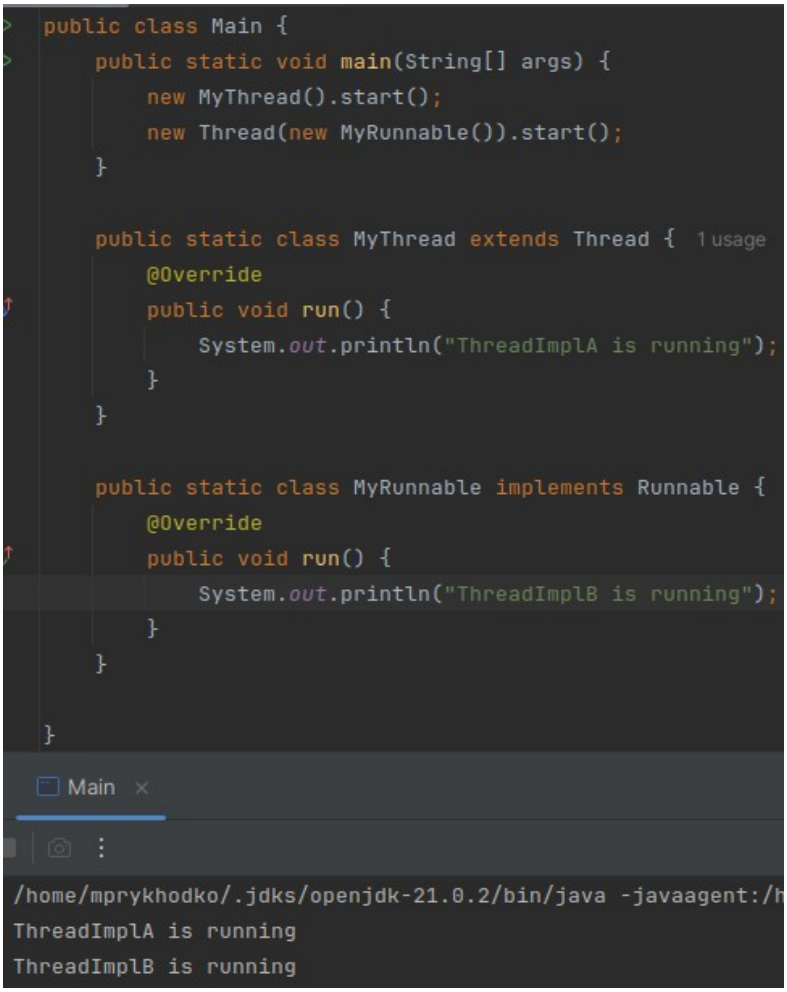


Рисунок 2.7 Приклад коду з CompletableFuture

2.1.5 Java NIO

Найперші версії Java (1995–2002) представили достатньо об’єктно-орієнтованого фасаду, щоб приховати деякі складніші деталі, але створення складного клієнт/серверного протоколу все ще вимагало багато шаблонного коду (і неабиякої кількості заглядань під витяжка, щоб усе працювало гладко). Ці перші

Java API підтримували лише так звані функції блокування, які надавалися рідними системними бібліотеками сокетів рис. 2.8.

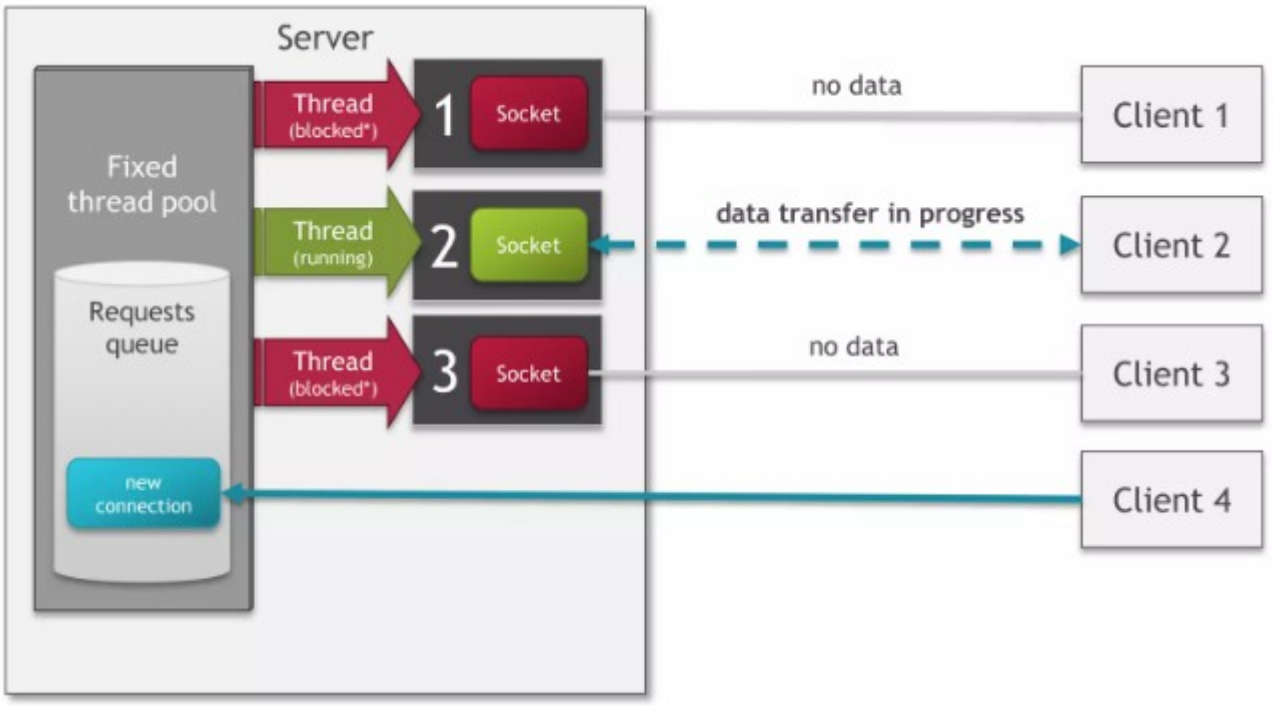


Рисунок 2.8 Приклад блокуючого клієнт-сервер зв’язку

Розглянемо наслідки такого підходу. По-перше, у будь-який момент багато потоків можуть бути неактивними, просто чекаючи появи в рядку вхідних або вихідних даних. Це може бути марною тратою ресурсів. По-друге, кожен потік вимагає виділення стекової пам’яті, розмір якої за замовчуванням коливається від 64 КБ до 1 МБ, залежно від ОС. По-третє, навіть якщо віртуальна машина Java (JVM) може фізично підтримувати дуже велику кількість потоків, накладні витрати на перемикання контексту почнуть викликати занепокоєння задовго до того, як буде досягнуто цього ліміту, скажімо, коли ви досягнете 10 000 підключень. На щастя, є альтернатива - Java NIO.

Підтримка Java для неблокуючого вводу-виводу була представлена в 2002 році з пакетом JDK 1.4 java.nio.

Java NIO дозволяє виконувати неблокуючий IO. Наприклад, потік може попросити канал прочитати дані в буфер. Поки канал зчитує дані в буфер, потік може робити щось інше. Після того, як дані зчитуються в буфер, потік може продовжити їх обробку. Те саме стосується запису даних у канали.

В Java NIO варто виділити три сутності це канал, буфер, та селектор.

Канал - це шлях, через який дані передаються між програмою та джерелом/приймачем даних (наприклад, файлом, сокетом або іншим пристроєм вводу-виводу).

Основні характеристики:

- Двосторонній: Канал може бути використаний для читання та запису даних.
- Низькорівневий доступ: Канал працює безпосередньо з джерелами даних, такими як файли чи мережеві сокети.
- Асинхронність: Використання каналів дозволяє працювати з даними не-блокуючим способом.

Приклади основних каналів:

- FileChannel: Для роботи з файлами.
- SocketChannel: Для роботи з TCP.
- DatagramChannel: Для роботи з UDP.

Буфер - це контейнер для зберігання даних, які ви хочете прочитати або записати через канал.

Основні характеристики:

- Зберігає дані: Буфер використовується як проміжний накопичувач.
- Обмежена ємність: Має фіксований розмір, який задається під час створення.
- Позиціонування та межі: Керує позицією (position), межами (limit) і ємністю (capacity), що дозволяє точно контролювати читання та запис даних.

Буфера бувають різних типів, основні з них: ByteBuffer, CharBuffer, DoubleBuffer, FloatBuffer, IntBuffer, LongBuffer, ShortBuffe.

Отже, дані читаються або записуються через буфер. В буфер дані потрапляють через канал. В один канал дані можуть записуватись або зчитуватись з декількох буферів. Канал створюється для доступу до ресурсу: файлу, сокета, тощо.

Селектор - це компонент, який може слідкувати за одним або більше екземплярів каналу і визначати, які канали готові, наприклад, для читання чи запису. Таким чином один потік може керувати декількома каналами, а отже кількома мережевими підключеннями.

Перевага використання лише одного потоку для обробки кількох каналів полягає в тому, що вам потрібно менше потоків для обробки каналів. Насправді ви можете використовувати лише один потік для обробки всіх ваших каналів. Перемикання між потоками є дорогим для операційної системи, і кожен потік також займає деякі ресурси (пам’ять) в операційній системі. Тому чим менше ниток ви використовуєте, тим краще рис. 2.9.

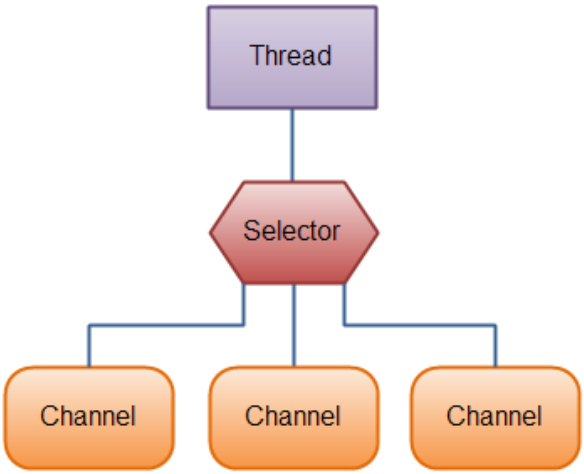


Рисунок 2.9 Java NIO: потік використовує селектор для обробки 3 каналів

На основі знань про деякі з інструментів Java бібліотеки nio можна побудувати загальну модель модель роботи сервера з використанням неблокуючого вводу/виводу та мультиплексування за допомогою Selector. Алгоритм функціонування можна описати так рис. 2.10:

1) Підключення клієнтів до сервера: Сервер приймає підключення клієнтів через канали (socket channel). Кожен клієнт обслуговується через окремий канал.
2) Мультиплексування за допомогою Selector: Selector перевіряє стан кожного каналу, щоб визначити, чи є нові дані для читання або запису. Якщо даних немає, канал ігнорується, щоб не блокувати сервер.
3) Обробка даних: Якщо на каналі є дані, ці дані передаються в буфер. Буфери забезпечують тимчасове зберігання даних для обробки, дозволяючи працювати з потоками незалежно від швидкості передачі даних.
4) Обробка бізнес-логіки: Потоки беруть дані з буферів і виконують обробку відповідно до бізнес-логіки.
5) Результати обробки можуть бути повернуті клієнтам через ті ж канали.

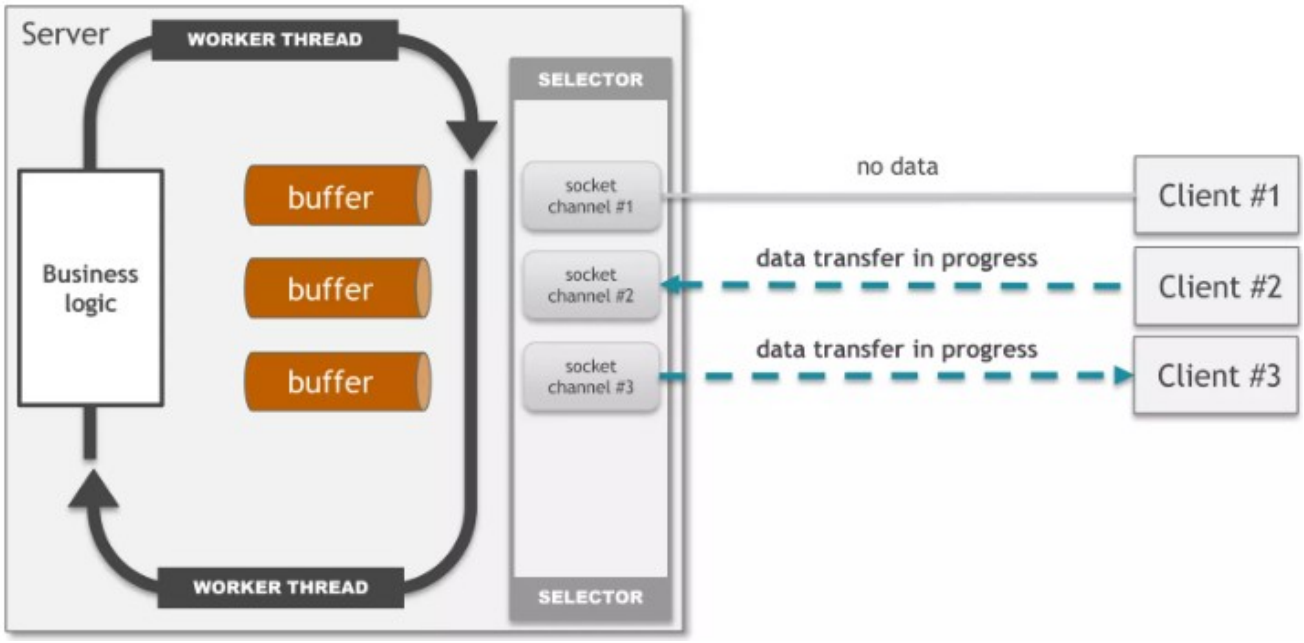


Рисунок 2.10 Приклад серверної архітектури побудованої на nio

2.1.6 RxJava

Я вже писав раніше, що джава класно інтегрується в реактивні системи. Тому логічно, що ця мова програмування також матиме власні реактивні бібліотеки і їх реалізації.

RxJava - це реалізація бібліотеки для платформи JVM на основі стандарту реактивних стрімів, яка підтримує реактивне програмування на основі патерну спостерігач. Ця бібліотека забезпечує зручний інтерфейс для створення асинхронних потоків даних.

Основні компоненти:

- Observable - джерело даних, яке генерує потік подій.
- Observer - об’єкт, який спостерігає за Observable та реагує на події.
- Schedulers - інструменти для управління потоками виконання.
- Operators - функції для трансформації, комбінування та фільтрації даних.

RxJava працює наступним чином. Спочатку створюємо Observable. Observable є основним джерелом даних. Його можна створити кількома способами:

just(): створює Observable з одного або кількох об'єктів. Observable.just("Привіт", "Світ") .subscribe(System.out::println);

fromIterable(): створює Observable з колекції. Observable.fromIterable(List.of(1, 2, 3, 4)).subscribe(System.out::println);

create(): дозволяє вручну визначати, як генеруються події.

```
Observable.create(emitter -> {
   emitter.onNext("Подія 1");
   emitter.onNext("Подія 2");
   emitter.onComplete();
}).subscribe(System.out::println);
```

RxJava має велику кількість операторів(методів). Оператори дозволяють трансформувати дані, фільтрувати їх, об’єднувати кілька потоків і виконувати інші операції.

map() — трансформує кожен елемент.

```
Observable.just(1, 2, 3)
        .map(x -> x * 2)
        .subscribe(System.out::println);
```

filter() — залишає тільки ті елементи, які відповідають умовам.

```
Observable.just(1, 2, 3, 4)
        .filter(x -> x % 2 == 0)
        .subscribe(System.out::println);
```

flatMap() — перетворює елементи на інші Observable.

```
Observable.just("A", "B")
        .flatMap(letter -> Observable.just(letter + "1", letter + "2"))
        .subscribe(System.out::println);
```

RxJava дозволяє легко керувати потоками виконання завдяки Schedulers:

- Schedulers.io() — для вводу/виводу.
- Schedulers.computation() — для обчислювальних задач.
- Schedulers.newThread() — для створення нових потоків.

Використання subscribeOn() і observeOn() дозволяє визначати, на яких потоках виконуватимуться різні етапи обробки:

```
Observable.just("Дані")
        .subscribeOn(Schedulers.io())
        .observeOn(Schedulers.computation())
        .map(data -> "Оброблені " + data)
```

.subscribe(System.out::println);

RxJava має вбудовані механізми обробки помилок:

- onErrorReturn() — повертає дефолтне значення.
- onErrorResumeNext() — переключається на інший Observable.
- retry() — повторює операцію у разі помилки.

На практиці RxJava зазвичай використовується для ефективної обробки великих обсягів даних. Але також може використовуватись для REST API при умові інтеграції з іншими фреймворками.

RxJava є потужним інструментом для побудови реактивних систем. Вона забезпечує високу продуктивність, масштабованість та гнучкість у роботі з потоками даних. Завдяки зручному API та інтеграції з іншими бібліотеками, RxJava дозволяє створювати сучасні додатки, які відповідають вимогам реального часу.

2.2 Протоколи передачі даних

2.1.2 Важливість правильного протоколу

У сучасних розподілених системах правильний вибір мережевого протоколу є критично важливим для забезпечення ефективності, надійності та безпеки комунікацій між пристроями та додатками. Це особливо актуально в умовах зростання кількості мережевих пристроїв і підвищення вимог до швидкості передачі даних, масштабованості та інтеграції з різноманітними системами. Вибір мережевого протоколу тісно пов'язаний з моделлю OSI, яка допомагає структурувати і стандартизувати комунікаційні процеси.

Модель OSI (Open Systems Interconnection) є концептуальною основою, що описує, як дані передаються в мережах. Вона складається з семи рівнів, кожен з яких виконує певні функції, що сприяють передачі даних від одного пристрою до іншого. Ця модель дозволяє зрозуміти, як різні протоколи взаємодіють між собою, і як правильно вибрати їх для певних умов:

- Фізичний рівень визначає фізичні характеристики передачі даних, такі як електричні сигнали або радіохвилі. Наприклад, Ethernet використовує стандарти цього рівня для кабельного з'єднання.

- Канальний рівень забезпечує надійну передачу даних через фізичне середовище, використовуючи протоколи, такі як Wi-Fi (802.11) або Ethernet (802.3).
- Мережевий рівень відповідає за маршрутизацію пакетів даних, використовуючи протоколи, такі як IP (Internet Protocol).
- Транспортний рівень гарантує цілісність передачі даних між хостами, використовуючи TCP (Transmission Control Protocol) або UDP (User Datagram Protocol).
- Сеансовий, представницький і прикладний рівні забезпечують управління сесіями, представлення даних і взаємодію користувачів із системою через такі протоколи, як HTTP, FTP, SMTP рис. 2.11.

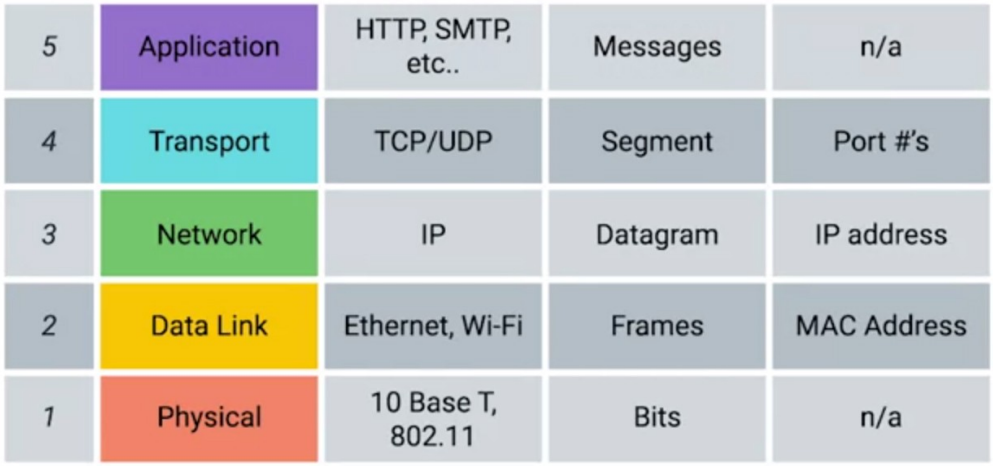


Рисунок 2.11 Модель OSI і протоколи

Кожен рівень моделі OSI спрямований на вирішення певних задач і визначає набір протоколів, які впливають на мережеву взаємодію.

В розрізі теми нас найбільше цікавить транспортний та прикладний рівні. При виборі протоколу я буду керуватись декількома основними характеристиками:

- Підтримка - це значить, що протокол має мати гарну підтримк ком'юніті, використовучи протокол має бути зручно програмувати, тобто протокол має мати гарний API.
- Стабільність або відмовостійкість. Неправильний протокол може призвести до втрати даних або виникнення затримок. Наприклад, TCP забезпечує надійну передачу завдяки механізмам підтвердження і повторної передачі

втрачених пакетів, що підходить для таких сценаріїв, як передача файлів або електронна пошта. Натомість, для потокового відео UDP краще підходить завдяки низьким затримкам, навіть якщо це відбувається за рахунок можливої втрати деяких пакетів.

- Ефективність та гарний перформанс. Правильний протокол дозволяє оптимально використовувати доступні ресурси, такі як пропускна здатність каналу, обчислювальна потужність і енергія. Наприклад, для передачі невеликих даних у реальному часі, таких як телеметрія IoT, більш доцільно використовувати легкі протоколи, такі як MQTT або CoAP, які мінімізують накладні витрати на передачу.

Вибір правильного мережевого протоколу є ключовим фактором у забезпеченні ефективної та надійної роботи будь-якої мережевої системи. Модель OSI надає чіткий концептуальний каркас для аналізу і прийняття рішень у цьому процесі. Розуміння характеристик кожного рівня моделі та специфіки доступних протоколів дозволяє створювати мережеві системи, які відповідають сучасним вимогам до продуктивності, безпеки та масштабованості.

Отже надалі планую розглянути чотири досить популярні протоколи, які зазвичай використовують в реактивних: websocket, http, grpc та rsocket.

2.2.2 HTTP

Http/1 - це старий перевірений часом протокол, який класно інтегрований у різні фреймворки, зокрема Spring. Мабуть саме цей протокол можна назвати найпопулярнішим протоколом обміну даними.

На жаль, цей протокол був досить давно розроблений. В той час вимоги до сервісів були зовсім інші. Саме тому протокол має деякі недоліки:

- Формат передачі даних - ця версія підтримує тільки текстові варіанти передачі даних, що є досить великими за обсягами і недостатньо безпечними.
- Для того щоб пришвидшити процес передачі даних, браузер відкриває декілька паралельних TCP з’єднань. На жаль, кількість таких підключень є обмеженою (до 6 підключень).

- Блокування черги або head-of-line blocking. Справа в тому у версії 1.1 реалізована конвеєрна обробка, тобто можна передавати декілька запитів один за одним в одному TCP підключені, на відмінну від версії 1.0. Але запити передані один за одним заблокуються, запит перед ними не буде оброблено, що спричиняє вузьке місце.
- Повільна швидкість завантаження веб-сторінки. Сервер тільки віддає дані, але не визначає пріоритет цих даних.
- Довгі заголовки - у цій версії заголовки ніяк не оптимізуються заголовки і немає можливості повторного використання.

Версія HTTP/2 була випущена через 18 років і була спрямована на оптимізацію версії HTTP/1. Основою версії HTTP/2 є можливість мати декілька повідомлень в одному TCP з’єднані, тобто запити або відповіді передаються не один за одним а паралельно. Також ці запити розбиваються на кадри, що в свою чергу кодуються у бінарний формат. Розбиття на кадри дає можливість використати один і той самий кадр в декількох різних повідомленнях одного TCP з’єднання.

Проблеми, що вирішує HTTP/2:

- Робота з двійковим кодом. Усі дані упаковуються в кадри, які представлені у бінарному форматі, таким чином обсяг даних зменшується, а передача даних стає більш безпечною.
- Мультиплексування. Одне TCP з‘єднання можна розбити на безліч потоків з повідомленнями, це допомогло вирішити проблему з блокуванням черг запитів. А можливим стало,тому що кожне повідомлення розбивається на кадри на одній сторонні (кожен кадр має ідентифікатор потоку) і збираються на іншій стороні.
- Оптимізація розміру заголовків. Для стиснення заголовків HTTP/2 використовує HPACK. Це дозволяє стискати окремі значення під час передачі, а індексований список попередньо переданих значень дозволяє нам кодувати повторювані значення шляхом передачі значень індексів, які можна використовувати для ефективного пошуку та реконструкції повних ключів і значень заголовків.

- Встановлення пріоритетів потокам. Тепер кожному потоку можна встановити вагу від 1 до 256 і залежність від іншого потоку, що в свою чергу дозволяє побудувати дерево пріоритетів.
- Функція server push. Тобто, на додаток до відповіді на вихідний запит, сервер може надсилати додаткові ресурси клієнту без того, щоб клієнт запитував кожен із них явно.

Однак HTTP/2 має також власні недоліки. Одним з яких є те, що ця версія протоколу використовує протокол TCP. А TCP в свою чергу гарантує повну передачу пакетів, так і ще й в тому порядку в якому вони були надіслані. Тобто, якщо якимсь чином (наприклад проблеми з якістю мережі) пакет буде загублений, то всі паралельні повідомлення заблокуються в одному місці і будуть чекати поки потрібний пакет не буде знайдений. Тому у 2019 році був випущений HTTP/3, який по суті був покликаний вирішити проблему з TCP, за допомогою використання протокола QUIC, що в свою чергу базується на протоколі UDP.

Протокол UDP це по суті протокол, що не зважає на втрачені файли, його зазвичай використовують, тоді коли коректність доставлених даних не є надто важливою, але продуктивність є основним параметром. QUIC є модифікацією над UDP і відповідає за ті ж самі функції, що і TCP, за те щоб всі пакети були повністю доставленими. Отже QUIC виконує схожі функції, що і TCP, але виправляючи, деякі недоліки.

Варто відзначити, що http/3 є досить новим протоколом, тому його ще мало використовують у продакшені.

2.2.3 Websocket

Websocket - це сучасний мережевий протокол, що створений для дуплексного зв’язку, тобто повідомлення можуть відправлятись як від сервера до клієнта, так і від клієнта до сервера. Протокол побудований поверх TCP протоколу із використанням HTTP. Головною особливістю і перевагою є те що на відмінну від http, Websocket встановлює постійне з’єднання, що зменшує накладні витрати при постійному трафіку.

Websocket працює таким чином:

1. Встановлення з'єднання З'єднання WebSocket починається з HTTP-запиту за допомогою методу GET, але з додатковим заголовком Upgrade. Цей запит ініціює handshake між клієнтом і сервером, де вони погоджуються перейти з HTTP на WebSocket.
2. Якщо сервер підтримує WebSocket, він відповідає статусом 101 Switching Protocols.
3. Двостороння передача даних Після завершення handshake клієнт і сервер встановлюють постійне з'єднання, через яке можуть обмінюватися даними в обох напрямках без повторної ініціалізації HTTP-запитів. Websocket підтримує різні формати даних, як текстові (JSON) так і бінарні (Protobuf, Avro).
4. Закриття може ініціювати як клієнт, так і сервер, використовуючи спеціальні контрольні фрейми CLOSE.

Перевагами протоколу є:

- Відносно швидкий. Websocket є досить швидким протоколом, за допомогою якого можна непогано розподілити ресурси, адже працює поверх tcp.
- Постійне з’єднання є також перевагою, адже якщо трафік постійний, то не потрібно кожен раз робити з’єднання між сервером і клієнтом.
- Дуплексний. Протокол дозволяє відправляти запити в дві сторони.

Недоліками протоколу є:

- Складність розробки. Якщо писати на голому протоколі, то це буде дуже невдалим рішенням. Саме тому є різні апі поверх протоколу, наприклаж, socket.io, але як на мене навіть з віідповідними API розробка на тому ж спрінгу набагато важча і довша ніж на інших протоколах.
- Поганий backpreasure. Backpreasure - це здатність клієнта і сервера куерувати трафіком. Сервер робить попередній запит на те скільки клієнт може обробити трафіку і той йому відповідає. За відсутності цієї характеристики трафік просто пропадає, якщо клієнт не здатен обробити навантаження. У Websocket немає власного реалізованого backpreasure, а використовується тільки той, що на рівні tcp.

- Постійне з’єднання. Ця характеристика є як перевагою, так і недоліком. Адже, для того, щоб підтримувати постійний конекшин потрвбні додаткові ресурси. Якщо трафік не є постійним, то ці ресурси будуть просто простоювати і не приносити жодної користі.

2.2.4 GRPC

GRPC(Google Remote Procedure Call) - це технологія обміну даними, розроблена компанією Google і випущений у користування у 2015 році. Це технологія, що використовує покращений підхід RPC і бінарний формат protobuf для передачі даних.

Розробники описують інтерфейс сервісу, використовуючи мову інтерфейсу, таку як Protocol Buffers (protobuf). Цей опис включає визначення доступних методів сервісу, вхідних та вихідних параметрів і типи повідомлень, які використовуються для обміну даними. За допомогою інструменту gRPC, на основі опису інтерфейсу, генерується код на мові програмування, який використовуватиметься як клієнтська та серверна сторона для взаємодії.

Файл .proto можна уявити у вигляді словника, за допомогою якого дані кодуються та декодуються до двійкового формату буферів протоколу та з нього. Використання буферів протоколів вимагає, щоб і клієнт, і сервер під час обміну даними gRPC мали доступ до того самого визначення схеми рис. 2.12.

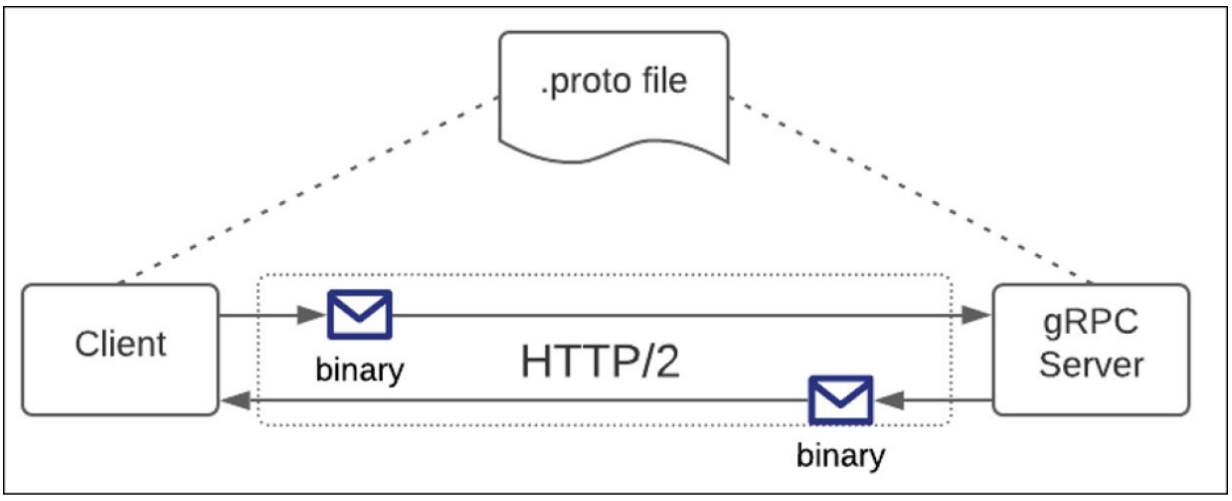


Рисунок 2.12 Схема використання .proto файлу

Як раніше було сказано, GRPC це покращений RPC підхід. Коли є інтерфейс (в нашому випадку він описується в proto-схемі), до якого мають доступ клієнтська і серверна сторона. На клієнті викликається метод заглушка, далі відправляється до серверної частини, де знаходиться реалізація цього методу, дані оброблюються і повертається готова відповідь до клієнтської частини.

Перевага використання двійкового формату як засобу обміну даними полягає в тому, що він підвищує продуктивність. Насправді порівняння protobuf формату і JSON є досить нетривіальне завдання, адже статистика порівнянь є досить різноманітною, оскільки вона залежить від багатьох факторів, таких як розмір повідомлення, складність логіки застосунку, розмір запиту та багато інших. Однак, деякі дослідження демонструють переваги використання gRPC у порівнянні з REST. У порівнянні з REST, gRPC має менший розмір пакета та менший час відправки запиту-відповіді, що може покращити продуктивність додатка.

За даними Microsoft Research, gRPC може зменшити розмір пакета майже на 90% та зменшити час відправки запиту-відповіді на 50% в порівнянні з REST. Інші дослідження показали, що gRPC може мати менше затримок при високій навантаженості, що може допомогти уникнути проблем зі шкалуванням додатка.

На жаль, використані grpc розробники можуть зіштовхнутись з деякими проблемами. Наразі неможливо реалізувати специфікацію HTTP/2 gRPC у браузері, оскільки немає API браузера з достатньо детальним контролем над запитами, і немає способу примусово використовувати HTTP/2, і навіть якщо він був, необроблені кадри HTTP/2 недоступні в браузерах. Отже, настав час, коли gRPC-Web виходить на сцену. gRPC-Web розроблено, щоб подолати цю прогалину, і він розроблений, щоб створити спосіб використання gRPC у веб-додатках і браузерах. У gRPC-web є дві важливі частини:

По-перше, реалізація gRPC-Web JavaScript у браузерах.

По-друге, gRPC-веб-проксі. Проксі перетворює запити з gRPC-Web на рідний формат gRPC. Існують різні проксі-сервери для gRPC-Web, один з них – це проксі-сервер Envoy .

Ось хороший приклад для вивчення дизайну gRPC-Web рис. 2.13:

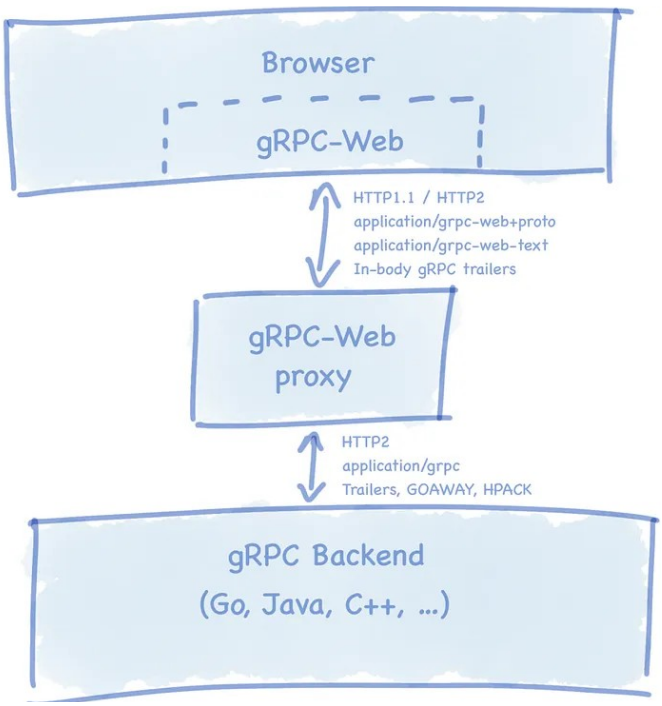


Рисунок 2.13 Схема використання grpc в браузері

Як я писав раніше важливою характеристикою під час розробки реактивної системи є backpreasure.

gRPC базується на протоколі HTTP/2, який має вбудовані можливості управління потоком через Window Size. Це дозволяє серверу або клієнту контролювати кількість даних, які можуть бути відправлені, але ще не підтверджені.

Особливості:

- Управління потоком відбувається на рівні протоколу HTTP/2.
- Потоки можуть бути тимчасово призупинені, якщо одна зі сторін не встигає обробляти дані.
- Розробнику доводиться додатково впроваджувати власну логіку для тоншого управління потоком (наприклад, через зворотний зв’язок між клієнтом і сервером).

Обмеження:

- Управління потоком у gRPC є грубим, оскільки HTTP/2 орієнтується лише на рівень мережевих буферів, не враховуючи логіку обробки даних у додатку. Тому розробникам треба це враховувати і додавати додаткову логіку обробки реквестів на стороні сервера і клієнта.
- Відсутність вбудованої реактивності.

2.2.5 RSocket

RSocket - це сучасний мережевий протокол, розроблений для ефективного та реактивного обміну даними між клієнтами та серверами. Він був створений для подолання обмежень традиційних протоколів, таких як HTTP/1.1, HTTP/2 та WebSocket, особливо в контексті реактивних систем. RSocket підтримує кілька моделей взаємодії, включаючи: запит-відповідь, стрімінг даних, канали та односторонню відправку.

RSocket побудований на основі концепцій реактивного програмування, підтримує асинхронність, використання потоків, а також високопродуктивну роботу через TCP, WebSocket або навіть HTTP/2.

Механізм забезпечується спеціальними frame-ами в протоколі, які сигналізують про кількість дозволених елементів у потоці. Потоки даних розбиваються на невеликі фрейми:

- Request frame для ініціалізації запитів.
- Payload frame для передачі даних.
- Error frame для обробки помилок.
- Cancel frame для скасування запиту.

Основні особливості Rsocket:

- RSocket підтримує backpressure - механізм контролю потоку даних, який дозволяє запобігати перевантаженню системи великим обсягом інформації. Цей механізм підтримує backpressure на рівні логічних даних - меседжів, а не на рівні байтів, як це зроблено в grpc, що значно спрощує розробку.
- Багато-модельна взаємодія. Запит-відповідь - типова модель, аналогічна HTTP. Запит-потік - сервер відповідає на запит клієнта потоком даних. Канали - двосторонній стрімінг між клієнтом і сервером. Односторонній потік - клієнт надсилає дані без очікування відповіді.
- Мультиплексування. Протокол дозволяє запускати кілька потоків даних через одне з'єднання, що зменшує накладні витрати та підвищує ефективність.
- Прозорість транспорту. RSocket може працювати поверх різних транспортних протоколів, таких як TCP, WebSocket або Aeron, без змін у коді.

RSocket - це потужний протокол, спеціально розроблений для сучасних реактивних систем. Він надає зручність і ефективність в управлінні потоками даних, знижує навантаження на ресурси та підтримує складні моделі взаємодії. Це робить його ідеальним вибором для масштабованих та продуктивних розподілених систем.

Саме RSocket виглядає для мене найбільш вдалим протоколом при побудові реактивної системи. Адже він має переваги над http за рахунок дуплексного конекшина і бінарного формату передачі даних. Він має перевагу на websocket через асинхронність і кращу зручність розробки сервісів. Він має перевагу над grpc через кращу реалізацію backpressure саме для реактивних систем. Особисто я вважаю, що за цим протоколом майбутнє, особливо. Коли QUIC стане більш використовуваним і популярним.

2.3 Реактивні фреймворки

2.3.1 Потреба в реактивних фреймворках

Java надає купу інструментів для побудови реактивних систем. Я вже, наприклад, розглянув бібліотеку nio для неблокуючих операції io. Також розглянув бібліотеку RxJava для реактивного програмування.

Але я переконаний, що цього недостатньо, що писати гарний, читабельний, а головне працюючий без помилок код, який добре розділений на різні абстрактні рівні.

Отже розглянемо, яку зручну підтримку не можна отримати пишучи тільки на Java без використання фреймворків.

Складно розробляти код сервера на голому nio або io. Також код часто повторюється. Але найбільша проблема, що його важко переписати з одного підходу на інший.

RxJava не є веб-фреймворком. Тому не пропонує:

- Вбудованих засобів для роботи з HTTP-запитами/відповідями.
- Управління життєвим циклом застосунку.

- Інтеграції з сучасними стандартами веб-розробки, як, наприклад, підтримка Servlet 3.1 чи Reactive Streams.
- Немає вбудованих інструментів для backpreasure.

RxJava має стрімку криву навчання. Розробникам доводиться багато часу приділяти освоєнню великої кількості операторів (flatMap, concatMap, merge тощо). Обробка помилок (error handling) в RxJava вимагає ретельного підходу, і помилки часто можуть залишатися непоміченими.

Відсутність централізованої екосистеми, RxJava надає інструменти для роботи з асинхронними потоками даних, але вона:

- Не пропонує модулів для інтеграції з базами даних, кешами, безпекою тощо.
- Не включає рішення для управління залежностями або конфігурацією.

Проблеми з багатопотоковістю, RxJava дозволяє легко переключати потоки (Schedulers), але:

- Неправильне використання може призвести до блокувань або зниження продуктивності.
- Важче управляти ресурсами (наприклад, з'єднаннями з базою даних).

Власне тому існують сторонні рішення - реактивні фреймворки. За допомогою, яких можна вирішити ту чи іншу проблему, що важко вирішити на Java. Я б розділив реактивні фреймворки на дві частини рис. 2.14:

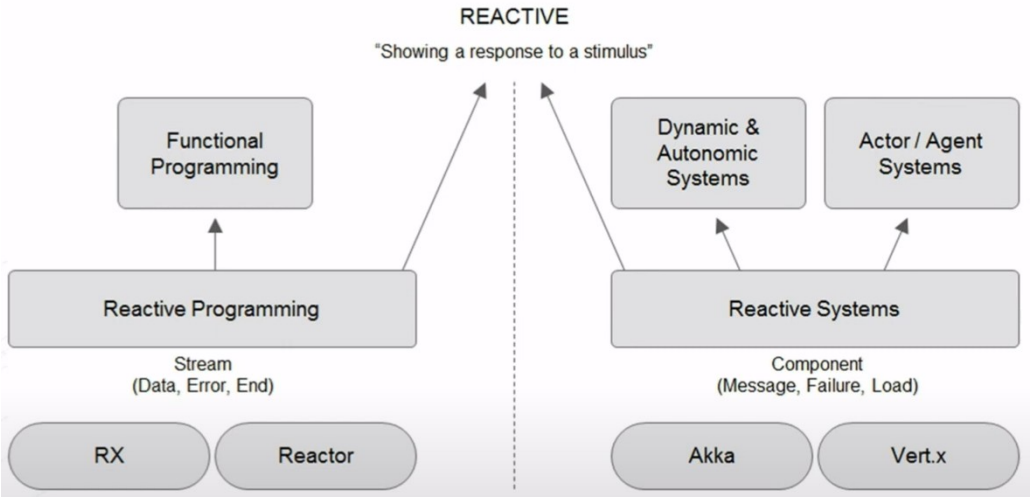


Рисунок 2.14 Групування реактивних фреймворків

Ця схема демонструє зв'язок між різними концепціями реактивного програмування та реактивних систем.

Фреймворки, для більш локального (на рівні сервісу) програмуванні взяли за основу стандарт реактивних стрімів. Обробка потоків даних (Stream), що складаються з подій: даних (Data), помилок (Error) і завершення (End). Програмування в таких інструментах базується на функціональному стилі, що дозволяє ефективно обробляти потоки даних. До таких фремфорків та бібліотек, я відношу:

- Rx (Reactive Extensions) - Java бібліотека для обробки асинхронних потоків даних.
- Reactor - бібліотека від Spring для роботи з реактивними потоками. Для веб застосунків використовується webflux, який будується на основі netty і reactor.

Фреймворки для побудови реактивних систем потрібні для створення динамічних систем, які реагують на події та адаптуються до змін. Такі фреймворки взяли за ідею маодель акторів або хоча б деякі принципи звідти. До таких фреймворків та бібліотек, я відношу:

- Akka - фреймворк для створення акторних систем. Повністю побудований на моделі акторів.
- Vert.x - інструмент для побудови асинхронних і масштабованих систем. Бере певні ідеї від моделі акторів.

2.3.2 Netty

Раніше я вже розглянув IO і NIO серверну архітектуру. Обидві мають власні переваги і недоліки. Також при написані серверу і клієнта на голому IO або NIO функціоналі мови програмування JAVA є досить складною задачкою, якщо ми говоримо про код, який справді буде працювати в продакшені і приносити користь. Також зазвичай більшість коду для реалізації серверу на “голій JAVA” часто повторюється.

Отже, як вже зрозуміло з контексту у JAVA ком’юніті виникла потреба у фреймворку, який би реалізовував шаблони для коду, що часто повторюється, оптимізував роботу з ресурсами, дозволив би розробників абстрагуватись від мережевого рівня програмування і тд. Саме через причини був створений фреймворк Netty.

Netty - це фреймворк для написання асинхронних мережевих додатків керованими подіями з інструментами для швидкої розробки підтримуваних високопродуктивних і масштабованих протокольних серверів і клієнтів.

Отже розгляну більш детально, які основні переваги надає Netty і спробую реалізувати невеличкий сервер за допомогою фреймворку.

Netty розширили структуру буффера, написавши власну реалізацію ByteBuf поверх реалізації джава класу ByteBuffer.

Переваги реалізації ChannelBuffer/ByteBuf над ByteBuffer:

1. Без автоматичного перевертання (flip): У ByteBuffer необхідно викликати flip() для перемикання між режимами запису та читання, що часто призводить до помилок і додаткової складності. У ByteBuf це зроблено автоматично.
2. Динамічний розмір: ByteBuf може динамічно змінювати розмір, тоді як розмір ByteBuffer фіксований після створення.
3. Менеджмент пам'яті: Netty підтримує ефективне керування пам'яттю через пул буферів, що зменшує кількість виділень і звільнень пам'яті.
4. Простий доступ до даних: У ByteBuf є методи для читання/запису різних типів даних (readInt(), writeInt(), readBytes(), тощо), що значно спрощує код.
5. Індексація: Можна читати й записувати дані за вказаним індексом без зміни позиції читання/запису.
6. Zero-Copy: Завдяки використанню таких механізмів, як slice, duplicate, та composite buffers, Netty дозволяє уникати копіювання даних між буферами, що значно підвищує продуктивність.
7. Пул буферів: Netty використовує пул для повторного використання буферів, що мінімізує накладні витрати на створення об'єктів і збільшує швидкодію.
8. CompositeByteBuf: Дозволяє об'єднувати кілька буферів у вигляді одного, що дозволяє працювати з великими наборами даних без фізичного об'єднання.
9. Direct vs Heap Buffers: ByteBuf дозволяє вибирати між буферами, розташованими в пам'яті JVM (Heap), або поза нею (Direct), залежно від потреб.

10. Reference counting: ByteBuf використовує підхід лічильників посилань (reference counting), що дозволяє точно контролювати життєвий цикл буферів і уникати витоків пам'яті.
11. Під час передачі даних між рівнями зв’язку дані часто потрібно об’єднувати або розділяти. Наприклад, якщо корисне навантаження розділене на кілька пакетів, його часто потрібно об’єднати для декодування. Традиційно дані з кількох пакетів об’єднуються шляхом їх копіювання в новий байтовий буфер. Netty підтримує підхід без копіювання, коли ByteBuf вказує на необхідні буфери, таким чином усуваючи необхідність виконувати копіювання.

Код реалізації сервера на io і nio суттєво відрізняється, тому буз використання фреймворку, як то кажуть підхід треба обирати “на березі”, адже в готовому рішені доведеться переписати дуже багато коду через невідповідність пакету і відсутність загального інтерфейсу. Та ж сама проблема при змінні протоколу в програмі

Netty має універсальний асинхронний інтерфейс вводу/виводу під назвою Channel, який абстрагує всі операції, необхідні для зв’язку «точка-точка». Тобто, як тільки ви написали свою програму на одному транспорті Netty, ваша програма може працювати на інших транспортах Netty. Netty надає низку важливих транспортів через один універсальний API:

- NIO (Non-blocking I/O): Використовує пакет java.nio.channels як основу. Реалізує неблокуюче введення-виведення, базуючись на селекторах. Універсальний і крос-платформений.
- Epoll (Linux-specific Non-blocking I/O): Використовує JNI для доступу до системного виклику epoll() та неблокуючого введення-виведення. Доступний лише для операційних систем Linux. Швидший за NIO у продуктивних середовищах. Підтримує специфічні функції Linux, такі як SO_REUSEPORT. Рекомендований для використання у продуктивних середовищах на Linux.
- OIO (Old Blocking I/O): Заснований на пакеті java.net. Використовує блокуючі потоки для роботи з введенням-виведенням. Застарілий і менш ефекти-

вний підхід. Використовується дуже рідко у сучасних додатках. Підходить, де блокування не є критичним.

- Місцевий (Local Transport): Локальний транспорт, який забезпечує комунікацію всередині однієї JVM. Використовується для оптимізації операцій у межах однієї програми. Підходить для сценаріїв, де не потрібна мережева передача.

Перемикання з одного транспорту на інший зазвичай займає лише кілька рядків змін, таких як вибір іншої ChannelFactory реалізації.

Крім того, ви навіть можете скористатися перевагами нових транспортів, які ще не написані (наприклад, транспорт зв’язку через послідовний порт), знову ж таки, замінивши лише пару рядків викликів конструктора. Крім того, ви можете написати власний транспорт, розширивши основний API.

Чітко визначена та розширювана модель подій є обов’язковою для програм, керованих подіями. Netty має чітко визначену модель подій, зосереджену на I/O. Це також дозволяє реалізувати власний тип події, не порушуючи існуючий код, оскільки кожен тип події відрізняється від іншого суворою ієрархією типів. Це ще одна відмінність від інших фреймворків. Багато фреймворків NIO не мають або мають дуже обмежене поняття моделі подій. Якщо вони взагалі пропонують розширення, вони часто порушують існуючий код, коли ви намагаєтеся додати спеціальні типи подій.

Коду при використанні Netty стає значно менше, щоб переконатись написав невеликий сервер, що матиме стандартну архітектуру програми написаної на Netty. Сервер буде мати певний пул потоків, що оброблюватиме роботу з сокетами на мережевому рівні і також окремий тред пул для обробки логіки перехоплювачів, які будуть читати та декодувати сповіщення і повертати якусь відповідь рис. 2.15.

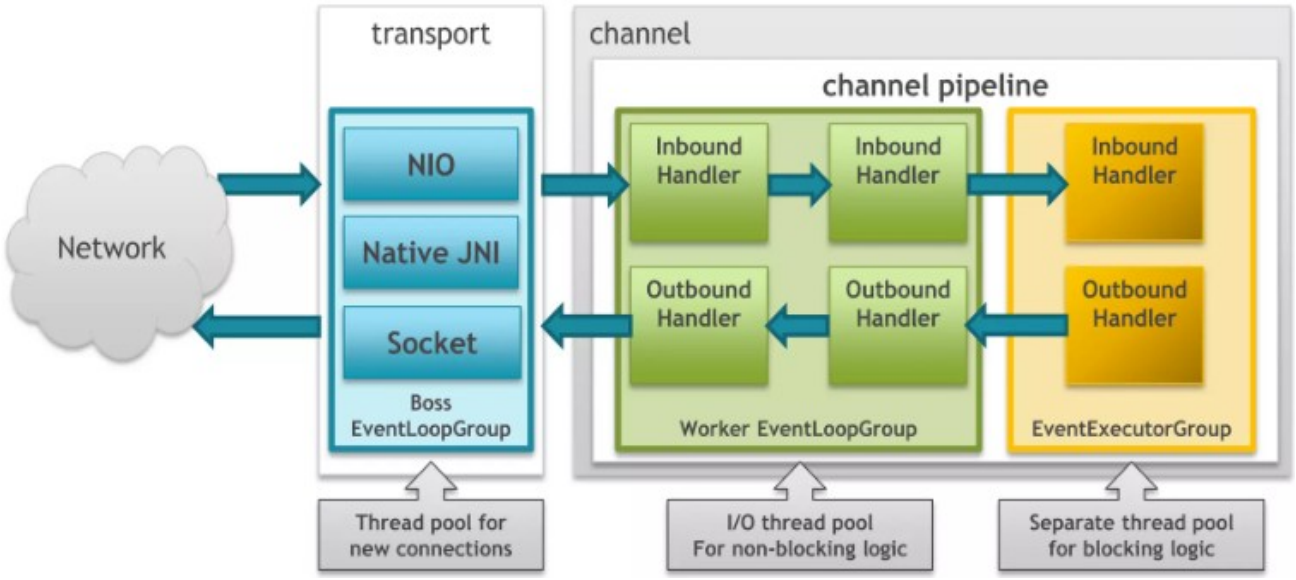


Рисунок 2.15 Приклад серверної архітектури побудованої на nio

Спочатку напишемо сервер. Першим кроком напишемо код, щоб сервер успішно стартував: “ДОДАТОК А”.

Пояснення:

1. Створив обробник EchoServerHandler що перевизначає методи з ChannelInboundHandlerAdapter. Хендлер читає дані, що отримав від клієнта і відправляє теж саме повідомлення назад на клієнта. Також хендлер вказує, що коли канал з’єднання буде прочитаний, то потрібно очистити буфер і закрити конекшин.
2. Вказав яким чином оброблювати запити, вказав неблокуючийц підхід new NioEventLoopGroup();
3. Далі створив new ServerBootstrap() - це заміняє дозволяє постійно слухати Channel.
4. Додав порт запуску нашого серверу
5. Додав хендлер

Після створення серверу, потрібно створити клієнта “ДОДАТОК В”.

Пояснення:

1. Вказав яким чином оброблювати запити, вказав неблокуючийц підхід new NioEventLoopGroup();
2. Створив new Bootstrap() - це клієнт сервера, він працює запускається, робить запит і зупиняє роботу, якщо з’єднання відсутнє.
3. Вказав адресу сервера

4. Створив new EchoClientHandler(). Цей перехоплювач відправляє повідомлення на сервер, як тільки з’являється конекшин і чекає на відповідь.

Для того, щоб клієнт або сервер працювали io замість nio ми можемо використовувати: OioEventLoopGroup і OioServerSocketChannel класи замість NioEventLoopGroup і NioServerSocketChannel відповідно.

Отже, розглянувши фреймворк Netty можна переконатись в його перевагах над написанням коду на джава без фреймворку. Також як можна буде переконатись далі, Netty є надзвичайно важливим для подальшої побудови більш високорівневих реактивних фреймворків.

2.3.3 Project Reactor і Webflux

Найбільш популярний фреймворк на Java - це Spring. Він містить у собі купу різних компонентів для різних задач. Коли Spring з’явився на ринку це значно спростило і пришвидшило розробку сервісів. Звичайно Spring містить компоненти, які використовуються для реактивного програмування Java сервісів. Навіть, варто сказати, що Spring має окремий проєкт, який займається реактивним програмуванням і який включає в себе купу різних компонентів. Цей проєкт називається Project Reactor.

Проект Reactor складається з набору модулів, перерахованих у документації Reactor. Основні модулі:

- Reactor Core - основний модуль, який є ядром Project Reactor і його включають в себе всі інші компоненти.
- Reactor Test - надає деякі утиліти для тестування реактивних потоків
- Reactor Extra - надає деякі додаткові оператори Flux
- Reactor Netty - неблокуючі клієнти та сервери TCP, HTTP та UDP з підтримкою зворотного тиску - на основі інфраструктури Netty
- Reactor Adapter - адаптер для інших реактивних бібліотек, таких як RxJava2 та Akka Streams
- Reactor Kafka - реактивний API для Kafka, який дозволяє публікувати та отримувати повідомлення в Kafka.

Розгляну більш детально Reactor Core компонент.

Reactor Core - це ядро бібліотеки Project Reactor, яка є одним із основних інструментів для реалізації реактивного програмування в екосистемі Java. Project Reactor базується на принципах реактивного програмування та стандарті Reactive Streams, забезпечуючи інструменти для побудови асинхронних, неблокуючих, високопродуктивних додатків.

Reactor Core створено для побудови реактивних потоків даних. Він забезпечує:

- Асинхронність — обробка даних відбувається в іншому потоці, без блокування основного потоку виконання.
- Неблокуюча модель — реактор використовує неблокуючий API, що дозволяє обробляти тисячі запитів одночасно, не вичерпуючи ресурси.
- Реактивні потоки — інтеграція з іншими бібліотеками через стандарт Reactive Streams.

Reactor Core надає два основних типи, які моделюють реактивні потоки даних: Mono і Flux.

Mono<T> - це реактивний тип, який представляє 0 або 1 елемент. Використовується, коли є необхідність обробляти одиничний результат або його відсутність рис. 2.16.

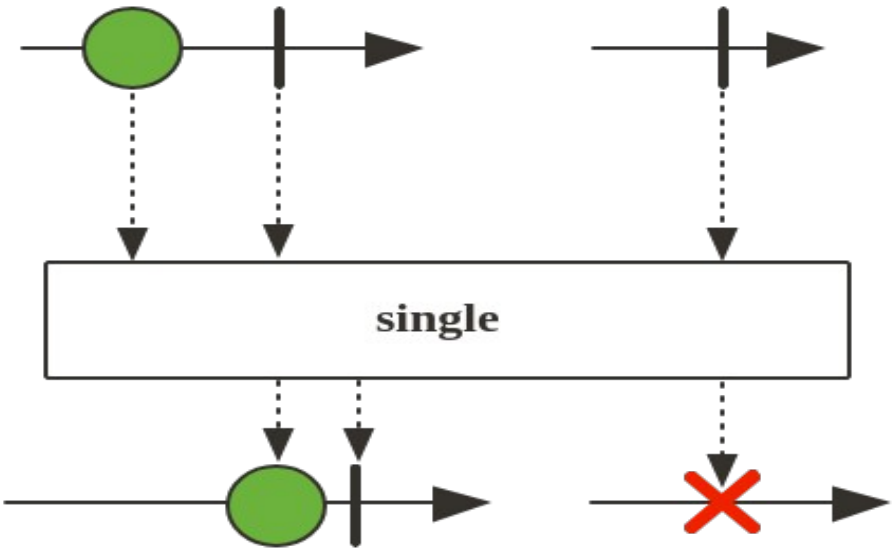


Рисунок 2.16 Інтерфейс Mono

Flux<T> - це реактивний тип, який представляє потік з 0 або більше елементів. Використовується для роботи з багатьма значеннями рис. 2.17.

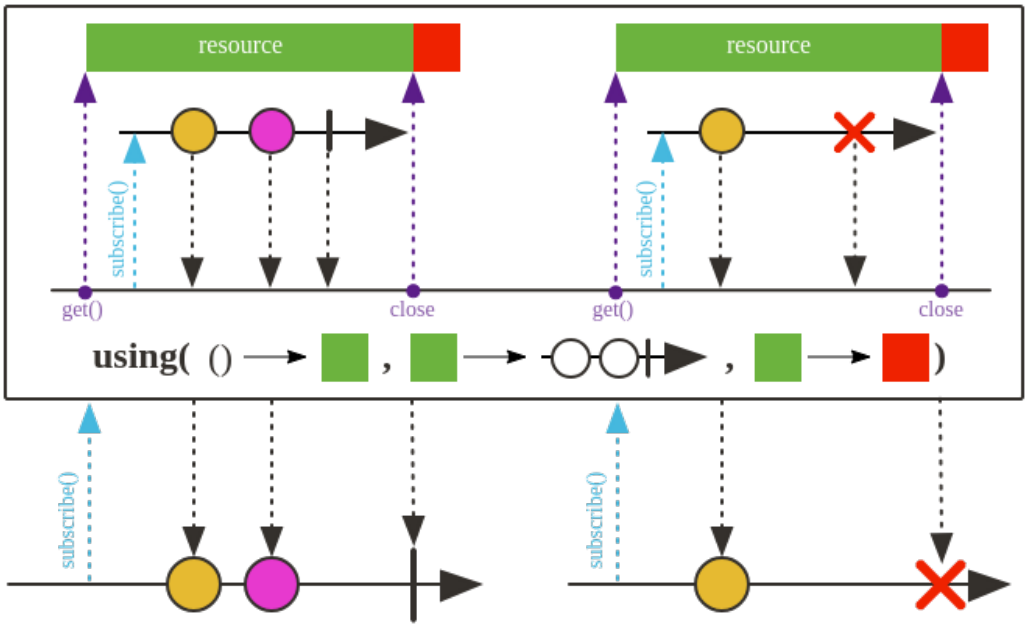


Рисунок 2.17 Інтерфейс Flux

Обидва вони завершуються або сигналом завершення, або помилкою, і вони викликають методи onNext, onComplete і onError нижче розташованого передплатника. Крім реалізації функцій, описаних у специфікації Reactive Streams, Flux та Mono надають набір операторів для підтримки перетворень, фільтрації та обробки помилок.

Існування Mono окремо від Flux у Reactor Core обумовлене різними сценаріями використання цих типів, а також спрощенням коду, підвищенням його читабельності та ефективності. Хоча Flux теоретично може обробляти і випадки з одним або жодним елементом, використання Mono має низку важливих переваг:

- Семантична ясність: mono чітко сигналізує, що результатом реактивного потоку буде тільки один елемент або його відсутність. Це дозволяє програмістам краще розуміти поведінку методу. При використанні Flux для таких сценаріїв можна сплутати потік із множинними елементами, що ускладнить розуміння коду.
- Оптимізація ресурсів: mono використовує оптимізовані внутрішні механізми, оскільки воно працює з простішим потоком (0 або 1 елемент). Flux, що підтримує множинні елементи, має більший накладний код для управління потоками даних.
- Зручність API: операції, доступні в Mono, зосереджені на роботі з одним елементом або відсутністю елемента, що спрощує використання API.

- Відсутність або помилка: mono дозволяє легко обробляти відсутність результату (empty) або помилки, що типово для запитів із потенційно відсутнім значенням.

Розглянемо головні оператори цих інтерфейсів, більшість з них повторюються як для Mono так і для Flux.

Для створення реактивного потоку можна використовувати рис. 2.18:

Flux.fromIterable(Arrays.asList(1,2,3));

або Flux.just(1,2,3);

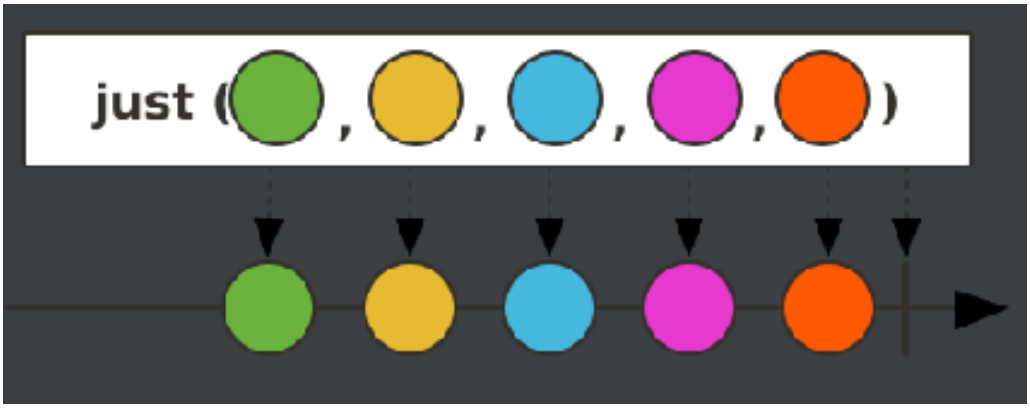


Рисунок 2.18 Відображення роботи методу just()

Flux та Mono - lazy.

Для того щоб запустити якусь обробку та скористатися даними, що лежать у Mono та Flux, потрібно на них підписатися за допомогою subscribe() рис. 2.19.

Flux.just(1,2,3).subscribe(System.out::println);

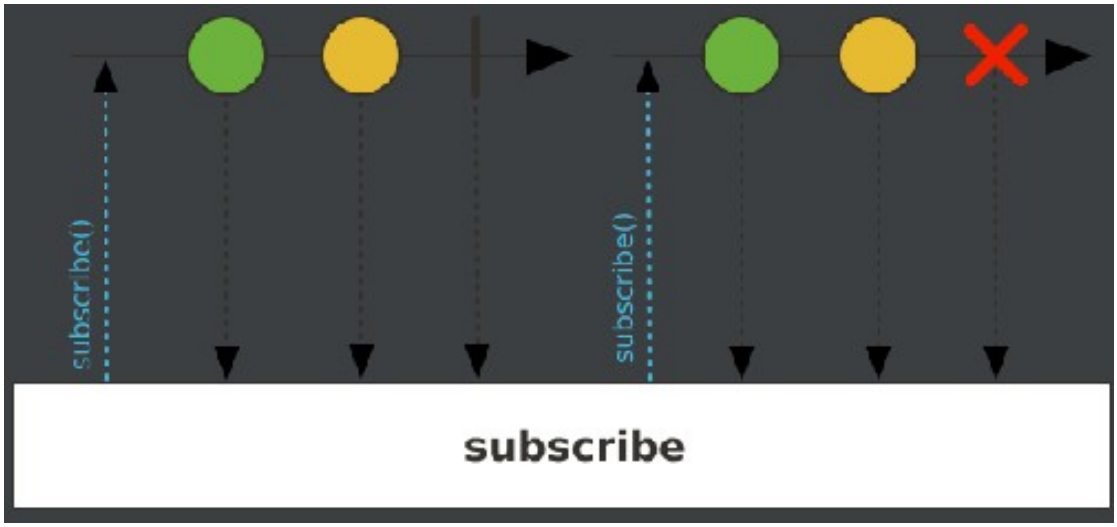


Рисунок 2.19 Відображення роботи методу subsribe()

Дані передаються в такий спосіб:

1. Викликається метод subscribe()

2. Потім створюється Subscription об'єкт
3. Після цього Subscriber викличе request() метод у Subscription класі, щоб вказати кількість об'єктів, які може обробити (якщо цей метод не викликається явно, запитується необмежену кількість об'єктів).
4. Subscriber може отримувати об'єкти за допомогою методу onNext() . Якщо передплатник отримує всі запитані об'єкти, він може запросити додаткові об'єкти або скасувати передплату, викликавши onComplete() . Якщо в якийсь момент виникає помилка, видавець викликає метод onError() на передплатнику.

Дані не почнуть надходити, доки ми не підпишемося — метод subscribe().

Коли ми викликаємо передплату, під капотом викликається Subscription , який запитує елементи з потоку (це означає, що він запитує кожен доступний елемент). Цей потік описується в інтерфейсі Subscriber як частина специфікації реактивних потоків, і фактично те, що було реалізовано за лаштунками в нашому виклику методу onSubscribe().

Наступний код:

```
List<Integer> elements = new ArrayList<>();
Flux.just(1,2,3,4).log().subscribe(elements::add);
```

Цей код видасть наступні логи:

```
reactor.Flux.Array.1 | onSubscribe([Synchronous Fuseable]
FluxArray.ArraySubscription)
reactor.Flux.Array.1 - | request(unbounded)
reactor.Flux.Array.1 - | onNext(1)
reactor.Flux.Array.1 - | onNext(2)
reactor.Flux.Array.1 - | onNext(3)
reactor.Flux.Array.1 - | onNext(4)
reactor.Flux.Array.1 | onComplete()
```

Де можна побачити як працює потік під капотом:

- onSubscribe() — викликається , коли ми підписуємося на потік

- request(unbounded) - викликається до часу ми викликаємо підписку (subscribe) , за лаштунками ми створюємо Subscription . Ця підписка запитує елементи потоку. І тут він запитує кожен доступний елемент.
- onNext() - викликається на кожному елементі
- onComplete() - викликається останнім, після отримання останнього елемента.

Нам завжди потрібно оператори для створення і підписки на потік. Між цими командами можуть бути будь-які інші, наприклад, ті що ми можемо зустріти в звичайних Java стрімах: map(), filter(), тощо. Також є особливі оператори, які є тільки в реакторі і які допомагають реагувати на події, що є головною характеристикою реактивності.

Оператори для зміни даних у потоці:

map(Function<T, R>) — перетворює кожен елемент потоку.

Flux.just(1, 2, 3).map(i -> i * 2); // 2, 4, 6

flatMap(Function<T, Publisher<R>>) - перетворює елемент у інший потік і об'єднує результати.

Flux.just(1, 2, 3).flatMap(i -> Flux.range(i, 2)); // 1, 2, 2, 3, 3, 4

buffer(int size) - групує елементи у списки заданого розміру.

Flux.range(1, 10).buffer(3); // [1, 2, 3], [4, 5, 6], [7, 8, 9], [10]

Оператори для відбору елементів із потоку:

filter(Predicate<T>) - алишає тільки ті елементи, які відповідають умові.

Flux.range(1, 10).filter(i -> i % 2 == 0); // 2, 4, 6, 8, 10

take(long n) - бере перші `n` елементів.

Flux.range(1, 10).take(3); // 1, 2, 3

skip(long n) - пропускає перші `n` елементів.

Flux.range(1, 10).skip(3); // 4, 5, 6, ..., 10

Оператори для об'єднання кількох потоків:

merge(Publisher<T>... sources) - об'єднує кілька потоків, дозволяючи їм працювати паралельно.

Flux.merge(Flux.just(1, 2), Flux.just(3, 4)); // 1, 2, 3, 4

concat(Publisher<T>... sources) - об'єднує кілька потоків послідовно.

Flux.concat(Flux.just(1, 2), Flux.just(3, 4)); // 1, 2, 3, 4

zip(Publisher<T>... sources) - об'єднує потоки, з'єднуючи елементи на одних і тих самих позиціях.

Flux.zip(Flux.just(1, 2), Flux.just("A", "B")); // (1, A), (2, B)

Оператори для обробки помилок у потоці:

onErrorResume(Function<Throwable, ? extends Publisher<T>>) - замінює помилку іншим потоком.

onErrorReturn(T fallback) - повертає значення замість помилки.

doOnError(Consumer<Throwable>) - виконує дію при виникненні помилки.

Додаткові оператори для контролю потоку або побічних ефектів:

doOnNext(Consumer<T>) - виконує дію для кожного елемента.

doOnComplete(Runnable) - виконує дію після завершення потоку.

delayElements(Duration) - додає затримку для кожного елемента.

timeout(Duration) - завершує потік із помилкою, якщо елементи не надходять вчасно.

Ці оператори дозволяють будувати потужні реактивні ланцюжки для будь-яких сценаріїв - від трансформації даних до обробки помилок та управління потоками.

Також важливо розуміти, що стріми в реакторі можуть вести себе по різному, можуть бути холдодним(дефолтна поведінка) і гарячими.

Холодні стріми - починають генерувати дані лише після підписки. Кожен підписник отримує свій унікальний потік даних. Дані генеруються заново для кожного підписника. Це означає, що кілька підписників отримають однакові дані, але у своїх потоках.

Напишемо невеличкий код:

Flux<Integer> coldStream = Flux.range(1, 5);

coldStream.subscribe(data->System.out.println("Subscriber 1: " + data));

coldStream.subscribe(data -> System.out.println("Subscriber 2: " + data));

Де результатом буде рис. 2.20:

```
Subscriber 1: 1
Subscriber 1: 2
Subscriber 1: 3
Subscriber 1: 4
Subscriber 1: 5

Subscriber 2: 1
Subscriber 2: 2
Subscriber 2: 3
Subscriber 2: 4
Subscriber 2: 5
```

Рисунок 2.20 Результат роботи холодних стрімів

Ми можемо побачити що обидва підписники отримали всі дані.

Гарячі стріми - дані починають генеруватися незалежно від підписників, як тільки стрім активовано. Всі підписники ділять один і той самий потік. Якщо підписник підключається пізніше, він отримає лише ті дані, які генеруються після його підключення.

Приклад: події в реальному часі: натискання клавіш, повідомлення з черг (Kafka, RabbitMQ), сенсори, стрімінгові дані.

Напишемо невеличкий код і для гарячих стрімів:

```
ConnectableFlux<Integer> hotStream = Flux.range(1, 5).publish(); // Робимо гарячий стрім
hotStream.connect(); // Запускаємо стрім
hotStream.subscribe(data -> System.out.println("Subscriber 1: " + data));
Thread.sleep(2000);
hotStream.subscribe(data -> System.out.println("Subscriber 2: " + data));
```

Де результатом буде рис. 2.21:

```
Subscriber 1: 1
Subscriber 1: 2
Subscriber 1: 3
Subscriber 1: 4
Subscriber 1: 5
Subscriber 2: 3
Subscriber 2: 4
Subscriber 2: 5
```

Рисунок 2.21 Результат роботи гарячих стрімів

Бачимо, що другий підписник отримав лише частину даних.

Іноді вам потрібно перетворити холодний стрім на гарячий або навпаки. Використовуйте методи на кшталт publish() або share() для перетворення холодного в гарячий. Перетворення з гарячого в холодний відбувається складніше: можна зберігати дані та передавати їх новим підписникам (наприклад, через ReplayProcessor).

Холодні стріми варто використовувати, коли кожен підписник має отримати повний набір даних. Гарячі стріми, коли дані в реальному часі мають транслюватися всім підписникам.

Так, як я планую побудувати реактивний веб застосунок не обійтися без компонента під назвою WebFlux. 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.

Додати в спрінг проект можна за допомогою залежності - spring-boot-starter-webflux. Якщо додати цю залежність і подивитись дерево залежностей можна побачити netty і reactor-core рис. 2.22.

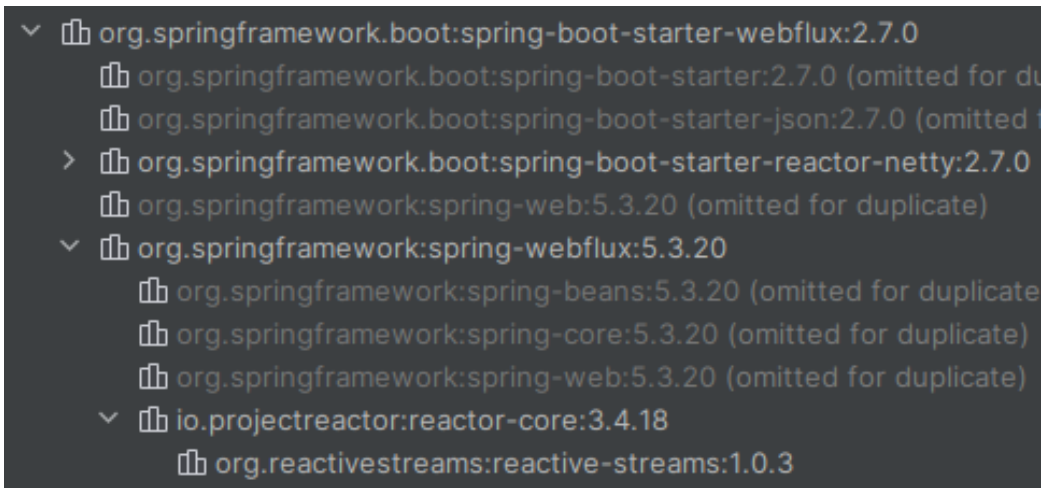


Рисунок 2.22 Дерево залежностей Webflux

Розробка коду REST клієнта на WebFlux не сильно відрізняється від того ж MVC, тому що ми можемо використовувати той самий API. Це є перевагою, адже можна перейти від звичайного програмування на реактивне.

Отже, Spring WebFlux - хороший вибір для проектів, які потребують високої масштабованості та ефективності. Однак для його навчання може знадобитися трохи більше зусиль і часу. Для проектів, які не потребують великої масштабованості або не передбачають одночасної обробки великого навантаження, простіший вибір, як-от Spring MVC, може мати більше сенсу.

2.3.4 Vert.x

У сучасній розробці програмного забезпечення важливим завданням є ефективне використання обчислювальних ресурсів, пам’яті, тощо. Це зумовлює пошук нових рішень. Одним із таких рішень є платформа Vert.x.

Vert.x - це реактивна платформа, яка дозволяє створювати високопродуктивні, масштабовані та неблокуючі додатки на базі JVM. Vert.x вирізняється серед інших рішень своєю гнучкістю, мультиплатформенністю та підтримкою різних мов програмування, що робить його універсальним вибором для розробників. Завдяки своїй подійно-орієнтованій архітектурі та підтримці реактивного підходу, Vert.x знаходить застосування в побудові мікросервісів, реального часу API, веб-додатків і розподілених систем. При розробці Vert.x велике значення приділили “легкості” цієї бібліотеки. Як з точки зору використання розробниками, так і по використанню ресурсів.

Насправді Vert.x надає цілу екосистему для розробки реактивних систем, основні це:

- Vert.x core - ядро і головний API
- Vert.x web - надає можливість використання високорівневих мережевих протоколів.
- Vert.x unit - бібліотека для тестування
- Vert.x auth - бібліотека для роботи з налаштуванням безпеки і прав клієнтів.

Я розберу більш детально дві основні бібліотеки це Vert.x core і Vert.x web, які я планую використати в майбутньому проекті.

Функціональність ядра досить обмежена - тут ви не знайдете таких речей, як доступ до бази даних, авторизація чи веб-функціональність високого рівня - такі речі можна додати за допомогою інших бібліотек. Це зроблено навмисно, щоб ядро важило як умога менше пам’яті.

Ядро Vert.x невелике та легке. Ви просто використовуєте ті частини, які хочете. Його також можна повністю вбудувати у ваші існуючі програми - ми не змушуємо вас структурувати свої програми особливим чином, щоб ви могли використовувати Vert.x.

Для початку розберу, на яких паттернах і ідеях побудований Vert.x.

Першим паттерном є Multi-Reactor. Щоб зрозуміти патерн Multi-Reactor, достатньо знати відомий патерн Reactor. Класичний Reactor говорить про те, що є якийсь 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 довгі завдання делегуються Worker-потокам і виконуються, не блокуючи Event Loop. На них повішено певний Callback, який буде викликаний, як тільки завдання буде виконане. Колбеки (callbacks) - це функції або методи, які передаються як параметри до іншої функції та викликаються після завершення певної операції.

У свою чергу, Multi-Reactor розширює цей паттерн, додаючи ще кілька потоків (додаткові Event Loop-и) рис. 2.23.

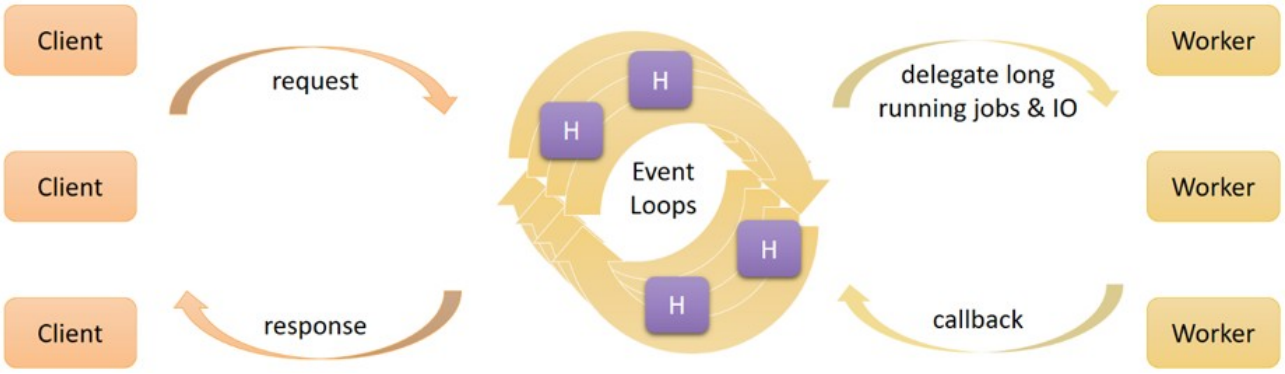


Рисунок 2.23 Паттерн Multi-Reactor

Наступним паттерном є EventBus або шина повідомлень.

Патерн Event Bus - це архітектурний підхід для обміну повідомленнями між різними компонентами програми, які можуть бути слабко зв'язані. Основна ідея цього патерну - створення централізованого каналу (шини), через який події та повідомлення передаються між відправниками (publishers) і споживачами (subscribers). EventBus у Vert.x є надзвичайно важливим компонентом, адже черз нього дані потрапляють до потрібних хендлерів, балансуються між декількома однаковими інстансами хендлерів. Також шинуу подій використовують як локально в рамках однієї JVM так і в рамках різних JVM. Тобто одною шиною можна зв’язати хендлери на двох різних віддалених машинах, а також абсолютно різні сервіси на різних мовах програмування, які використовують Vert.x рис. 2.24.

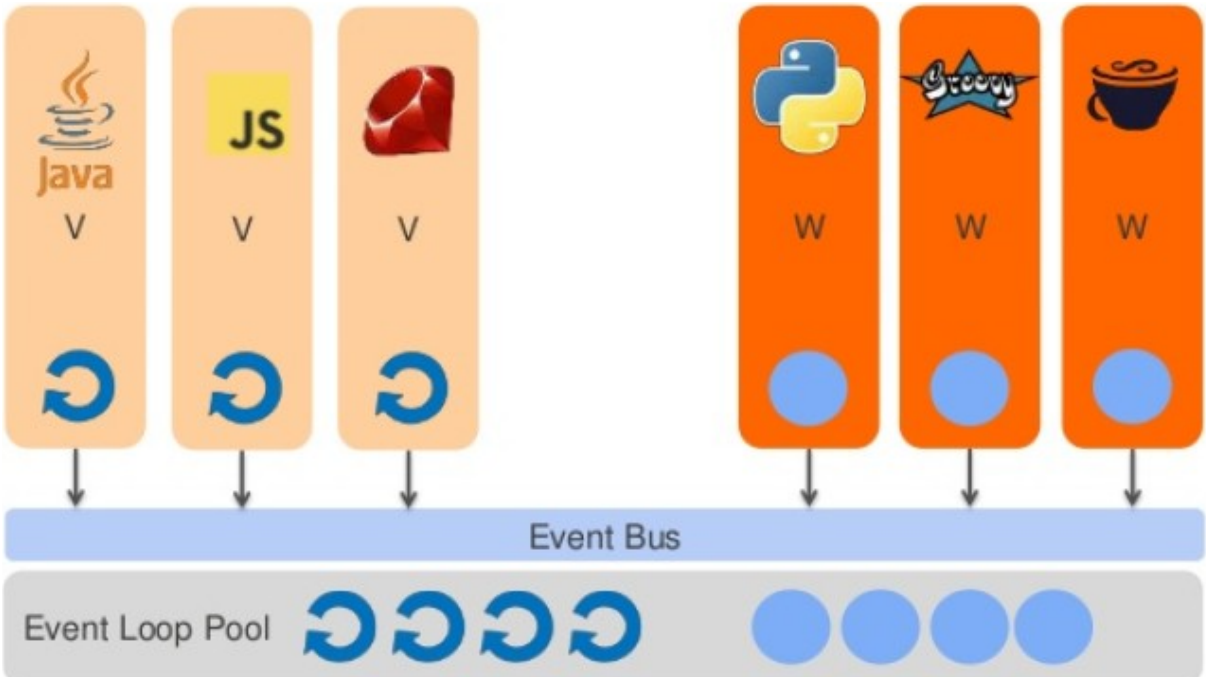


Рисунок 2.24 Паттерн EventBus

Наступним підходом є контейнеризація. Коли чцємо контейнеризацція одразу на думку спадає докер, за допомогою якого можна запустити декілька інстансів з одного екземпляру коду. Vert.x звичайно не заміняє і не виконує функції докера, але він також має можливість масштабування. Це реалізовано за допомогою класу Verticle. По суті вертікл це клас обгортка над хендлерами, яка є абстракцією між vert.x і транспортним рівнем і джава кодом. Вертікл працює на EventLoop. Може бути ситуація, коли кількість EventLoop менша за кількість вертиклів. Тоді декілька вертиклів може працювати на одному циклі подій, але це буде працювати синхронно, без переривання контексту(спочатку виконався один вертікл, потім другий).

Отже, загальна архітектура Vert.x виглядатиме так рис. 2.25:

Рисунок 2.25 Загальний вигляд архітектури Vert.x

1. Події розподіляються через Event Bus, який слугує центральним каналом комунікації між різними компонентами системи.

2. Кожен потік обробляє події для одного або кількох вертіклів (Verticles).
3. Вертікли A, B, C відповідають за асинхронну обробку завдань. Цикли подій працюють у неблокуючому режимі, тому завдання обробляються дуже швидко. Verticle A приймає подію, обробляє її та передає результат у Event Bus.Verticle, Verticle B і Verticle C можуть обробляти події паралельно, виконуючи різні функції.
4. Коли необхідно виконати блокуючу операцію (наприклад, взаємодія з базою даних), Vert.x використовує метод executeBlocking(). Ця задача передається у Default Worker Thread Pool, який гарантує, що блокуючі операції не блокують основні потоки подій.
5. Worker Verticles(вертікли-працівники) - використовуються для виконання тривалих і блокуючих завдань. Наприклад, якщо обробка події потребує більше часу або взаємодії з I/O-системами, завдання передається до Worker Verticle. Worker Verticle має власний пул потоків, що дозволяє виконувати завдання паралельно, не впливаючи на основні цикли подій.

Важливо також розуміти, що Vert.x зазвичай працює поверх netty. Алгоритм інтеграції netty з Vert.x відбувається наступним чином:

1. Netty через BossEventLoopGroup приймає нове TCP-з'єднання.
2. З'єднання передається у WorkerEventLoopGroup для обробки.
3. Декодери (наприклад, HttpRequestDecoder) перетворюють сирі байти в об'єкти HTTP.
4. Vert.x додає свій обробник у ChannelPipeline, щоб передати дані у Verticles.
5. Подія передається у відповідний Verticle.
6. Verticle обробляє логіку (наприклад, бізнес-правила) і може відправити відповідь у EventBus, звідки подія потрапить до наступних вертиклів в рамках однієї JVM.
7. Vert.x передає результати обробки назад у Netty через ChannelPipeline.
8. Netty кодує вихідні дані (через HttpResponseEncoder) і відправляє клієнту.

Vert.x Web - це модуль для Vert.x, який надає додаткові високорівневі функції для створення веб-додатків. Він розширює базову функціональність Vert.x

HttpServer, додаючи підтримку маршрутизації, роботи з запитами/відповідями, а також інші корисні інструменти для веб-розробки.

Основний функціонал Vert.x Web:

- Маршрутизація: Vert.x Web надає зручний API для створення маршрутизаторів, які обробляють HTTP-запити на основі URL-шляхів, методів і інших критеріїв.
- Підтримка Middleware: Vert.x Web підтримує використання middleware для обробки запитів перед тим, як вони досягнуть основного маршруту. Це дозволяє реалізовувати авторизацію, логування, або інші операції на рівні запитів:
- Обробка статичних ресурсів: ви можете легко обслуговувати статичні файли (HTML, CSS, JS) зі свого додатка.
- Шаблонізація: підтримка різних двигунів шаблонів, таких як FreeMarker, Thymeleaf, Handlebars, Jade тощо. Дозволяє легко створювати динамічні HTML-сторінки.
- Аутентифікація та авторизація: підтримка різних механізмів аутентифікації: Oauth2, JWT, Basic Authentication.

3 ПОРІВНЯННЯ РЕАКТИВНИХ ФРЕЙМВОРКІВ: WEBFLUX І VERT.X

3.1 Критерії оцінювання

Основною ідеєю цієї частини роботи і в цілому всієї дипломної роботи є порівняння двох інструментів для створення реактивного програмного забезпечення - WebFlux і Vert.x. Вони мають багато схожостей, але і багато відмінностей. Власне вони були створенні з різними ідеями і підходами. Саме тому неможливо порівняти ці два інструменти без чітко визначених критеріїв оцінювання. Критерії оцінювання, що будуть наведенні далі є виключно суб’єктивними і придумані мною виходячи з мого власного досвіду.

Отже, найважливішим критерієм реактивних фреймворків особисто для мене є продуктивність. Цей критерій є найважливішим, адже зазвичай, коли ми говоримо про реактивні системи - спадає на думку велике навантаження. Але не все так просто, адже в це поняття я включаю декілька підпунктів:

- Пропускна здатність: кількість запитів/секунд (RPS) фреймворк може обробити.
- Час відповіді (Latency): середній і в різних персентилях(зазвичай важливі p95, p99). Важливо оцінити, наскільки стабільний фреймворк під навантаженням.
- Час очікування: скільки часу запити проводять у черзі перед обробкою.
- Споживання ресурсів: використання CPU та RAM під час навантаження.

Багато речей можуть піти не так, коли система перебуває під навантаженням. Система повинна виконувати багато операцій одночасно і відповідати на різні запити від різної кількості користувачів. Щоб підготуватися до цих ризиків продуктивності, команди використовують навантажувальне тестування.

Але хороша стратегія навантажувального тестування вимагає більше, ніж просто виконання одного сценарію. Різні моделі трафіку створюють різні профілі ризику для програми. Для комплексної підготовки команди повинні перевірити систему на різних типах тестів рис. 3.1:

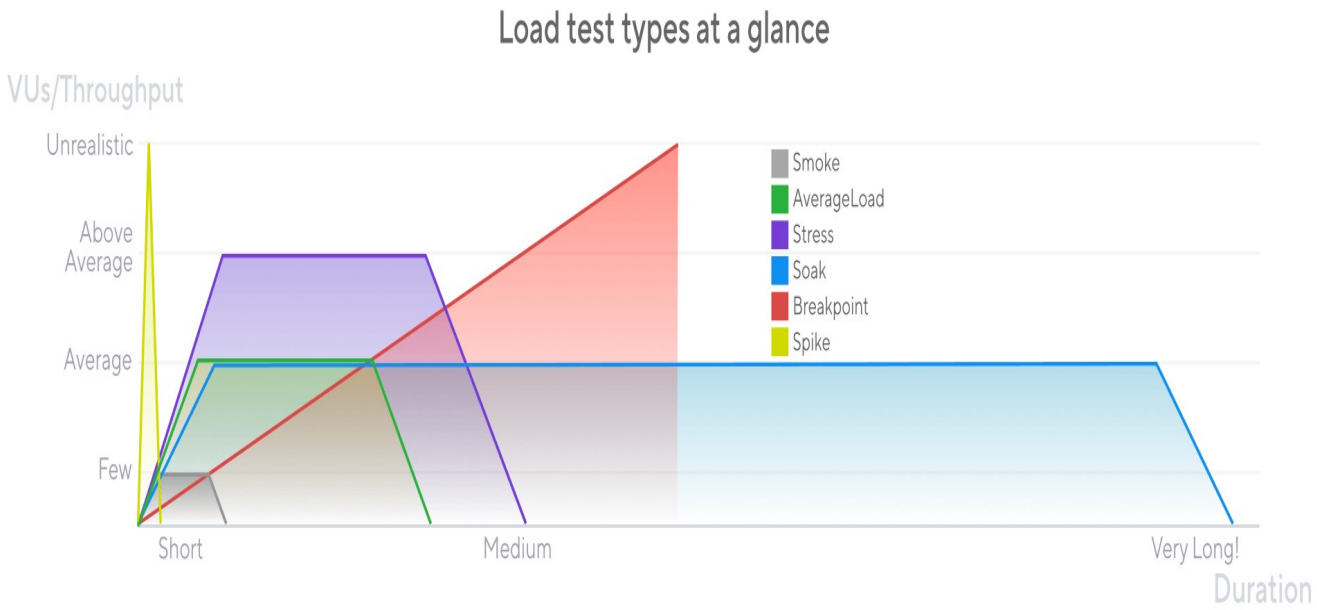


Рисунок 3.1 Типи тестів навантаження

- Димові тести підтверджують, що ваш сценарій працює та що система працює адекватно за мінімального навантаження.
- Тест із середнім навантаженням оцінює, як ваша система працює за очікуваних нормальних умов.
- Стрес-тести оцінюють, як система працює на своїх межах, коли навантаження перевищує очікуване середнє.
- Випробовування на витримку оцінюють надійність і продуктивність вашої системи протягом тривалого часу.
- Спикові тести підтверджують поведінку та виживання вашої системи у випадках раптового, короткочасного та значного збільшення активності.
- Тести контрольних точок поступово збільшують навантаження, щоб визначити межі пропускної здатності системи.

Наступним важливим критерієм є функціональні можливості. Я маю на увазі наскільки добре продумана архітектура і написана екосистема. Цей пункт поглинає багато інших:

- Можливість обробки помилок: наскільки гнучко фреймворк дозволяє обробляти помилки на різних рівнях: на рівні запитів, потоків або глобально. Реактивні фреймворки повинні підтримувати механізми, які запобігають збоям системи під час критичних помилок.

- Можливість масштабуватись: наскільки добре фреймворк підтримує масштабування компонентів у межах одного вузла та підтримка масштабування системи на кілька вузлів.
- Підтримка backpreassure.
- Інтеграція з іншими інструментами і технологіями: базами даних, брокерами повідомлень, протоколи, тощо.

Наступним не менш важливим критерієм є досвід розробки. Цей критерій також можна розділити на декілька підпунктів: швидкість розробки, легкість освоєння, рівень розвитку ком’юніті та кількість ресурсів.

3.2 Допоміжні інструменти для порівняння фреймворків

Порівняння WebFlux і Vert.x виявилось не зовсім тривіальною задачею, тому мені довелось використати додаткові інструменти: K6, Docker, Kubernetes, GitHub, IntellijIdea.

3.2.1 IntelliJ IDEA

IntelliJ IDEA - це одна з найпопулярніших інтегрованих середовищ розробки (Integrated Development Environment, IDE) для програмування на Java та інших мовах. Вона була розроблена компанією JetBrains та з моменту випуску першої версії у 2001 році стала стандартом у галузі розробки програмного забезпечення завдяки своїм розширеним функціональним можливостям, зручному інтерфейсу та підтримці широкого спектру технологій.

IntelliJ IDEA вирізняється серед інших IDE такими ключовими особливостями:

- Розумна підтримка коду: IntelliJ IDEA забезпечує автоматичне доповнення коду, перевірку синтаксису, статичний аналіз і пропозиції щодо рефакторингу. Це значно підвищує продуктивність розробників.
- Інтеграція з системами контролю версій: IDE підтримує популярні системи контролю версій, такі як Git, SVN та Mercurial. Інтеграція дозволяє легко

виконувати операції коміту, злиття, перегляду історії та вирішення конфліктів.

- Підтримка багатьох мов: окрім Java, IntelliJ IDEA підтримує такі мови програмування, як Kotlin, Scala, Python, JavaScript, TypeScript, та інші. Це дозволяє розробникам використовувати одну IDE для різних проектів.
- Вбудовані інструменти для роботи з базами даних: IntelliJ IDEA містить функціонал для підключення до баз даних, написання SQL-запитів та перегляду структури даних безпосередньо з IDE.
- Підтримка фреймворків і технологій: середовище має потужну підтримку популярних фреймворків, таких як Spring, Hibernate, Vert.x, а також інструментів для створення веб-додатків, зокрема Java EE.
- Вбудована система тестування: IDE підтримує JUnit, TestNG та інші фреймворки для автоматизації тестування, що спрощує розробку програмного забезпечення з високою якістю коду.
- Плагіни: IntelliJ IDEA має велику кількість плагінів, які дозволяють розширити функціональність середовища для підтримки специфічних інструментів чи мов програмування.

IntelliJ IDEA - це потужний інструмент для розробки програмного забезпечення, який значно спрощує процес розробки, тестування та підтримки програмного коду. У межах дипломного проекту використання цієї IDE дозволило підвищити ефективність роботи, зменшити кількість помилок і забезпечити структурованість процесу розробки.

3.2.2 GitHub

GitHub - це популярна платформа для хостингу репозиторіїв, заснованих на системі контролю версій Git. GitHub залишається найбільш популярною завдяки своїй універсальності, розвинутій екосистемі та широкій підтримці спільноти.

Чому обирають GitHub? GitHub має низку переваг, які роблять його більш привабливим для багатьох команд розробників:

- GitHub є найпопулярнішою платформою серед розробників, що робить його стандартом де-факто для спільних проектів. Завдяки цьому нові розробники

мають високу ймовірність уже володіти навичками роботи з GitHub, що спрощує адаптацію до нових проектів.

- GitHub має безліч інтеграцій з популярними сервісами, такими як Jira, Slack, Trello, IntelliJ IDEA та інші. Інструмент GitHub Actions дозволяє автоматизувати CI/CD процеси без необхідності використовувати сторонні рішення.
- На GitHub розміщено понад 330 мільйонів відкритих репозиторіїв, включаючи проекти великих компаній, таких як Microsoft, Google, та інших. Це створює можливість для легкого пошуку бібліотек, відкритого коду та участі у відомих open-source проектах.
- Інтерфейс GitHub інтуїтивно зрозумілий і зручний навіть для новачків. Функціонал Pull Requests та Reviews дозволяє легко контролювати якість коду у великих командах.
- GitHub пропонує безкоштовний тарифний план для необмеженої кількості відкритих репозиторіїв із базовими функціями. Це робить його ідеальним для навчальних проектів, open-source ініціатив та невеликих стартапів.
- Інструмент на основі штучного інтелекту, який допомагає писати код, прогнозуючи наступні дії розробника. GitLab не має прямого аналога цього функціоналу.

Особисто я обрав саме це сховище віддалених репозиторіїв через декілька причин. Я вже маю досвід працювати з GitHub, навідмінну від інших. Вже маю прив’язаний аккаунт Copilot. GitHub має величезну кількість готових рішень, які можна взяти за шаблон.

3.2.3 Docker і Kubernetes

Docker - це відкрита платформа для розробки, доставки та запуску додатків у контейнерах. Він став стандартом де-факто у світі розробки завдяки своїй здатності вирішувати численні проблеми, пов'язані з сумісністю середовищ, розгортанням і масштабуванням.

Docker використовується для ізоляції додатків у контейнерах, що мають усі необхідні залежності, конфігурації та бібліотеки. Це дозволяє забезпечити стабільну роботу додатків у будь-якому середовищі. Основні переваги Docker:

- Портативність: контейнери працюють однаково на будь-якій платформі - від локального комп'ютера розробника до хмарного сервісу або серверного середовища.
- Ефективність використання ресурсів: контейнери використовують спільне ядро операційної системи, що робить їх легшими і швидшими за віртуальні машини.
- Швидкість розгортання: завдяки використанню готових образів додатки запускаються за секунди, а не хвилини.
- Спрощення DevOps-процесів: Docker дозволяє створювати та тестувати додатки у середовищах, які повністю відповідають виробничим, знижуючи ризик.
- Масштабування: Docker забезпечує легке масштабування додатків завдяки використанню інструментів, таких як Docker Compose і Docker Swarm.

Docker складається з кількох ключових компонентів, кожен з яких грає важливу роль у його роботі, основні компоненти:

- Docker Engine - це основа Docker, яка включає: Daemon, CLI, API. Демон відповідає за створення, запуск і управління контейнерами. Консольний інтерфейс - інструмент для взаємодії з Docker Engine через команди. API надає програмний інтерфейс для управління Docker.
- Образи Docker (Docker Images) - це шаблони, які містять всі необхідні файли, залежності та інструкції для запуску контейнера. Образи є основою для створення контейнерів.
- Контейнери - це ізольовані середовища, створені на основі образів. Контейнер містить все необхідне для запуску додатка: код, бібліотеки, системні інструменти тощо.

Docker Hub - офіційний реєстр для зберігання та спільного використання образів Docker. Також існують приватні реєстри, які можна розгортати локально.

- Docker Compose - інструмент для управління багатоконтейнерними додатками. Він дозволяє описати конфігурацію додатків у файлі `docker-compose.yml` і легко запускати їх за допомогою однієї команди.

- Volumes - система для управління даними контейнерів. Volumes дозволяють зберігати дані поза межами життєвого циклу контейнера, забезпечуючи їхню стійкість.

Можна побачити, що докер надає велику кількість компонентів і функціоналу. Але мені для порівняння фреймворків потрібний більш зручний функціонал для масштабування додатків. І такий інструмент існує - це Kubernetes (k8s) - це оркестратор контейнерів, що автоматизує розгортання, масштабування та управління контейнеризованими застосунками. Основні завдання:

- Оркестрація контейнерів (Docker, containerd).
- Автоматичне масштабування.
- Моніторинг та самовідновлення.

Загальна схема роботи Kubernetes є рис. 3.2:

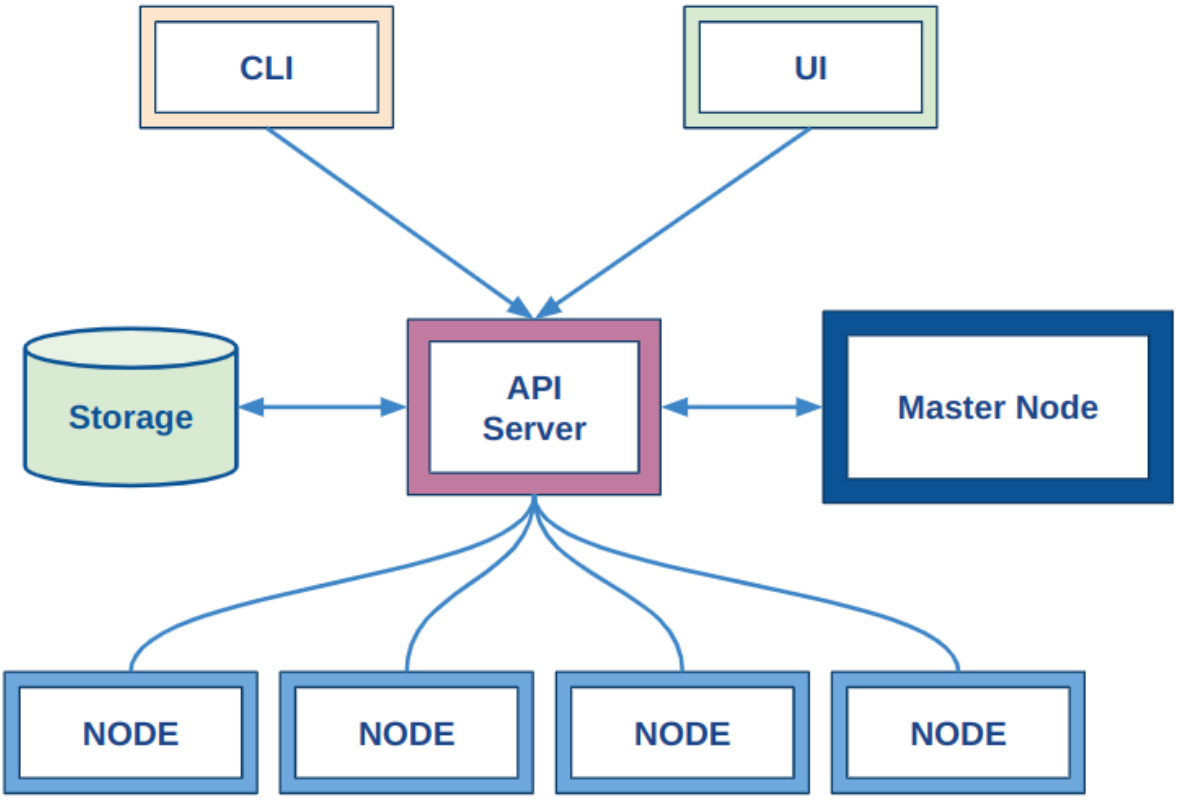


Рисунок 3.2 Загальна схема роботи Kubernetes

API Server - це основний компонент Control Plane у Kubernetes. Він виступає єдиною точкою входу для взаємодії з кластером: для користувачів, інструментів і компонентів. Обробляє запити від CLI, UI, і інших внутрішніх компонентів, взаємодіє з etcd для зберігання стану кластера, займається передачею команд на Nodes

для виконання через kubelet. API Server використовує RESTful API для управління ресурсами кластера.

CLI (Command Line Interface) - це інтерфейс командного рядка для взаємодії з API Server. Найчастіше використовується kubectl як основний CLI-інструмент для Kubernetes через який надсилаються запити до API Server для управління об'єктами Kubernetes: створення, оновлення, видалення. Також за допомогою операторів можна отримувати інформацію про стан кластера. Має більш розширений функціонал ніж UI.

UI - це графічний інтерфейс, який дозволяє взаємодіяти з Kubernetes кластером без командного рядка.

Storage відображає сховище для стану кластеру. У Kubernetes цим сховищем є etcd. Дані зберігаються у форматі ключ-значення. Будь-які зміни у кластері (наприклад, створення Pod'ів) реєструються в etcd через API Server.

Master Node - це вузол, що керує Kubernetes кластером. Усі компоненти Control Plane (API Server, Controller Manager, Scheduler, etcd) працюють на Master Node. Управляє розгортанням та масштабуванням подів та моніторить стан кластеру та прийняття рішень (наприклад, перенесення подів у разі збоїв). Також займається розподіленням завдань на Worker Nodes.

Worker Nodes - це робочі вузли, які виконують додатки у Kubernetes. На кожному Worker Node працюють такі компоненти:

- Kubelet: Компонент, який взаємодіє з API Server та запускає контейнери.
- Kube-proxy: Компонент, що управляє мережею та маршрутизує трафік між сервісами.
- Container Runtime: Виконує контейнери (наприклад, Docker або containerd).

Kubernetes цілий ряд об’єктів, які він ствою є і керує ними: pod, replicaSet, service, deployment, configMap, secret, ingress.

Pod - це найменша одиниця у Kubernetes, яка містить один або кілька контейнерів. Усі контейнери в подах працюють на одному вузлі(Node) та розділяють ресурси, такі як: мережевий простір та спільний файловий простір. Поди забезпечують ізоляцію між контейнерами.

Service - це абстракція мережі, що забезпечує стабільний доступ до подів через єдину IP-адресу. Поди є ефемерними та можуть змінюватися, але service забезпечує постійний доступ, незважаючи на зміни. Типи service:

- ClusterIP (за замовчуванням): доступ до подів тільки всередині кластера.
- NodePort: відкриває сервіс на портах вузлів, дозволяючи зовнішній доступ.
- LoadBalancer: використовується для зовнішнього доступу через балансувальник навантаження хмарного провайдера.
- ExternalName: Перенаправляє запит на зовнішнє DNS-ім'я.

Deployment - це об'єкт для управління розгортанням подів у кластері Kubernetes. Deployment дозволяє рис. 3.3:

- Масштабувати Pod'и (змінювати кількість реплік).
- Робити оновлення без простою (Rolling Update).
- Повертатися до попередніх версій (Rollback).

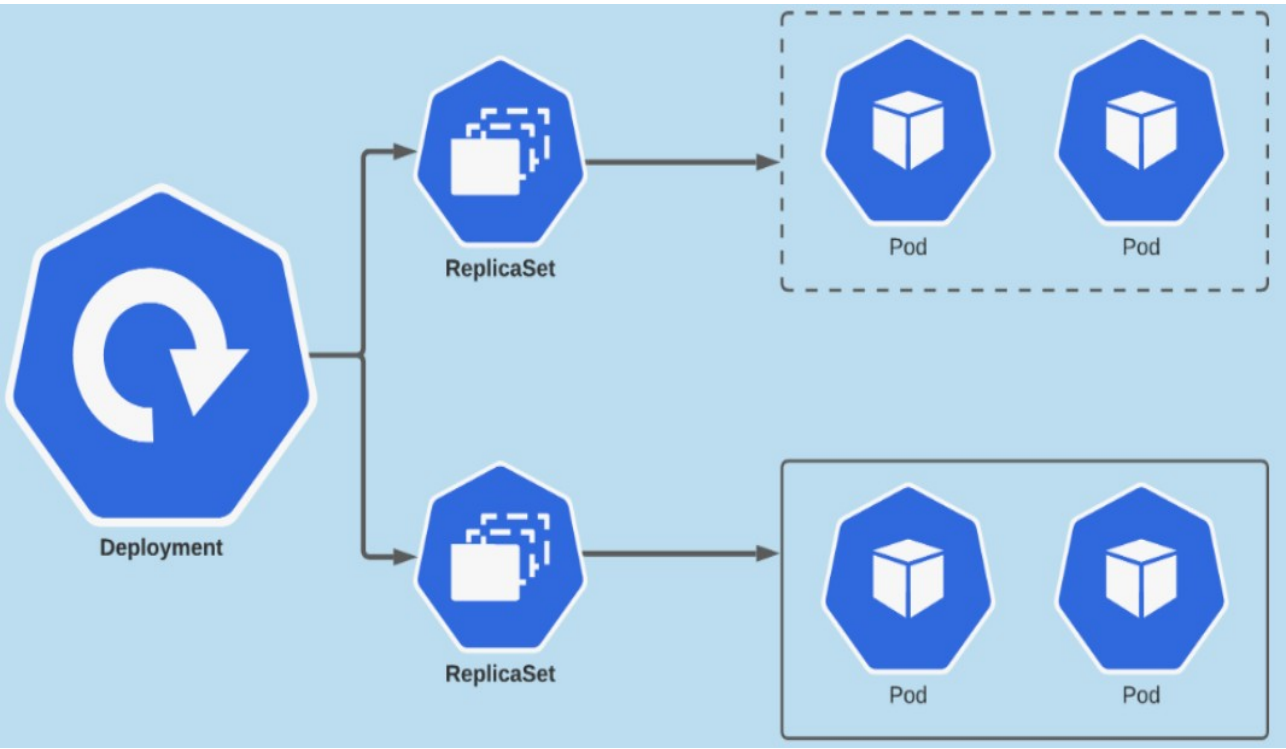


Рисунок 3.3 Ієрархія об’єктів в Kubernetes

ConfigMap використовується для зберігання конфігурацій у вигляді пар ключ-значення. Замість жорсткого кодування конфігурації в контейнері, ConfigMap дозволяє передавати її у вигляді змінних середовища або файлів.

Secret використовується для зберігання конфіденційних даних(паролі, токени, ключі). Дані у Secret закодовані, наприклад, у base64.

Ingress - це об'єкт Kubernetes, який дозволяє управляти зовнішнім доступом до сервісів через HTTP/HTTPS. Ingress використовує контролери, такі як NGINX Ingress Controller або Traefik. Функції Ingress:

- Маршрутизація трафіку на різні сервіси за шляхами або хостами.
- SSL/TLS термінація для захищеного доступу.
- Балансування навантаження на рівні HTTP.

У своєму порівнянні я використовуватиму Kubernetes для підняття декількох інстансів і управляння ними.

3.2.4 K6

K6 - це сучасний інструмент для тестування продуктивності, який дозволяє перевіряти, як веб-додатки та API поводяться під різними навантаженнями. Завдяки своїй простоті, гнучкості та можливості створення скриптів на JavaScript, K6 є зручним рішенням як для новачків, так і для досвідчених інженерів з продуктивності.

У процесі розробки та експлуатації програмного забезпечення критично важливо забезпечити високу продуктивність і стабільність системи. K6 дозволяє:

- Перевірити стабільність під навантаженням: імітувати тисячі одночасних користувачів для оцінки, як система справляється з піковим трафіком та визначати максимальну пропускну здатність додатку.
- Запобігти збоям: ідентифікувати вузькі місця у системі до виходу продукту в продакшн та вчасно оптимізувати код або архітектуру.
- Покращити якість користувацького досвіду: гарантувати швидку та надійну роботу додатків навіть при високих навантаженнях.
- Забезпечити автоматизацію тестів: інтегрувати тести продуктивності у CI/CD конвеєр для регулярної перевірки системи під час релізів.

Особисто для мене найбільшою перевагою K6 є майже нативна підтримка візуалізації. Адже k6 це продукт Grafana. K6 також досить легкий у вивченні і у запуску команд з терміналу. Також має гарно описану документацію, що пришвидшує вивчення технології.

3.3 Порівняння продуктивності

Порівняння продуктивності є найважчим етапом, адже потребує написання тестових кейсів.

Вирішив почати з найлегшого тестового прикладу, написавши прості сервера на webflux і vert.x, що повертають hello world.

Код для WebFlux рис. 3.4 та рис. 3.5:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
```

Рисунок 3.4 Залежність в pom.xml для webflux

WebfluxDemoApplication.java

```
package com.webfluxdemo.webflux_demo;

import ...

@SpringBootApplication
public class WebfluxDemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(WebfluxDemoApplication.class, args);
    }

}
```

HelloWorldController.java

```
package com.webfluxdemo.webflux_demo.api;

import ...

@RestController  no usages
public class HelloWorldController {

    @GetMapping("/hello")  no usages
    public String hello() { return "Hello World!"; }
}
```

Рисунок 3.5 Код сервісу HelloWorld на webflux

Отже, клас WebfluxDemoApplication відповідає за запуск контексту і вебсервера відповідно — це стандартний спрінговий клас, тому не буду на ньому. Клас HelloWorldController є контролером і він описує логіку, яка буде

виконана(поверне Hello World!), якщо юзер викличе посилання http://localhost:8080/hello, де 8080 - це стандартний порт в spring-web.

Код для Vert.x рис. 3.6 та рис. 3.7:

```xml
<dependency>
  <groupId>io.vertx</groupId>
  <artifactId>vertx-web</artifactId>
</dependency>
```

Рисунок 3.6 Залежність в pom.xml для vert.x

MainLauncher.java

```java
public class MainLauncher {
  public static void main(String[] args) {
    Vertx vertx = Vertx.vertx();
    vertx.deployVerticle(new MainVerticle(), res -> {
      if (res.succeeded()) {
        System.out.println("Deployment id is: " + res.result());
      } else {
        System.out.println("Deployment failed!");
      }
    });
  }
}
```

MainVerticle.java

```java
public class MainVerticle extends AbstractVerticle {  3 usages

  @Override
  public void start(Promise<Void> startPromise) {
    Router router = Router.router(vertx);

    router.route( s: "/hello").handler(req -> {
      req.response()
        .putHeader( s: "content-type",  s1: "text/plain")
        .end( s: "Hello World!");
    });

    vertx.createHttpServer() HttpServer
        .requestHandler(router)
        .listen( port: 8888) Future<HttpServer>
        .onComplete(http -> {
          if (http.succeeded()) {
            startPromise.complete();
            System.out.println("HTTP server started on port 8888");
          } else {
            startPromise.fail(http.cause());
          }
        });
  }
}
```

Рисунок 3.7 Код сервісу HelloWorld на Vert.x

Отже клас MainLauncher - деплоїть один інстанс MainVerticle. vertx.deployVerticle() - це метод Vert.x, який використовується для розгортання вертіклів у поточному екземплярі Vert.x.

new MainVerticle() - створення об'єкта вашого вертікла. Клас MainVerticle має реалізовувати інтерфейс io.vertx.core.Verticle або зазвичай розширює клас AbstractVerticle.

res -> { ... } — це асинхронний колбек, який обробляє результат розгортання вертікла.

if (res.succeeded()) - перевірка, чи успішно завершилось розгортання вертікла. Метод res.succeeded() повертає true, якщо розгортання пройшло успішно.

Якщо розгортання успішне, метод result() повертає ідентифікатор розгортання (Deployment ID), який можна використовувати для управління розгорнутим верті-клом. Якщо розгортання не вдалося (res.succeeded() == false), виконуватиметься блок else.

Клас MainVerticle - це реалізація загального абстрактного класу вертикля.

1. start() - це метод, який викликається, коли вертікл розгортається.
2. Router.router(vertx) - створення Router. Router - це компонент із Vert.x Web, який відповідає за маршрутизацію HTTP-запитів. Router.router(vertx) створює об'єкт маршрутизатора, прив'язаного до поточного екземпляра Vert.x.
3. router.route("/hello") - створює маршрут для шляху /hello.
4. handler(req -> { ... }) - додає обробник для цього маршруту.
5. req.response() - отримує об'єкт для формування HTTP-відповіді.
6. putHeader("content-type", "text/plain") - додає заголовок відповіді, що вказує на тип текстового вмісту.
7. vertx.createHttpServer() - створює HTTP-сервер.
8. .requestHandler(router) - передає маршрутизатор router як обробник запитів для сервера.
9. .listen(8888) - запускає сервер і прив'язує його до порту 8888.
10. .onComplete() - це асинхронний колбек, який обробляє результат запуску сервера.
11. http.succeeded(): Перевіряє, чи сервер успішно запущено.
12. startPromise.complete(): Сигналізує про успішний запуск вертікла.
13. Виводиться повідомлення "HTTP server started on port 8888" у консоль.\

Після написання простенького коду для серверів на vert.x і webflux я запустив тести використовуючи інструмент k6, попередньо написавши скрипт ДОДАТОК Д. Цей тестовий сценарій робить:

1. Протягом перших 30 секунд кількість віртуальних користувачів поступово збільшується до 50.
2. Стабільне навантаження:
3. Протягом 1 хвилини одночасно працюють 50 користувачів, які надсилають запити на http://localhost:8888/hello.
4. Протягом останніх 30 секунд кількість користувачів зменшується до 0.
5. Кожен користувач виконує HTTP-запит до сервера, перевіряє відповідь на статус 200 OK, робить паузу на 1 секунду перед наступним запитом.

Я взяв найпоказовіші метрики на мою думку і результат звів до табличного вигляду для кращого порівняння табл. 3.1.

Таблиця 3.1 Результати при 50 одночасних запитах

Метрика	WebFlux (1)	WebFlux (2)	Vert.x (1)	Vert.x (2)
http_req_duration (avg)	1.73ms	1.81ms	**1.07ms**	**1.11ms**
http_req_duration (p90)	2.93ms	3.17ms	**1.59ms**	**1.72ms**
http_req_duration (p95)	3.71ms	3.89ms	**1.89ms**	**1.98ms**
http_req_waiting (avg)	1.6ms	1.68ms	**951µs**	**992µs**
http_req_receiving (avg)	95.55µs	96.27µs	**91.99µs**	**91.51µs**
http_req_sending (avg)	34.07µs	34.54µs	**34.18µs**	**33.94µs**
http_reqs	4528	4526	4531	4530
iteration_duration (avg)	1s	1s	1s	1s
max latency (http_req_duration)	39ms	90.4ms	**16.35ms**	**18.33ms**

Висновок аналізу:

Час відповіді (http_req_duration).Vert.x демонструє кращу продуктивність за середнім часом відповіді (avg=1.07ms та 1.11ms) у порівнянні з WebFlux (avg=1.73ms та 1.81ms).

За p90 та p95 (час для 90% і 95% запитів) Vert.x також стабільно швидший: Vert.x: p90 ≈ 1.59-1.72ms, p95 ≈ 1.89-1.98ms.

WebFlux: p90 ≈ 2.93-3.17ms, p95 ≈ 3.71-3.89ms.

Час очікування відповіді (http_req_waiting):

Vert.x має значно менший середній час очікування (951µs і 992µs) порівняно з WebFlux (1.6ms і 1.68ms).

Час отримання відповіді (http_req_receiving):

Обидва фреймворки демонструють схожі результати в цій метриці (близько 91-96µs).

Час надсилання запитів (http_req_sending):

Показники для Vert.x і WebFlux практично однакові: ~34µs.

Максимальна затримка (max latency):

У WebFlux максимальна затримка значно більша (до 90.4ms) у порівнянні з Vert.x (16.35ms та 18.33ms).

Кількість оброблених запитів:

Загальна кількість оброблених запитів для обох фреймворків майже однакова (4528-4531), що свідчить про стабільність під навантаженням.

Загальний висновок: Vert.x демонструє кращу продуктивність у порівнянні з WebFlux, особливо за середнім часом відповіді (http_req_duration) і максимальними затримками. WebFlux працює стабільно, але показує вищі затримки, особливо у пікових наченнях.

50 одночасних запитів виглядають не дуже переконливо. Власне тому я провів ще декілька тестів де збільшив навантаження до 100 і до 300 одночасних запитів. Результати табл. 3.2 та табл. 3.3:

Таблиця 3.2 Результати при 100 одночасних запитах

Параметр	WebFlux	Vert.x
Time taken (сек)	98.3 s	**62.8 s**
Requests per second	101,599	**159,261**
Minimum latency (мс)	0.02 ms	**0.01 ms**
10 pct. latency (мс)	**0.18 ms**	0.52 ms
25 pct. latency (мс)	**0.2 ms**	0.54 ms
Median latency (мс)	**0.28 ms**	0.57 ms
Mean latency (мс)	0.98 ms	**0.63 ms**
75 pct. latency (мс)	**0.58 ms**	0.6 ms
90 pct. latency (мс)	0.99 ms	**0.91 ms**
99 pct. latency (мс)	9.19 ms	**1.24 ms**
Maximum latency (мс)	381.3 ms	**36.3 ms**
Avg. CPU usage (%)	229%	**88%**
Avg. Memory usage (MB)	507 MB	**267 MB**

Таблиця 3.3 Результати при 300 одночасних запитах

Параметр	WebFlux	Vert.x
Time taken (сек)	99 s	**64.7 s**
Requests per second	100,775	**154,610**
Minimum latency (мс)	0.02 ms	**0.01 ms**
10 pct. latency (мс)	**0.57 ms**	1.62 ms
25 pct. latency (мс)	**0.6 ms**	1.67 ms
Median latency (мс)	**0.94 ms**	1.73 ms
Mean latency (мс)	2.97 ms	**1.94 ms**
75 pct. latency (мс)	1.86 ms	**1.85 ms**
90 pct. latency (мс)	3.23 ms	**2.81 ms**
99 pct. latency (мс)	49.58 ms	**3.48 ms**
Maximum latency (мс)	350.57 ms	**65.79 ms**
Avg. CPU usage (%)	233%	**87%**
Avg. Memory usage (MB)	508 MB	**268 MB**

Vert.x виконує завдання набагато швидше: Для 100 підключень: 62.8 с проти 98.3 с у WebFlux. Для 300 підключень: 64.7 с проти 99 с. Це свідчить про кращу обробку високого навантаження у Vert.x.

Vert.x обробляє більше запитів на секунду: для 100 підключень: 159,261 RPS (+57% порівняно з WebFlux). Для 300 підключень: 154,610 RPS (+53%).

Vert.x показує меншу середню затримку (Mean latency): 0.63 ms та 1.94 ms проти 0.98 ms та 2.97 ms у WebFlux.

WebFlux споживає значно більше CPU: 229%-233% проти 87%-88% у Vert.x.

Memory: Vert.x використовує вдвічі менше пам'яті: 267-268 MB проти 507-508 MB у WebFlux.

Отже, можна зробити висновки, що Vert.x має кращу продуктивність і краще справляється з великою кількістю одночасних запитів від клієнта до сервера.

3.4 Функціональні можливості та досвід роботи з фреймворками

Vert.x використовує Event Loop модель і забезпечує подієво-орієнтований підхід до обробки запитів. Завдяки модульності та використанню Verticles і Event Bus, розробники отримують повний контроль над потоками подій та асинхронними операціями. Spring WebFlux базується на Reactive Streams і Project Reactor з підтримкою декларативного програмування через Mono та Flux. Ця архітектура добре інтегрується в Spring екосистему, але стає важчим моніторинг і трасування системи.

У Vert.x основний фокус - асинхронна обробка з використанням Future, Promise або callback API. Основні компоненти - Verticles (вертікли), які дозволяють будувати модульні застосунки. WebFlux використовує декларативний та реактивний підхід із Mono (для одиничних значень) та Flux (для потоків даних). Це дозволяє легко обробляти backpressure й асинхронні операції.

У Vert.x помилки обробляються через колбеки або об'єкти Promise/Future. Для обробки помилок у HTTP-серверах потрібно вручну налаштувати

FailureHandler. WebFlux має зручну обробку помилок "із коробки" через методи onErrorResume, doOnError або глобальний обробник @ControllerAdvice.

Vert.x підтримує кластеризацію "із коробки" завдяки Event Bus, що дозволяє організувати зв'язок між різними вузлами. Горизонтальне масштабування реалізується через розгортання кількох вертіклів. WebFlux забезпечує масштабування через Spring Cloud, що інтегрує інструменти для виявлення сервісів і балансування навантаження.

Vert.x пропонує інструменти для ручного контролю backpressure за допомогою Pump API. Це більш низькорівневий підхід. У WebFlux backpressure підтримується автоматично через Reactive Streams, що робить його зручнішим для розробників.

Vert.x підтримує HTTP/1.1, HTTP/2, WebSocket, gRPC, MQTT і є дуже продуктивним для роботи з HTTP-серверами. WebFlux підтримує HTTP/1.1, HTTP/2, WebSocket, RSocket, що дозволяє використовувати сучасні протоколи у Spring-екосистемі.

Vert.x має окремі модулі для роботи з базами даних (JDBC, MongoDB, Redis) та брокерами повідомлень (Kafka, RabbitMQ). WebFlux має перевагу завдяки інтеграції зі Spring Boot, Spring Data, Spring Security, що спрощує підключення інструментів та сервісів.

Якщо архітектурно і за продуктивністю Vert.x мав перевагу, то от WebFlux виглядає більш привабливим з точки зору девелопменту, аджзе працює в екосистемі Spring.

Vert.x складніший для освоєння через Event Loop модель та ручне управління асинхронними операціями. Він вимагає глибокого розуміння асинхронного програмування. Spring WebFlux простіший у вивченні для розробників, які вже працюють зі Spring Framework. Декларативна модель з Mono та Flux інтуїтивно зрозуміла.

Vert.x вимагає більше часу для налаштування і розробки складних сценаріїв, але надає більшу гнучкість та контроль. У WebFlux висока швидкість розробки завдяки готовим Spring Boot Starter модулям, що дозволяють швидко створювати проєкти.

Vert.x підтримує тестування через Vert.x Unit Test та JUnit, але потребує додаткової конфігурації для інтеграційних тестів. WebFlux має вбудовану підтримку для тестування через Spring Test, що дозволяє легко писати юнит-тести, інтеграційні тести і мокати сервіси.

Vert.x має менше ком'юніті порівняно зі Spring, але воно активне. Документація добре структурована, але ресурсів і статей значно менше. WebFlux користується підтримкою величезного Spring ком'юніті і стабільною підтримкою від Spring Pivotal. Навчальних ресурсів, прикладів і готових рішень дуже багато.

Vert.x забезпечує гнучкість, але потребує більше ручної конфігурації. Немає такої кількості готових "starter"-модулів, як у Spring. WebFlux має велику кількість Spring Boot Starter модулів для баз даних, безпеки, моніторингу та інших задач.

Отже, Vert.x краще підходить для проєктів, де потрібна висока продуктивність, масштабованість та гнучкість. Має потужну підтримку подієво-орієнтованої архітектури з кластеризацією і низьким споживанням ресурсів. Але вимагає більше часу для навчання й розробки, адже має менше навчальних ресурсів і слабшу підтримку ком’юніті.

Spring WebFlux ідеальний для швидкої розробки завдяки Spring Boot та декларативному API. Легкий у вивченні для команд, що вже працюють зі Spring Framework. Має вбудовані рішення для моніторингу, тестування та інтеграції з базами даних і сервісами.

ВИСНОВКИ

Розібравшись в темі реактивних систем, можна сказати, що це справді потужний підхід особливо в системах з великою навантаженістю і ймовірністю виникнення непередбачуваних виняткових ситуацій. Реактивність підтримується на багатьох мовах програмування і реалізована багатьма бібліотеками і фреймворками. У дипломній роботі я порівняв два дуже популярних інструменти для побудови реактивних систем на Java: Vert.x і WebFlux.

Vert.x показав кращу продуктивність при збільшенні одночасних підключень від клієнтів. Також Vert.x добре себе проявив при порівнянні архітектури. Він приймає запити асинхронно, натомість виконує операції синхронно в одному потоці, що забезпечує кращий моніторинг процесингу запита. Також фреймворк має можливість легко масштабування вертикально в рамках однієї JVM за рахунок збільшення кількості вертиклів. Для горизонтального масштабування також має можливість інтеграції з найпопулярнішими кластерними менеджерами.

Натомість WebFlux є легшим при вивченні, адже має купу ресурсів і знайомий Spring API. Насправді WebFlux також добре себе зарекомендував при тестах навантаження. Хоча він і поступається Vert.x, але показує гарні результати.

Отже, я б використовував Vert.x якщо розроблював сервіс з нуля і мені потрібна була б краща пропускна здатність. WebFlux використав би, коли вже маю готовий сервіс, що працює з Spring MVC і мені потрібно замінити на асинхронний веб сервер для кращої обробки великої кількості одночасних запитів.

В свою чергу хотів би застерегти він надмірного використання реактивності і асинхронності. Якщо немає чітко визначеної стратегії або нагальної потреби до використання асинхронності, я можу зробити висновок що краще використовувати звичайне синхронне програмування, адже воно легше при розробці і підтримці програмного забезпечення.

ПЕРЕЛІК ПОСИЛАНЬ

1. Тартачний Олександр, Як Intel винайшов мікропроцесор, а Intel 4004 зробив сучасні комп’ютери можливими, 15 листопада 2023р [Електронний ресурс]. - https://speka.media/yak-intel-vinaisov-mikroprocesor-i-zrobiv-sucasni-kompyuteri-mozlivimi-v7k27p
2. Jakob Jenkov, Concurrency vs. Parallelism, 11 листопада 2024 [Електронний ресурс]. - https://jenkov.com/tutorials/java-concurrency/concurrency-vs-parallelism.html
3. Jakob Jenkov, Java NIO Tutorial, 13 жовтень 2020 [Електронний ресурс]. - https://jenkov.com/tutorials/java-concurrency/concurrency-vs-parallelism.html
4. Serkan Sari, Concurrency vs Parallelism, 8 червня 2023 [Електронний ресурс]. - https://www.baeldung.com/cs/concurrency-vs-parallelism
5. David Bevans, Asynchronous vs. Synchronous Programming: Key Similarities and Differences, 14 жовтня 2024 [Електронний ресурс]. - https://www.mendix.com/blog/asynchronous-vs-synchronous-programming
6. Реактивний маніфест, 16 серпня 2014 [Електронний ресурс]. - https://www.reactivemanifesto.org/
7. Anonymous, Як створити сервер на базі Netty: особливості та приклади, 12 серпня 2023 [Електронний ресурс]. - https://habr.com/ru/articles/685518/
8. Norman Maurer, Netty in Action, 10 вересня 2016 [Електронний ресурс]. - https://livebook.manning.com/book/netty-in-action
9. Jamie Allen, Applied Akka Patterns, 5 жовтня 2015 [Електронний ресурс]. - https://www.oreilly.com/library/view/applied-akka-patterns/9781491934876/ch01.html
10. Siddharth Dutta, Spring Boot WebFlux vs Vert.x: Performance Comparison for Hello World Case, 18 липня 2023 [Електронний ресурс]. - https://medium.com/deno-the-complete-reference/springboot-webflux-vs-vert-x-performance-comparison-for-hello-world-case-41a6bd8e9f8c

ДОДАТОК А

Код сервера Netty

```
public class EchoServer {
    public static void main(String[] args) throws Exception {
        new EchoServer().start();
    }

    public void start() throws Exception {
        final EchoServerHandler serverHandler = new
EchoServerHandler();
        EventLoopGroup group = new NioEventLoopGroup();
        try {
            ServerBootstrap b = new ServerBootstrap();
            b.group(group)
             .channel(NioServerSocketChannel.class)
             .localAddress(new InetSocketAddress(8080))
             .childHandler(new ChannelInitializer<SocketChannel>() {
                 @Override
                 public void initChannel(SocketChannel ch) throws
Exception {
                     ch.pipeline().addLast(serverHandler);
                 }
             });
            ChannelFuture f = b.bind().sync();
            System.out.println(EchoServer.class.getName() +" started
and listening for connections on " + f.channel().localAddress());
            f.channel().closeFuture().sync();
        } finally {
            group.shutdownGracefully().sync();
        }}}
```

ДОДАТОК Б

Код EchoServerHandler

```
@Sharable
public class EchoServerHandler extends ChannelInboundHandlerAdapter
{
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) {
        ByteBuf in = (ByteBuf) msg;
        System.out.println(
                "Server received: " +
in.toString(CharsetUtil.UTF_8));
        ctx.write(in);
    }

    @Override
    public void channelReadComplete(ChannelHandlerContext ctx)
            throws Exception {
        ctx.writeAndFlush(Unpooled.EMPTY_BUFFER)
                .addListener(ChannelFutureListener.CLOSE);
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx,
        Throwable cause) {
        cause.printStackTrace();
        ctx.close();
    }
}
```

ДОДАТОК В

Код Netty клієнта

```
public class EchoClient {

    public void start() throws Exception {
        EventLoopGroup group = new NioEventLoopGroup();
        try {
            Bootstrap b = new Bootstrap();
            b.group(group)
                .channel(NioSocketChannel.class)
                .remoteAddress(new InetSocketAddress(8080))
                .handler(new ChannelInitializer<SocketChannel>() {
                    @Override
                    public void initChannel(SocketChannel ch) {
                        ch.pipeline().addLast(
                                new EchoClientHandler());
                    }
                });
            ChannelFuture f = b.connect().sync();
            f.channel().closeFuture().sync();
        } finally {
            group.shutdownGracefully().sync();
        }
    }

    public static void main(String[] args) throws Exception {
        new EchoClient().start();
    }
}
```

ДОДАТОК Г

Код EchoClientHandler

```
@Sharable
public class EchoClientHandler
    extends SimpleChannelInboundHandler<ByteBuf> {
    @Override
    public void channelActive(ChannelHandlerContext ctx) {
        ctx.writeAndFlush(Unpooled.copiedBuffer("Netty rocks!",
                CharsetUtil.UTF_8));
    }

    @Override
    public void channelRead0(ChannelHandlerContext ctx, ByteBuf in)
{
        System.out.println(
                "Client received: " +
in.toString(CharsetUtil.UTF_8));
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx,
        Throwable cause) {
        cause.printStackTrace();
        ctx.close();
    }
}
```

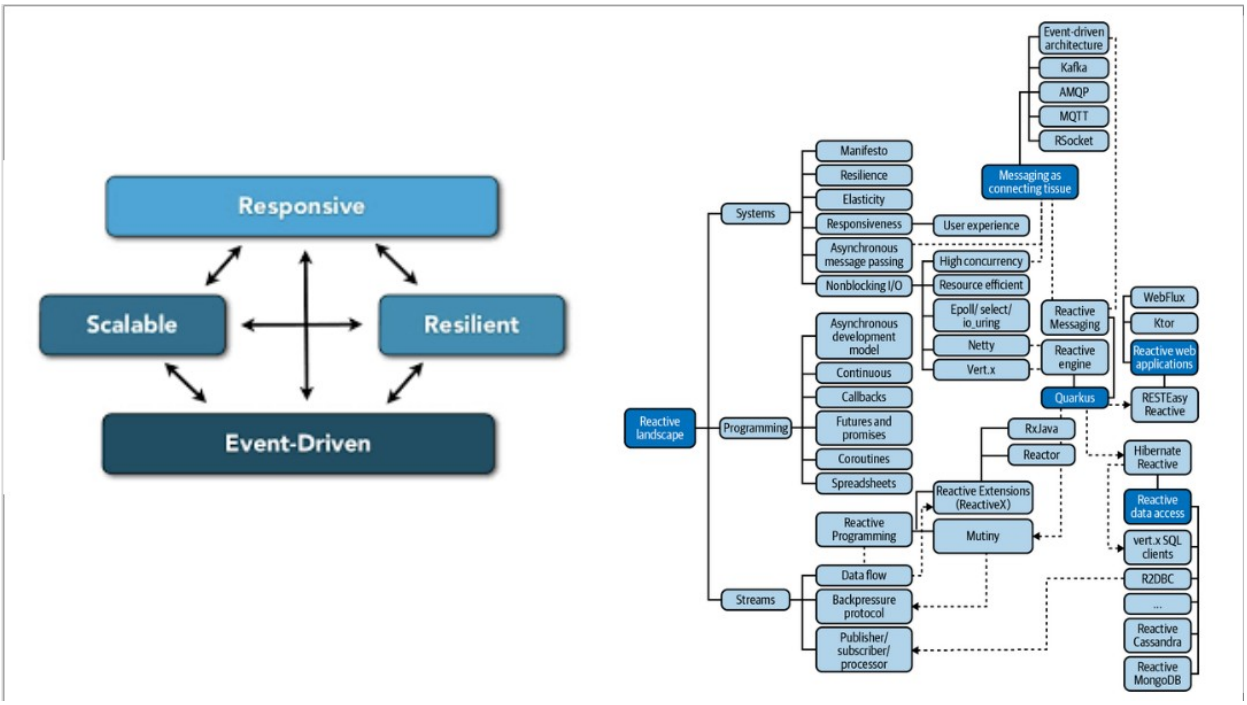
ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНИХ НАУК

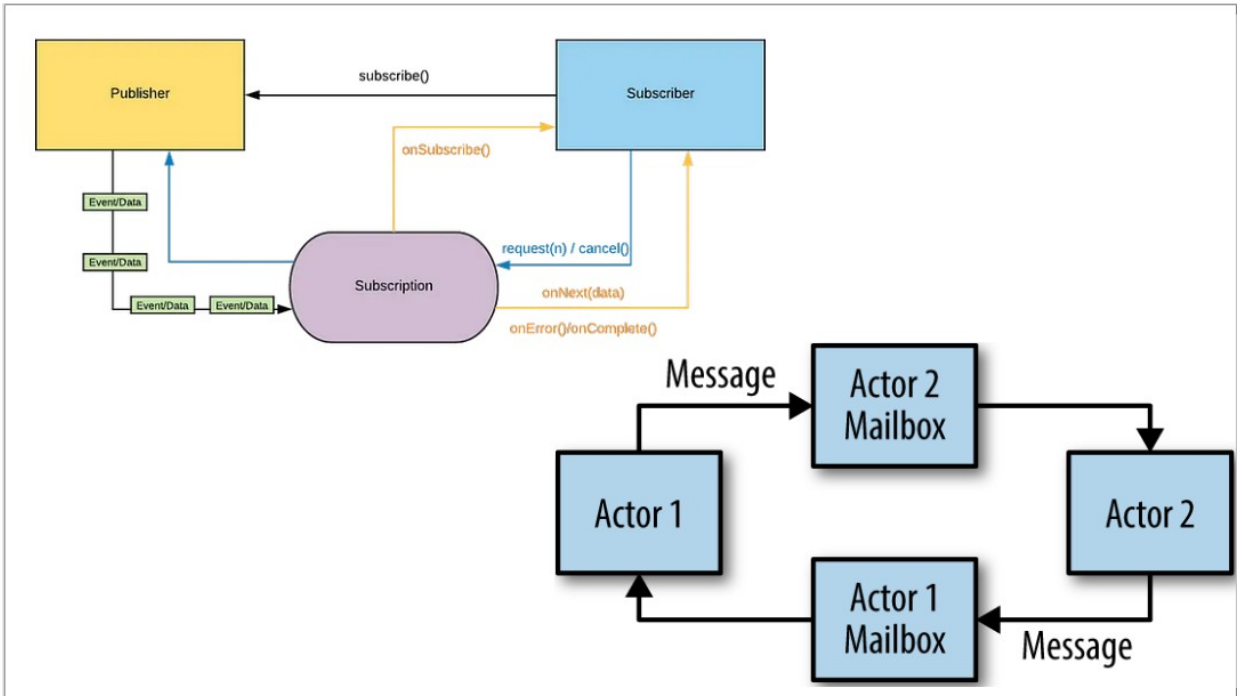
КВАЛІФІКАЦІЙНА РОБОТА
на тему: «Порівняльний аналіз Vert.x та Spring WebFlux у реактивному програмуванні на Java»

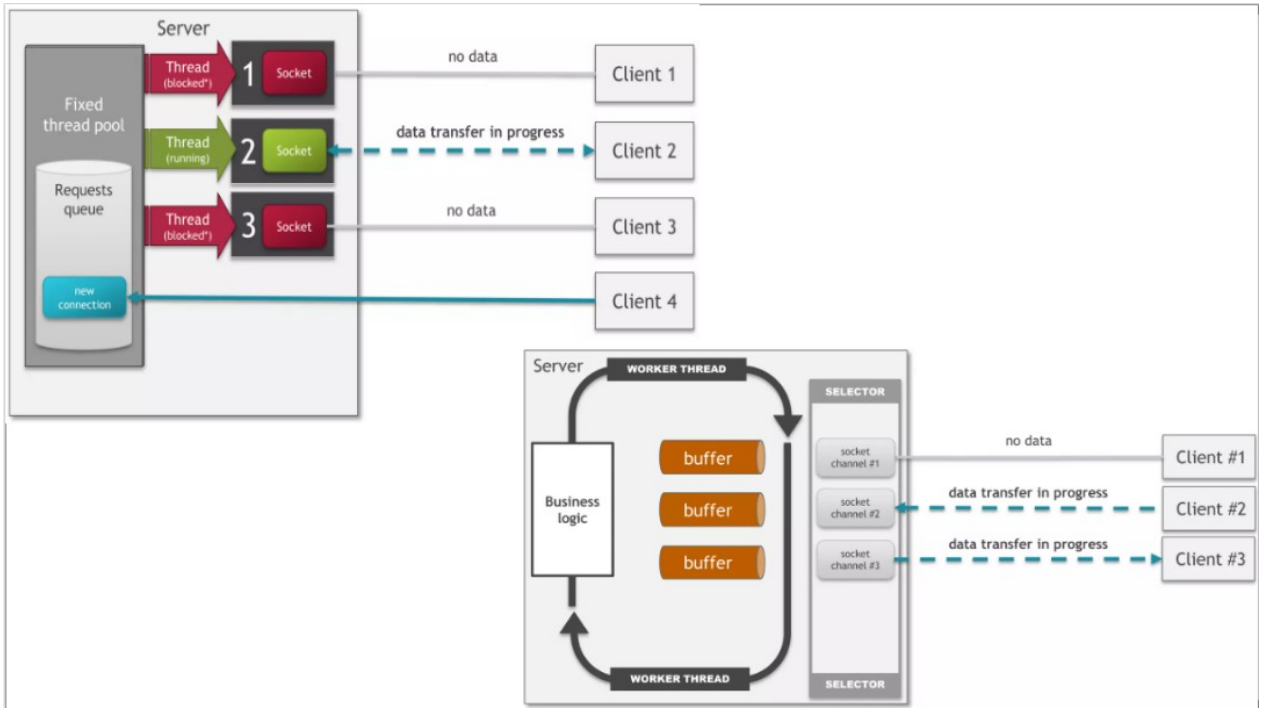
на здобуття освітнього ступеня магістра
зі спеціальності 122 Комп'ютерні науки

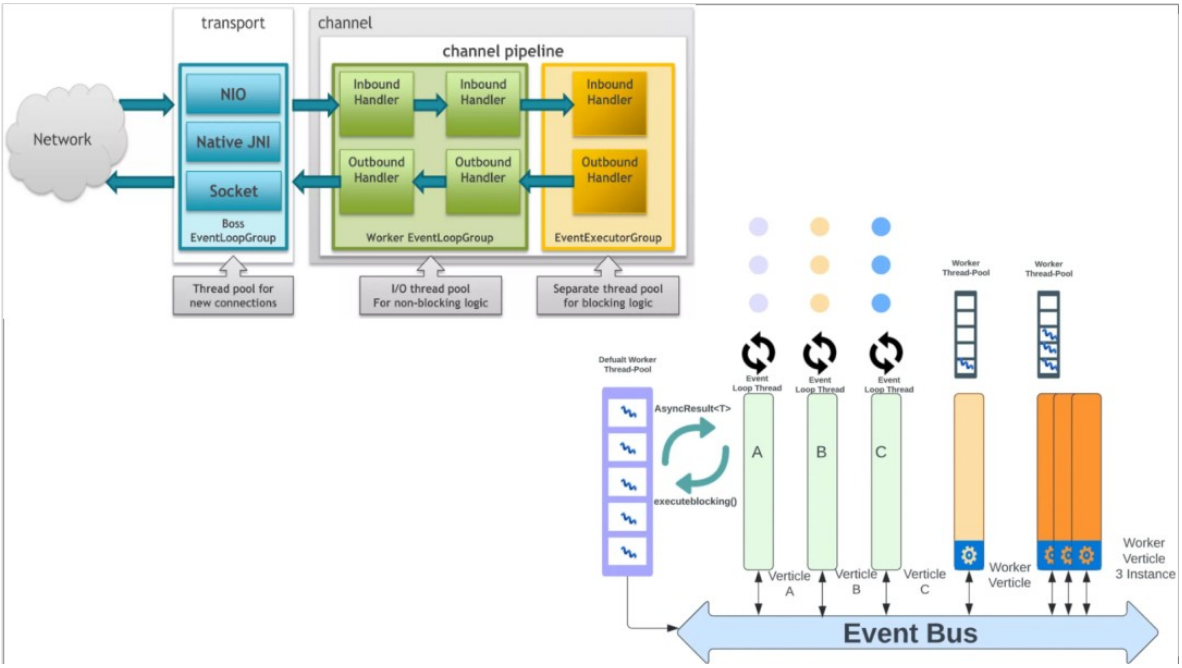
Виконав: здобувач вищої освіти групи КНДМ-63 Приходько Максим Юрійович

Київ 2024




Publisher
subscribe()
Subscriber
onSubscribe()
Event/Data
Event/Data
Event/Data
Event/Data
Subscription
request(n) / cancel()
onNext(data)
onError()/onComplete()
Message
Actor 2 Mailbox
Actor 1
Actor 2
Actor 1 Mailbox
Message


Server
Fixed thread pool
Thread (blocked*)
1
Socket
Thread (running)
2
Socket
Thread (blocked*)
3
Socket
Requests queue
new connection
no data
Client 1
data transfer in progress
Client 2
no data
Client 3
Client 4
Server
WORKER THREAD
Business logic
buffer
buffer
buffer
SELECTOR
socket channel #1
socket channel #2
socket channel #3
WORKER THREAD
SELECTOR
no data
Client #1
data transfer in progress
Client #2
data transfer in progress
Client #3



50 одночасних запитів

Метрика	WebFlux (1)	WebFlux (2)	Vert.x (1)	Vert.x (2)
http_req_duration (avg)	1.73ms	1.81ms	**1.07ms**	**1.11ms**
http_req_duration (p90)	2.93ms	3.17ms	**1.59ms**	**1.72ms**
http_req_duration (p95)	3.71ms	3.89ms	**1.89ms**	**1.98ms**
http_req_waiting (avg)	1.6ms	1.68ms	**951µs**	**992µs**
http_req_receiving (avg)	95.55µs	96.27µs	**91.99µs**	**91.51µs**
http_req_sending (avg)	34.07µs	34.54µs	**34.18µs**	**33.94µs**
http_reqs	4528	4526	4531	4530
iteration_duration (avg)	1s	1s	1s	1s
max latency (http_req_duration)	39ms	90.4ms	**16.35ms**	**18.33ms**

100 одночасних запитів

Параметр	WebFlux	Vert.x
Time taken (сек)	98.3 s	**62.8 s**
Requests per second	101,599	**159,261**
Minimum latency (мс)	0.02 ms	**0.01 ms**
10 pct. latency (мс)	**0.18 ms**	0.52 ms
25 pct. latency (мс)	**0.2 ms**	0.54 ms
Median latency (мс)	**0.28 ms**	0.57 ms
Mean latency (мс)	0.98 ms	**0.63 ms**
75 pct. latency (мс)	**0.58 ms**	0.6 ms
90 pct. latency (мс)	0.99 ms	**0.91 ms**
99 pct. latency (мс)	9.19 ms	**1.24 ms**
Maximum latency (мс)	381.3 ms	**36.3 ms**
Avg. CPU usage (%)	229%	**88%**
Avg. Memory usage (MB)	507 MB	**267 MB**

300 одночасних запитів

Параметр	WebFlux	Vert.x
Time taken (сек)	99 s	**64.7 s**
Requests per second	100,775	**154,610**
Minimum latency (мс)	0.02 ms	**0.01 ms**
10 pct. latency (мс)	**0.57 ms**	1.62 ms
25 pct. latency (мс)	**0.6 ms**	1.67 ms
Median latency (мс)	**0.94 ms**	1.73 ms
Mean latency (мс)	2.97 ms	**1.94 ms**
75 pct. latency (мс)	1.86 ms	**1.85 ms**
90 pct. latency (мс)	3.23 ms	**2.81 ms**
99 pct. latency (мс)	49.58 ms	**3.48 ms**
Maximum latency (мс)	350.57 ms	**65.79 ms**
Avg. CPU usage (%)	233%	**87%**
Avg. Memory usage (MB)	508 MB	**268 MB**