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

КВАЛІФІКАЦІЙНА РОБОТА

на тему:

«Дослідження продуктивності високонавантажених HTTP серверів, працюючих на базі асинхронного середовища виконання та розроблених з використанням реактивної моделі програмування на мові 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.

____________________ Михайло ДЬЯЧЕНКО

Виконав: здобувач вищої освіти гр. КНДМ-63
Дьяченко Михайло Костянтинович
Керівник: доктор філософії
Березовська Юлія Володимирівна
Рецензент:

Київ 2023

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
«____» ___________ 20__ р.

ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

Дьяченко Михайла Костянтиновича

1. Тема кваліфікаційної роботи: Дослідження продуктивності високонавантажених HTTP серверів, працюючих на базі асинхронного середовища виконання та розроблених з використанням реактивної моделі програмування на мові Java,
керівник кваліфікаційної роботи Юлія БЕРЕЗОВСЬКА, доктор філософії,
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «19» жовтня 2023р. №. 145

2. Строк подання кваліфікаційної роботи 20.12.2023 р.

3. Вихідні дані до кваліфікаційної роботи: Науково-технічна література з питань, пов’язаних з темою роботи

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити):

1. Аналіз проблеми швидкості обробки HTTP-запитів у сучасних веб-додатках;

2. Методи і засоби побудови веб-додатків й порівняння їх продуктивності;

3. Розробка програмного забезпечення та тестування веб-додатків із використанням різних архітектурних підходів.

5. Перелік ілюстративного матеріалу: презентація

6. Дата видачі завдання 19.10.2023 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір науково-технічної літератури	19.10-05.11.23	
2	Визначення предмета та об’єкта дослідження	05.11-12.11.23	
3	Аналіз проблеми швидкості обробки HTTP-запитів у сучасних веб-додатках	13.11-19.11.23	
4	Методи і засоби побудови веб-додатків й порівняння їх продуктивності	20.11-25.11.23	
5	Розробка програмного забезпечення та тестування веб-додатків із використанням різних архітектурних підходів	27.11-03.12.23	
6	Вступ, висновки, реферат	04.12-10.12.23	
7	Розробка доповіді та демонстраційних матеріалів	11.12-20.12.23	
8	Попередній захист роботи	21.12-29.12.23	

Здобувач вищої освіти _______________ Михайло ДЬЯЧЕНКО

Керівник кваліфікаційної роботи _______________ Юлія БЕРЕЗОВСЬКА

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня магістра: 58 стор., 51 рис., 1 табл., 10 джерел.

Мета роботи – підвищення продуктивності та відмовостійкості високонавантажених веб-додатків шляхом використання асинхронного підходу обробки HTTP-запитів.

Об’єкт дослідження – процес обробки HTTP-запитів у високонавантажених веб-додатках.

Предмет дослідження – модель обробки HTTP-запитів у високонавантажених веб-додатках.

Наукове завдання – розробка серверної частини веб-додатків з використанням різних альтернативних моделей обробки HTTP-запитів.

Методи дослідження – програмування, тестування, аналіз та порівняння продуктивності та відмовостійкості веб-додатків.

Проведено аналіз проблеми швидкості обробки HTTP-запитів в умовах сучасних вимог до веб-додатків. Розглянуті альтернативні до синхронних підходи до розробки веб-додатків.. Розроблено серверну частину веб-додатків на мові Java. Було проведено тестування навантаження та проаналізовані й порівняні результати.

Отримані результати свідчать про перспективи використання асинхронної моделі обробки HTTP-запитів з залученням сучасних популярних фреймворків, таких як Spring WebFlux, та надають кращі показники продуктивності та відмовостійкості в умовах високого навантаження та інтеграції з сторонніми програмами.

КЛЮЧОВІ СЛОВА: АСИНХРОННА ОБРОБКА HTTP-ЗАПИТІВ, ФРЕЙМВОРК SPRING WEBFLUX, МІКРОСЕРВІСНА АРХІТЕКТУРА, МОВА ПРОГРАМУВАННЯ JAVA

ABSTRACT

Text part of the master's qualification work: 58 pages, 51 pictures, 1 table, 10 sources.

The goal of the work is to increase the performance and failure resilience of highly loaded web applications by using an asynchronous approach to processing HTTP requests.

The object of research is the process of handling HTTP requests in highly loaded web applications.

The subject of the research is a model of processing HTTP requests in highly loaded web applications.

The task is the development of the server-side web applications using different alternative models of processing HTTP requests.

Methods of research – development, testing, analysis and comparison of performance and failure resilience of created web applications.

An analysis of the problem of processing HTTP requests speed under conditions of modern requirements for web applications was completed. The synchronous approach alternatives of the development of web applications was researched. The server-side web applications were developed using the Java language. Load testing was performed and the results analyzed and compared.

The obtained results indicate the prospects of using an asynchronous model of processing HTTP requests with the involvement of modern popular frameworks such as Spring WebFlux, and provide better indicators of performance and failure resilience under conditions of high load and integration with third-party services.

KEYWORDS: ASYNCHRONOUS PROCESSING OF HTTP REQUESTS, SPRING WEBFLUX FRAMEWORK, MICROSERVICE ARCHITECTURE, JAVA PROGRAMMING LANGUAGE

ЗМІСТ

1 АНАЛІЗ ПРОБЛЕМИ ШВИДКОСТІ ОБРОБКИ HTTP-ЗАПИТІВ У СУЧАСНИХ ВЕБ-ДОДАТКАХ.. 12
1.1 Аналіз популярності мови програмування Java та моделей обробки HTTP-запитів, які використовуються у веб-додатках.. 12
1.2 Тренди архітектурних підходів до розробки..20
1.2.1 Архітектура, орієнтована на хмарні сервіси..20
1.2.2 Мікросервісна архітектура... 21
1.2.3 Контейнеризація... 22
1.3 Тренди використання I/O на мові Java...23
1.4 Реактивне програмування.. 24
1.4.1 Основні переваги та недоліки реактивного програмування..................24
1.4.2 Порівняння реактивного та імперативного підходу програмування на мові програмування Java...25
2 МЕТОДИ І ЗАСОБИ ПОБУДОВИ ВЕБ-ДОДАТКІВ Й ПОРІВНЯННЯ ЇХ ПРОДУКТИВНОСТІ.................................. 27
2.1 Мова програмування Java.. 27
2.2 Технології для розробки веб-додатків Spring Framework............................. 27
2.3 Технологія контейнеризації Docker.. 29
2.4 Управління залежностями та збіркою проекту Apache Maven.......................30
2.5 Інтегроване середовище розробки IntelliJ IDEA.. 31
2.6 Інструмент для тестування веб-додатків Apache JMeter...............................32
2.7 Інструмент для профілювання та аналізу продуктивності Java-додатків VisualVM..33
2.8 Система керування базами даних PostgreSQL...34
2.9 Інструмент для керування версіями схеми бази даних Liquibase................. 35
2.10 Фреймворк для трансформації об’єктів у середині Java програми MapStruct.. 36
2.11 Доступ до бази даних з Java програми за допомогою Spring Data..............37
3 ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ВЕБ-ДОДАТКІВ ІЗ ВИКОРИСТАННЯМ СИНХРОННОЇ ТА АСИНХРОННОЇ МОДЕЛІ ОБРОБКИ HTTP-ЗАПИТІВ..38
3.1 Функціональність та високорівнева архітектура...38
3.2 Налаштування та заповнення даними таблиці бази даних PostgreSQL....... 40
3.3 Розробка веб-додатку.. 42
3.3.1 Мікросервіс ads-utility-bills-service..42
3.3.2 Створення веб-додатку administrative-data-service................................ 45
3.3.3 Тестування додатку administrative-data-service працюючого на базі

"thread per request" моделі.. 50
3.3.4 Модификація веб-додатку administrative-data-service............................. 57
3.3.5 Тестування додатку administrative-data-service працюючого на базі "event driven design" моделі...60
3.4 Порівняння результатів тестів...65
ВИСНОВКИ...69
ПЕРЕЛІК ПОСИЛАНЬ... 70

Вступ

У сучасному світі, де конкуренція на ринку веб-додатків є вкрай високою, статистика та тренди вказують на те, що швидкість обробки інформації стає не просто важливим фактором, але і критично необхідним для забезпечення успіху додатків. Згідно із звітами Akamai [10], понад 40% користувачів покидають веб-сайт, якщо завантаження сторінки займає більше 3 секунд.

Мікросервісна архітектура веб-додатків набуває популярності через свою здатність до високої масштабованості та гнучкості. За останні два роки більше 60% розробників віддають перевагу мікросервісам [9], щоб забезпечити легку підтримку та невелику залежність між компонентами додатку. З іншого боку, збільшення кількості сторонніх веб-сервісів свідчить про те, наскільки важливо ефективно інтегрувати зовнішні ресурси для покращення функціональності веб-додатка.

Синхронна обробка HTTP-запитів на серверній стороні довгий час була основним підходом у веб-розробці. Однак у сучасних умовах ця модель зіткнулася з численними викликами та обмеженнями, що ускладнюють ефективність та швидкість веб-додатків. Синхронна обробка HTTP-запитів на серверній стороні, яка традиційно використовується у веб-розробці, зазнає викликів та обмежень. Блокування потоків у синхронному підході є однією з найбільш поширених проблем, що впливає на продуктивність додатків.

Зараз, у світлі зростаючої складності веб-систем та вимог до миттєвої відповіді, синхронна модель обробки HTTP-запитів на серверній стороні виявляється недостатньою для забезпечення високої продуктивності. Основний акцент робиться на пошуку більш ефективних та масштабованих рішень, які можуть задовольнити потреби сучасних веб-додатків у швидкості та відмовостійкості. Асинхронність, реактивність та використання неблокуючих

фреймворків стають все більш популярними альтернативними варіантами для оптимізації швидкості та ефективності серверної обробки HTTP-запитів.

1 АНАЛІЗ ПРОБЛЕМИ ШВИДКОСТІ ОБРОБКИ HTTP-ЗАПИТІВ У СУЧАСНИХ ВЕБ-ДОДАТКАХ

1.1 Аналіз популярності мови програмування Java та моделей обробки HTTP-запитів, які використовуються у веб-додатках

Згідно з багатьма відомими джерелами статистичної інформації, одним з яких є індекс TIOBE [6], мова програмування JAVA була і залишається лідером на всесвітньому ринку протягом останніх двадцяти років як зображено на рисунку 1.1.

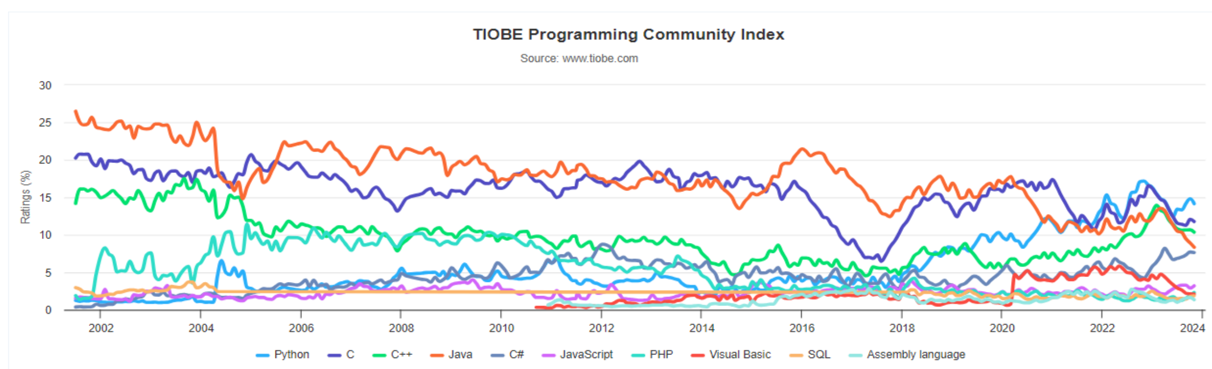


Рисунок 1.1 – Індекс програмної спільноти за роками

Відповідно до опитувань проведених w3techs [7], позицію мови програмування Java на ринку з точки зору популярності та трафіку порівняно з найпопулярнішими серверними мовами програмування. Технологія в нижньому правому куті використовується багатьма сайтами, але переважно сайтами із середнім рейтингом трафіку. Технологія у верхньому лівому куті використовується меншою кількістю сайтів, але переважно сайтами з високим трафіком. Найкращим місцем буде правий верхній кут. Таким чином, ми маємо

велику кількість програм написаних на мові програмування Java, велика частка яких є веб-додатками (рис. 1.2.).

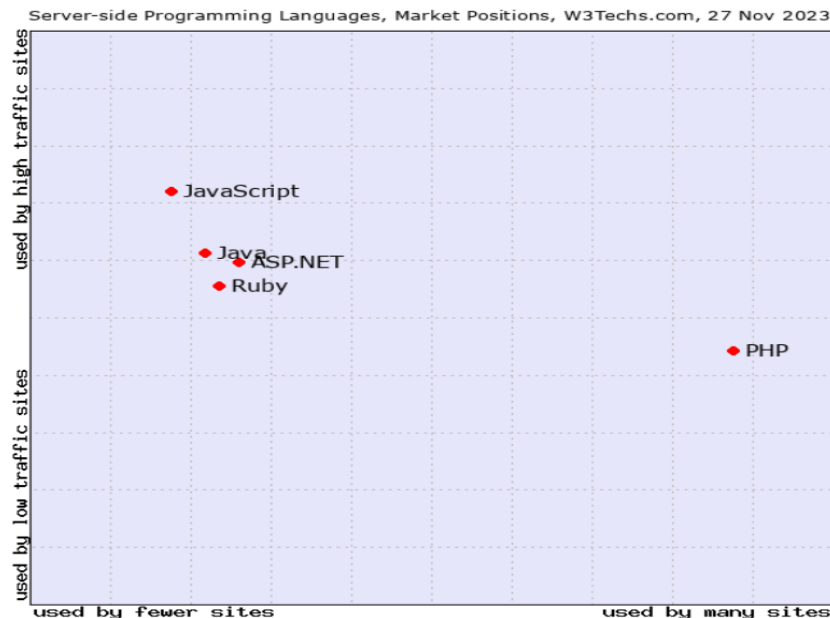


Рисунок 1.2 – Графік популярності мов програмування залежно від навантаженості програм

Далі наведені деякі основні поняття, які зустрічаються в тексті.

Java – це високорівнева, об'єктно-орієнтована та платформо-незалежна мова програмування, яка широко використовується в розробці веб-додатків та інших сферах.

Для передачі гіпертексту, який визначає, які дані та яким чином обмінюються між комп’ютерами у веб-інтернеті використовується протокол Hypertext Transfer Protocol (HTTP). HTTP є основним протоколом для передачі ресурсів (таких як HTML-документи, зображення, таблиці стилів, скрипти та інші) з веб-серверів до веб-клієнтів (наприклад, веб-браузерів).

Програмне забезпечення, яке призначено для використання через веб-браузер називають веб-додатком. Він доступний користувачам через інтернет або локальну мережу. Веб-додаток використовується для вирішення певних завдань, інтерфейс клієнтської частини якого надається через веб-браузер, що

робить його доступним для користувачів на різних платформах та пристроях. Веб-додаток складається з клієнтської частини (зазвичай веб-браузера) та серверної частини (сервера, який обробляє запити та надає відповіді). Доступ до веб-додатку зазвичай відбувається через протокол HTTP або його безпечну версію HTTPS, який забезпечує передачу даних між клієнтом і сервером. Приклади веб-додатків включають соціальні мережі, електронну пошту, онлайн-магазини, онлайн-банкінг, групи для спільної роботи, віртуальні офіси та багато інших. Вони забезпечують зручний та доступний спосіб використання програмних послуг через веб-браузер.

Архітектурний підхід до розробки програмного забезпечення, при якому додаток розбивається на невеликі та незалежні компоненти називається мікросервісами. Кожен мікросервіс є самодостатнім та виконує конкретну функцію або послугу, і вони можуть взаємодіяти між собою через Application Programming Interface (API). Мікросервісна архітектура стала популярною через свою гнучкість, забезпечення швидкої розробки та впровадження, а також здатність легко масштабувати та підтримувати додаток у змінному середовищі.

Термін “програмний потік” (“thread”) відноситься до найменшої одиниці виконання програмного коду в операційній системі. Програмні потоки є легковаговими, окремими шляхами виконання, які можуть одночасно працювати в межах одного процесу.

У термінах програмування та операційних систем існує поняття Input/Output (I/O) або ввод/вивід що вказує на обмін даними між програмою та зовнішніми пристроями або ресурсами, такими як файли, мережа, консоль, друкувальні апарати та інші пристрої вводу/виводу.

За виконання операцій програм та керування різними аспектами комп’ютерної системи відповідає Central Processing Unit (CPU). CPU або центральний процесор – це основний обчислювальний компонент комп’ютера. Він є “серцем” комп’ютера, відповідає за виконання арифметичних та логічних операцій, обробку інструкцій та керування роботою інших компонентів.

Типовий веб-додаток може мати кілька тисяч активних клієнтів із незалежними запитами від кожного з них, які надходять кілька разів на хвилину або навіть секунду. Кожен такий запит потребує ресурсів на сервері для обробки, і чим швидше запит буде оброблений, тим краще.

Мова JAVA пропонує два варіанти роботи з потоками вводу/виводу даних.

Java IO (Input/Output) використовується для виконання операцій читання та запису . Пакет java.io містить усі класи, необхідні для операцій вводу та виводу. Потоки Java IO блокуються. Це означає, що коли потік викликає операції запису або читання, цей потік блокується, поки не буде даних для читання, або поки дані не будуть повністю записані. Потік тим часом не може робити нічого іншого. Таким чином, для підтримання паралельних операцій необхідно створення нових Java потоків (рис. 1.3).

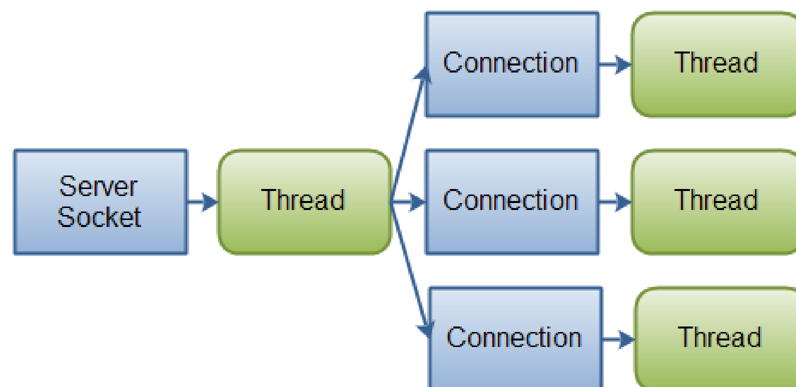


Рисунок 1.3 – Схема обробки синхронних I/O на мові Java

Java IO використовується у веб-серверах написаних на мові Java які використовують традиційну модель обробки HTTP-запитів “thread per request”, принцип якої є виділення окремого програмного потоку для обробки кожного запиту. Коли кількість активних програмних потоків, яким потрібен час CPU для виконання обчислень, перевищує кількість ядер на фізичному комп'ютері де розгорнутий веб-додаток, операційна система (ОС) розподіляє час процесора між активними потоками програми шляхом контекстного перемикання між ними.

Відносно легко програмувати системи за допомогою моделі “thread per request”, оскільки обробка кожного запиту ізольована від інших, порядок виконання коду зазвичай є послідовним і не залежить від кількості клієнтів. У разі спільних структур даних та інших ресурсів, які використовуються з декількох програмних потоків, між потоками необхідно використовувати методи синхронізації, щоб запобігти перегонам даних (Data Race).

Незважаючи на те, що потокова модель не є складною для реалізації, дозволяє обробляти декілька завдань одночасно, а потоки широко підтримуються мовами програмування, вона все ще має певні недоліки та обмеження. Використання потоків робить програми складними для розуміння та непередбачуваними. Необхідно підтримувати належний стиль програмування, щоб усунути небажаний недетермінізм, якого можна досягти за допомогою методів синхронізації. Однак, синхронізацію може бути складно виконати правильно, і вона схильна до помилок, які призводять до ненадійності, не кажучи вже про проблеми, які важко налагодити, як взаємоблокування, спричинені неправильною синхронізацією. У той же час надмірне використання синхронізації зменшує частку коду, що виконується паралельно, і додає накладні витрати на блокування, що спричиняє зниження продуктивності та не дає задовільного результату.

Іншим аргументом проти потокової моделі, яка не включає людський фактор, є споживання пам’яті. Кожен програмний потік підтримує свій власний стек (область) у пам'яті. Таким чином, використання пам'яті зростатиме лінійно до кількості потоків. Як мінімум, це призведе до додаткових витрат з точки зору інфраструктури, щонайбільше це може спричинити проблеми з доступністю програми, коли кількість потоків і розмір їхнього стека не обмежені, і створюється занадто багато, що досягає обмежень пам’яті сервера.

Окрім проблем з пам’яттю, під час великого навантаження, коли для обробки запитів створюється велика кількість потоків, вони починають конкурувати за час CPU. Це призводить до постійного перемикання контексту між

ними, що займає ресурси CPU, які інакше могли б бути використані для обчислень, додає накладні витрати на планування та збільшує ймовірність промахів кешу.

Описані проблеми викликають занепокоєння щодо масштабованості запропонованої моделі, коли кількість клієнтів, які ви очікуєте обслуговувати одночасно, обчислюється десятками тисяч або більше. З усіма перевагами та недоліками такого підходу, потокова модель та її варіації були найпопулярнішим вибором для серверних програм протягом двох десятиліть і досі залишаються такими. Це підходить для більшості типових серверних систем і допомагає використовувати багатоядерність сучасних CPU.

Іншим варіантом роботи з потоками вводу/виводу даних в мові Java є Java NIO (New IO) було представлено з JDK 4 для реалізації високошвидкісних операцій вводу-виводу. Це альтернатива стандартним IO API. Неблокуючий режим Java NIO дозволяє потоку запитувати дані про читання з каналу та отримувати лише те, що доступно на даний момент, або взагалі нічого, якщо дані наразі недоступні. Замість того, щоб залишатися заблокованим, доки дані не стануть доступними для читання, Java потік може працювати з чимось іншим (рис. 1.4).

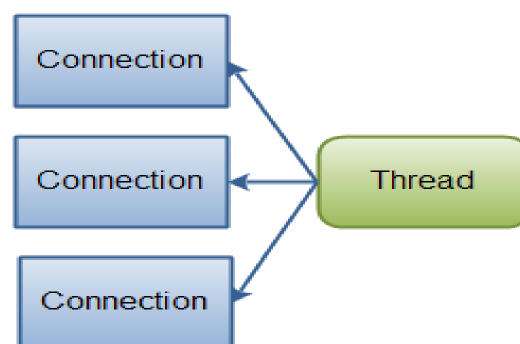


Рисунок 1.4 – Схема обробки асинхронних I/O на мові Java

Java NIO дозволяє керувати декількома каналами (мережевими з’єднаннями чи файлами), використовуючи лише один або декілька потоків, але недолік

полягає в тому, що розбір даних може бути дещо складнішим, ніж під час читання даних із блокуючого потоку.

Java NIO лежить в основі веб рішень на мові Java працюючих на моделі обробки HTTP-запитів “event driven design”.

Сигнал, який створює компонент після досягнення заданого стану є подією (event). Основним компонентом дизайну “event driven design”, є цикл подій (event loop). Він забезпечує координацію одного програмного потоку з декількома I/O з’єднаннями, різко зменшуючи кількість потоків, необхідних для роботи системи. Цикл подій обробляє події з черги подій послідовно та повертається одразу після реєстрації обробника такої події. Система може ініціювати події завершення операцій, такі як виклик бази даних або виклик стороннього сервісу. Цикл подій може визвати зареєстрований обробник такої події та повернути результат клієнту. (рис. 1.5)

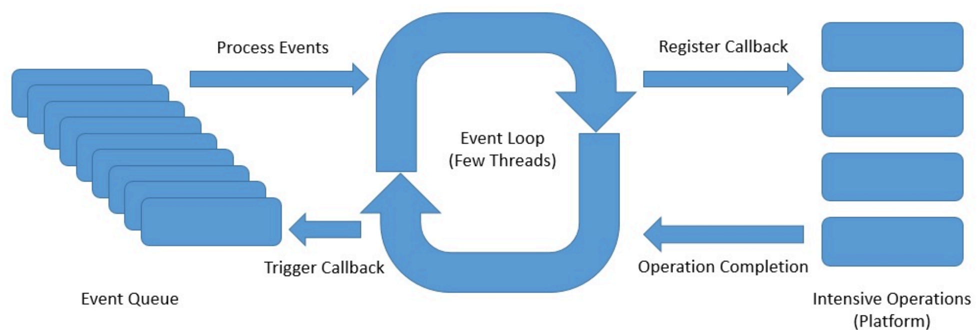


Рисунок 1.5 – Модель “event driven” з циклом подій

Завдання основного потоку полягає в безперервному управлінні чергою подій та делегуванні подій обробнику. Відповідальність обробника подій полягає в тому, щоб знати, як обробляти певні типи подій і виконувати відповідну дію. Це має вирішальне значення, що обробка події є швидкою та не затримує цикл подій, що в свою чергу призведе до зниження продуктивності та збільшення затримок.

Це веде до наступного важливого компонента в “event driven design” -

неблокуюче виконання операцій I/O. Таким чином, результат запиту на виконання I/O операції повертається негайно, навіть якщо така операцію не завершена, дозволяючи не блокувати програмний потік виконання та запроваджуючи паралелізм для операцій I/O. Після завершення операції I/O подія, яка сповіщатиме про те, що обробку можна продовжити, буде виключена. Цей підхід ґрунтується на підтримці та продуктивності неблокуючих механізмів вводу/виводу, наявних в ОС.

До цього моменту було описано загальний робочий процес системи, керованої подіями, і те, як вона використовує переваги неблокуючих механізмів вводу/виводу в операційних системах.

Інше питання полягає в тому, що станеться, якщо обробка подій не пов'язана зі швидкістю вводу/виводу, а просто виконує важкі обчислення, які потребують багато часу, і як можна скористатися перевагами багатоядерних CPU. З цієї причини на практиці системи, керовані подіями, не обмежуються одним обробником подій, але зазвичай мають N обробників подій, кожен з яких відповідає за конкретну подію та виділяє окремий програмний потік для виконання. Таким чином, існує обмежена низька кількість програмних потоків, що не залежить від кількості одночасних з'єднань, які працюють паралельно, ефективно використовуючи ресурси багатоядерного процесора та мінімізуючи накладні витрати на перемикання контексту, наявні в моделі на основі потоків.

Програма, розроблена на "event driven design", не обмежена кількістю програмних потоків, якими може ефективно керувати операційна система, а масштабованість залежить, натомість, від продуктивності однопотокового циклу подій.

Недоліком підходу, керованого подіями, є те, що його складно програмувати. Це ускладнює керування програмою та розуміння послідовності виконання команд. Також, режим відлагодження та пошуку помилок стає складнішим, у порівнянні з моделлю "thread per request", де код виглядає більш природно.

1.2 Тренди архітектурних підходів до розробки

1.2.1 Архітектура, орієнтована на хмарні сервіси

Cloud-native architecture (архітектура, орієнтована на хмарні сервіси) — це підхід до розробки та експлуатації програмного забезпечення, який спеціально розроблено для використання в хмарних середовищах. Цей підхід базується на наборі принципів та практик, які роблять розробку, впровадження та масштабування додатків у хмарних сервісах більш ефективним та гнучким. У свою чергу хмарне середовище вказує на використання ресурсів та послуг, які надаються через Інтернет. Замість того, щоб розміщувати та управляти власною інфраструктурою, користувачі можуть отримати доступ до віртуальних ресурсів, таких як обчислювальна потужність, зберігання даних та сервіси, через мережу, зазвичай за допомогою веб-інтерфейсу. Хмарні програми створюються з використанням мікросервісів і є високомасштабованими та стійкими, що робить їх ідеальним рішенням для компаній, які хочуть скористатися гнучкістю та масштабованістю хмарного середовища.

Відповідно до останнього звіту Gartner, до 2023 року понад 70% глобальних організацій використовуватимуть ту чи іншу форму хмарної архітектури для стимулювання цифрових бізнес-ініціатив. Це величезне зростання порівняно з приблизно 20% організацій, які використовували хмарну архітектуру в 2019 році. Отже, очевидно, що хмарна архітектура швидко стає нормою в сфері розробки програмного забезпечення.

Однак, як і у випадку з будь-якою новою технологічною тенденцією, існують проблеми та міркування, про які підприємства повинні пам’ятати, використовуючи хмарну архітектуру. Наприклад, забезпечення безпеки хмарних додатків і даних є критично важливим, оскільки вони часто піддаються більшій кількості потенційних вразливостей і векторів атак, ніж локальні програми. Крім

того, керування та моніторинг складної архітектури мікросервісів може бути складним завданням і вимагає спеціальних навичок та інструментів. Незважаючи на ці проблеми, переваги хмарної архітектури очевидні, й компанії, які зможуть успішно її впровадити, матимуть хороші можливості для успіху в епоху цифрових технологій.

1.2.2 Мікросервісна архітектура

Мікросервісна архітектура – це підхід до розробки програмного забезпечення, який передбачає розбиття програм на невеликі незалежні програми (сервіси), які можна розробляти з використання різних мов програмування та розгортати окремо. Цей підхід дозволяє компаніям розробляти та розгортати програмне забезпечення швидше та з більшою гнучкістю, ніж традиційні монолітні програми.

Згідно з опитуванням O'Reilly Media, 69% організацій або зараз використовують мікросервіси, або планують запровадити їх у найближчому майбутньому. Це чітке свідчення того, що архітектура мікросервісів набирає популярності та, ймовірно, продовжить це робити в найближчі роки.

Цей підхід також вимагає вищого рівня координації та зв’язку між командами розробників таких сервісів, щоб забезпечити безперебійну співпрацю всіх незалежних компонентів.

Незважаючи на труднощі, переваги архітектури мікросервісів роблять її привабливим вибором для компаній, які прагнуть залишатися гнучкими та конкурентоспроможними в сучасному цифровому середовищі, що швидко розвивається.

1.2.3 Контейнеризація

Контейнеризація — це підхід до розробки програмного забезпечення, який передбачає упаковку додатків та їхніх залежностей у контейнери. Контейнери — це ізольовані середовища, які можуть працювати будь-де не конфліктуючи з оточенням операційної системи комп'ютера де ці контейнери розгорнуті. Контейнеризація також дозволяє спростити розгортання, керування версіями та обслуговування програм.

Ще однією перевагою контейнерної архітектури програмного забезпечення є портативність, яку вона пропонує. Контейнери можна без проблем переміщувати з одного середовища в інше, що полегшує розробникам роботу над проектами, не турбуючись про проблеми сумісності між різними операційними системами чи обладнанням. Це особливо корисно у випадках, коли програми потрібно переміщувати між середовищами розробки, тестування та виробництва. Крім того, контейнеризація дозволяє ефективніше використовувати ресурси, оскільки кілька контейнерів можна запускати на одному хості, зменшуючи потребу в додатковому обладнанні. Ці переваги сприяли широкому впровадженню контейнерної архітектури програмного забезпечення в різних галузях.

Відповідно до звіту Grand View Research, очікується, що розмір світового ринку контейнерних технологій до 2025 року досягне 8,2 мільярда доларів США, зростаючи на 26,5% у середньому з 2019 по 2025 рік. Це приголомшливі темпи зростання та чіткий показник того, що потреба в рішеннях з контейнерною архітектурою буде тільки зростати .

1.3 Тренди використання I/O на мові Java

Звичайна практика реалізовувати великі корпоративні Java-додатки за принципом “thread per request”, де відповідна кількість потоків виконують складні обчислення на серверній стороні. Зв’язок між клієнтською частиною програми та серверною часто прихований за рівнями абстракцій, які надають фреймворки, що часто в середині використовують підхід синхронного блокування I/O.

Незвично бачити розробника, який працює зі стандартним пакетом I/O запропонованим у Java, щоб реалізувати ефективний спосіб мережевого зв’язку. Розробник зосереджується на розробці функціональності системи, а фреймворки виконують усе інше, включаючи “брудну” роботу мережевого зв’язку, з невеликим втручанням і налаштуваннями, якщо це необхідно.

Коли все більше і більше користувачів починають використовувати систему, використання ресурсів також збільшується, як правило, лінійно. Коли один сервер досягає своїх меж навантаження, найпоширенішим рішенням проблеми є масштабування шляхом додавання нових вузлів, отже, більшої загальної потужності, що дозволяє обробляти більше навантаження, але за ціною дорожчих рахунків за апаратне забезпечення. Такий спосіб розробки корпоративного програмного забезпечення є прийнятним, якщо він задовольняє потреби бізнесу та вимоги. Однак, це не означає, що даний спосіб виконання завдань є найкращим і не найефективнішим.

Один із найпопулярніших фреймворків в екосистемі Java, Spring, вперше представив проект Reactor у 2015 році, який зараз включає реактивний стек з неблокуючим I/O, поступово відходячи від старішого стеку MVC з синхронним блокуючим I/O.

1.4 Реактивне програмування

Реактивне програмування (Reactive Programming) - це парадигма програмування, в якій програма взаємодіє з потоками даних і реагує на зміни. Основна ідея полягає в тому, щоб дозволити компонентам системи автоматично реагувати на зміни в стані та автоматично оновлювати себе відповідно.

1.4.1 Основні переваги та недоліки реактивного програмування

До переваг реактивного програмування належать:

– асинхронність: реактивні системи ефективно використовують асинхронний підхід, що дозволяє обробляти події та зміни без блокування виконання програми;

– декларативність: реактивний код зазвичай є більш декларативним, тобто розробник описує, що потрібно зробити, а не як це робити. Це може полегшити читання і розуміння коду;

– ідентичність стану: реактивні системи зазвичай мають стан, який може автоматично оновлюватися при зміні вхідних даних. Це спрощує управління станом програми;

– складність керування помилками: застосування реактивного програмування може полегшити обробку та маніпулювання помилками в асинхронних операціях.

Однак, реактивне програмування має й ряд недоліків:

– складність вивчення: для деяких розробників перехід до реактивного підходу може бути неспростовним через відмінності в менталітеті та підходах;

– суттєва збільшений обсяг коду: реактивний код може бути більшим за імперативний аналог через введення додаткових концепцій та операторів;

– складність налагодження: відстеження асинхронних потоків та вивчення того, як вони взаємодіють, може зробити налагодження більш складним;

– ризик переповнення подіями (Event Overflow): якщо неправильно реалізовувати, може виникнути надмірне накопичення подій, що може призвести до великої кількості обчислень та споживання ресурсів.

Загалом, реактивне програмування має свої переваги, особливо в асинхронних і розподілених системах, але вибір підходу повинен враховувати конкретні потреби та характеристики проекту.

1.4.2 Порівняння реактивного та імперативного підходу програмування на мові програмування Java

Reactor – це бібліотека реактивного програмування для обробки асинхронних подій.

В імперативному прикладі, що наведено на рисунку 1.6 маємо список чисел, та ітеруємося через нього, подвоюючи кожне число та виводячи результат.

```
import java.util.Arrays;
import java.util.List;

public class ImperativeExample {
    public static void main(String[] args) {
        // Список чисел
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

        // Подвоєння кожного числа та виведення результату
        for (Integer number : numbers) {
            int doubled = number * 2;
            System.out.println(doubled);
        }
    }
}
```

Рисунок 1.6 – Імперативний підхід без використання Reactor

У реактивному прикладі використовується Reactor для створення Flux (реактивний потік) зі списку чисел. Застосовується функція map, яка подвоює

кожне число. Коли визначається ланцюжок операцій, результат автоматично виводиться через метод subscribe (рис. 1.7).

```
import reactor.core.publisher.Flux;

public class ReactiveExample {
    public static void main(String[] args) {
        // Створення Flux зі списку чисел
        Flux<Integer> numbersFlux = Flux.just(1, 2, 3, 4, 5);

        // Подвоєння кожного числа та виведення результату
        numbersFlux
            .map(number -> number * 2)
            .subscribe(System.out::println);
    }
}
```

Рисунок 1.7 – Реактивний підхід з використанням Reactor

Реактивний підхід дозволяє зручно працювати з асинхронними операціями та реагувати на події, забезпечуючи ефективнішу обробку потоків даних. Реакторні бібліотеки, такі як Reactor, допомагають спростити роботу зі створенням, обробкою та об'єднанням реактивних потоків.

2 МЕТОДИ І ЗАСОБИ ПОБУДОВИ ВЕБ-ДОДАТКІВ Й ПОРІВНЯННЯ ЇХ ПРОДУКТИВНОСТІ

2.1 Мова програмування Java

Як вже було встановлено в розділі 1, мова програмування Java знаходиться серед лідерів на ринку в контексті побудови середніх і великих корпоративних веб-додатків. Тому вона є актуальним вибором для проведення порівняльного аналізу продуктивності додатків написаних з використанням різних підходів.

Версія Java вибрана остання зі списку версій з довготривалою підтримкою, 21(LTS) (рис.2.1).

Oracle Java SE Support Roadmap*†

Release	GA Date	Premier Support Until	Extended Support Until	Sustaining Support
8 (LTS)**	March 2014	March 2022	December 2030*****	Indefinite
9 - 10 (non-LTS)	September 2017 - March 2018	March 2018 - September 2018	Not Available	Indefinite
11 (LTS)	September 2018	September 2023	January 2032*****	Indefinite
12 - 16 (non-LTS)	March 2019 - March 2021	September 2019 - September 2021	Not Available	Indefinite
17 (LTS)	September 2021	September 2026****	September 2029****	Indefinite
18 (non-LTS)	March 2022	September 2022	Not Available	Indefinite
19 (non-LTS)	September 2022	March 2023	Not Available	Indefinite
20 (non-LTS)	March 2023	September 2023	Not Available	Indefinite
21 (LTS)	September 2023	September 2028****	September 2031****	Indefinite
22 (non-LTS)***	March 2024	September 2024	Not Available	Indefinite
23 (non-LTS)***	September 2024	March 2025	Not Available	Indefinite
24 (non-LTS)***	March 2025	September 2025	Not Available	Indefinite
25 (LTS)***	September 2025	September 2030	September 2033	Indefinite

Рисунок 2.1 – Інформація про версії Java терміни їх підтримки

2.2 Технології для розробки веб-додатків Spring Framework

Spring Framework є одним із найпопулярніших фреймворків для розробки веб-додатків на мові програмування Java. Це повнофункціональне рішення, яке надає ряд модулів та підходів для побудови різних типів додатків, включаючи веб-додатки. Основними тезами, які характеризують Spring є:

– легкість використання та конфігурації;

– безпека;

– інтеграція;

– модульність та аспектно-орієнтоване програмування;

– підтримка мікросервісної архітектури;

– розширюваність та гнучкість.

Загалом, Spring Framework став стандартом у галузі розробки веб-додатків на мові Java завдяки своїй надійності, гнучкості та розширюваності. Він активно використовується в індустрії та має широку спільноту розробників, яка сприяє його постійному розвитку та підтримці.

Spring MVC та Spring WebFlux модулі є двома альтернативними варіантами для побудови веб-додатків у фреймворку Spring.

Модуль Spring MVC використовується для розробки веб-додатків. Він базується на архітектурі Model-View-Controller. Це синхронний фреймворк, який використовує архітектурний підхід “thread per request design”. Використовується сервлет-контейнер для обробки HTTP-запитів. Тип виклику синхронний, тобто кожен HTTP-запит обробляється в межах одного потоку.

Модуль Spring WebFlux використовується для розробки веб-додатків і є альтернативою Spring MVC. Це реактивний фреймворк, побудований на основі реактивної програмної парадигми. Він використовує асинхронний та неблокуючий підхід для обробки HTTP-запитів. Підтримує асинхронне програмування та реактивні підходи. Може використовувати різні реактивні реалізації, такі як Project Reactor або RxJava. Тип виклику асинхронний, тобто може ефективно обробляти багато HTTP-запитів, використовуючи обмежену кількість програмних потоків.

2.3 Технологія контейнеризації Docker

Docker – це платформа для розробки, доставки та експлуатації програмного забезпечення в контейнерах. Контейнери – це виконувані пакети програмного забезпечення, які включають у себе все необхідне для виконання коду, включаючи системні бібліотеки, залежності, налаштування та код самої програми. Вони ізолюють програмний код від середовища виконання та забезпечують однакове виконання на різних системах.

Основні поняття Docker:

– образ (Image): відображення контейнера, що включає у себе всі необхідні файли та конфігурації для роботи програми;

– контейнер (Container): екземпляр імейки, який виконується в ізольованому середовищі;

– Dockerfile: файл конфігурації, що визначає, як створити імейку;

– Docker Hub: централізований репозитарій образів, до якого можна отримати доступ та від якого можна публікувати образи.

Також, є інструмент для визначення та управління багатоконтейнерними Docker застосунками, що називається Docker Compose. Він дозволяє визначити всі складові додатку (сервіси, мережі, об'єми зберігання тощо) у файлі docker-compose.yml, а потім однією командою створити та запустити весь стек.

Основні поняття Docker Compose:

– docker-compose.yml: YAML-файл, в якому описуються всі сервіси, мережі та інші компоненти додатку;

– сервіс (Service): це контейнер або група контейнерів, які роблять певне завдання;

– мережа (Network): визначає спосіб, як контейнери взаємодіють один з одним;

– volume: забезпечує постійне зберігання даних, доступне контейнерам;

– команди Docker Compose: наприклад, “docker-compose up” для запуску додатку, “docker-compose down” для зупинки та видалення ресурсів.

2.4 Управління залежностями та збіркою проекту Apache Maven

Apache Maven – це інструмент для управління проектами та залежностями в області розробки програмного забезпечення. Maven спрощує процес збірки, залежностей та управління проектом, надаючи стандартизований підхід до створення проектів.

Основні концепції та функціональність Maven:

– POM (Project Object Model): Maven використовує POM для описування проекту. POM – це XML-файл, який містить інформацію про проект, таку як його залежності, конфігурації плагінів, версія проекту та інше;

– збірка (Build): Maven автоматизує процес збірки проекту. За допомогою конфігурації у POM та плагінів Maven може виконувати різні завдання під час збірки, такі як компіляція коду, створення JAR-або WAR-файлів, виконання тестів та інше;

– залежності (Dependencies): Maven дозволяє визначати та управляти залежностями проекту. Залежності описують бібліотеки чи інші компоненти, які використовуються у проекті;

– репозитарії (Repositories): Maven використовує репозитарії для зберігання та отримання бібліотек та інших артефактів. Maven це робить легко завантажувати залежності з центрального репозитарію та інших власних репозитаріїв;

– плагіни (Plugins): Maven розширює свою функціональність за допомогою плагінів. Плагіни можуть виконувати різноманітні завдання під час збірки, тестування, документації та інше;

– цілі (Goals): цілі визначають конкретні завдання, які повинні виконатися плагіном Maven. Наприклад, цілі можуть включати компіляцію коду, створення артефактів, виконання тестів тощо.

Maven дозволяє створювати проекти різного типу (Java, Scala, C# тощо) та спрощує управління їх залежностями та збіркою. Це допомагає підтримувати порядок та стандартизацію в розробці програмного забезпечення.

2.5 Інтегроване середовище розробки IntelliJ IDEA

IntelliJ IDEA – інтегроване середовище розробки (IDE) виробництва компанії JetBrains. IntelliJ IDEA призначено для розробки програмного забезпечення на мовах програмування Java, Kotlin, Groovy, Scala, Android і багатьох інших.

Основні риси IntelliJ IDEA включають:

– розумний редактор коду: IntelliJ IDEA має потужний та інтелектуальний редактор коду, який надає автоматичне доповнення коду, перевірку помилок на льоту, рефакторинг та інші корисні функції;

– система підтримки мов: забезпечує розуміння коду на різних мовах програмування та взаємодію з відповідними інструментами розробки;

– аналіз коду та рефакторинг: IntelliJ IDEA має вбудовані інструменти для аналізу коду, виявлення помилок, а також для проведення різноманітних операцій рефакторингу для покращення структури та якості коду;

– вбудовані інструменти для розробки під Android: IntelliJ IDEA підтримує розробку під Android, включаючи створення Android-додатків;

– система управління версіями: підтримка популярних систем управління версіями, таких як Git, SVN, Mercurial;

– підтримка фреймворків: IntelliJ IDEA інтегрується з різними фреймворками, такими як Spring, Hibernate, Maven, Gradle та інші;

– підтримка тестування: можливість створювати, виконувати та аналізувати тести прямо з інтерфейсу розробки;

– інструменти для веб-розробки: IntelliJ IDEA також надає можливості для розробки веб-додатків, включаючи підтримку HTML, CSS, JavaScript і фреймворки, такі як Angular, React та інші.

IntelliJ IDEA використовується багатьма розробниками як потужний та зручний інструмент для розробки різних типів програмного забезпечення.

2.6 Інструмент для тестування веб-додатків Apache JMeter

Apache JMeter – це вільний, відкритий інструмент для тестування продуктивності та навантаження веб-додатків. JMeter дозволяє вам створювати та виконувати різні види тестів для оцінки продуктивності та стійкості веб-систем.

Основні можливості JMeter включають:

– тестування навантаження: JMeter дозволяє симулювати одночасну роботу багатьох користувачів для вимірювання продуктивності та стійкості веб-додатків під навантаженням;

– створення сценаріїв тестування: ви можете створювати складні сценарії тестування, включаючи послідовності HTTP-запитів, запити до баз даних, завантаження файлів та інші дії;

– підтримка різних протоколів: JMeter підтримує тести для різних протоколів, таких як HTTP, HTTPS, JDBC, JMS, FTP та інші;

– моніторинг ресурсів: інструмент надає можливість моніторити різні метрики ресурсів, такі як використання центрального процесора, обсяг оперативної пам'яті, швидкість передачі даних тощо;

– графічний інтерфейс та можливість скриптингу: JMeter має графічний інтерфейс для створення тестових сценаріїв, а також можливість роботи з XML-подібними файлами конфігурації для більш складних налаштувань;

– розширюваність: ви можете розширювати функціональність JMeter за допомогою плагінів та розширень.

JMeter є корисним інструментом для тестування продуктивності, веб-додатків та служб, і використовується розробниками, тестувальниками та інженерами з навантаження для перевірки ефективності та стійкості додатків перед їх випуском в експлуатацію.

2.7 Інструмент для профілювання та аналізу продуктивності Java-додатків VisualVM

VisualVM (VisualVM profiler) – це інструмент для профілювання та аналізу продуктивності Java-додатків. Він надає графічний інтерфейс для взаємодії з Java-додатками, що використовуються на віртуальній машині Java (JVM). VisualVM є частиною Java Development Kit (JDK) та постачається разом з JDK, починаючи з версії JDK 6 Update 7.

Основні можливості VisualVM включають:

– моніторинг ресурсів: відображення основних метрик продуктивності, таких як використання центрального процесора, обсяг оперативної пам'яті, кількість потоків та інші;

– профілювання коду: VisualVM дозволяє збирати детальну інформацію про роботу додатка, включаючи витрати часу на виконання конкретних методів, використання пам'яті та інші параметри, які можуть бути корисні для виявлення проблем продуктивності;

– моніторинг та керування потоками: відображення інформації про потоки в додатку та можливість їх аналізу та керування;

– аналіз кучі пам'яті (heap): збір інформації про використання Java-кучі пам'яті, виявлення витоків пам'яті, аналіз графіків використання пам'яті;

– взаємодія з MBeans: взаємодія з MBeans для керування та моніторингу Java-додатків через JMX (Java Management Extensions);

– підтримка профілювання додатків на віддалених JVM: VisualVM може бути використаний для профілювання додатків, які виконуються на віддалених серверах.

VisualVM надає розробникам та адміністраторам зручний інтерфейс для вивчення та аналізу продуктивності Java-додатків, допомагаючи виявляти та вирішувати можливі проблеми з продуктивністю та оптимізувати роботу додатків.

2.8 Система керування базами даних PostgreSQL

PostgreSQL – це об'єктно-реляційна система керування базами даних (СКБД), яка відповідає стандартам SQL та визначається як "вільне та відкрите програмне забезпечення". Вона має високий рівень надійності та можливостей, а також розширюється за допомогою багатьох додаткових функцій та розширень.

Основні риси PostgreSQL:

– об'єктно-реляційна модель: PostgreSQL підтримує об'єктно-реляційний підхід до зберігання даних, що дозволяє використовувати реляційні таблиці, а також об'єкти, такі як перегляди, функції та тригери;

– синтаксис SQL: повністю сумісна зі стандартами мова структурованого запиту SQL, що дозволяє розробникам використовувати знайому мову для роботи з базою даних;

– підтримка геоданих: PostgreSQL має вбудовану підтримку для роботи з геоданими, що робить його придатним для геоінформаційних систем (ГІС);

– розширені можливості веб-розробки: велика кількість розширень та допоміжних засобів для розробки веб-додатків;

– транзакції та відновлення: PostgreSQL підтримує транзакції з високим рівнем ізоляції та можливість відновлення бази даних після збоїв;

– розширення через PL/pgSQL: підтримка процедурно-орієнтованих мов програмування, таких як PL/pgSQL, PL/Tcl, PL/Perl, PL/Python, що дозволяє вбудовувати бізнес-логіку в базу даних;

– підтримка JSON: вбудована підтримка для роботи з JSON, включаючи можливість використання JSONB (бінарний JSON) для більш ефективного зберігання та операцій з документами JSON.

PostgreSQL використовується в різних галузях, включаючи веб-розробку, геоінформаційні системи, наукові дослідження та інші області, де важлива висока ефективність та надійність системи керування базами даних.

2.9 Інструмент для керування версіями схеми бази даних Liquibase

Liquibase – це вільний та відкритий інструмент для керування версіями схеми бази даних. Він дозволяє розробникам та адміністраторам вести контроль за структурою бази даних і автоматизувати процеси міграції баз даних. LiquidBase підтримує різні системи керування базами даних, такі як PostgreSQL, MySQL, Oracle, SQL Server та інші.

Основні функції та концепції LiquidBase:

– декларативний підхід: LiquidBase використовує декларативний підхід до змін в базі даних, що означає, що ви описуєте, як має виглядати бажана структура бази даних, а не як змінити її. Це полегшує підтримку та відслідковування версій;

– XML, YAML, SQL, або Groovy для опису змін: LiquidBase дозволяє вам використовувати різні формати для опису міграцій, такі як XML, YAML, SQL, або навіть Groovy-скрипти;

– крос-платформеність: підтримка різних СКБД та платформ, що дозволяє вам використовувати LiquidBase для роботи з різними системами;

– версіювання схеми: LiquidBase веде журнал версій змін схеми бази даних, що дозволяє легко відслідковувати та керувати змінами;

– розгалуження та маркування: можливість використовувати гілки для розгалужень та маркування для організації та управління різними версіями проектів;

– підтримка масштабованості: здатність розгортати міграції в режимі реального часу, щоб уникнути проблем з масштабованістю;

LiquidBase є корисним інструментом для розробників та адміністраторів баз даних, які шукають засіб для ефективного керування версіями та автоматизації міграцій в базах даних.

2.10 Фреймворк для трансформації об’єктів у середині Java програми MapStruct

MapStruct – це фреймворк для генерації коду в мові Java, який спрощує перетворення об'єктів між різними класами (mapping). Зазвичай, такі перетворення виникають, коли вам потрібно скопіювати дані з одного об'єкта в інший, наприклад, при роботі з DTO (Data Transfer Object) та сутностями в програмі.

Основні риси MapStruct:

– анотації: MapStruct використовує анотації для вказівки, які поля або методи слід враховувати при генерації коду для мапінгу;

– генерація коду: фреймворк генерує ефективний та оптимізований код для виконання мапінгу без необхідності написання його вручну;

– підтримка різних типів мапінгу: MapStruct підтримує мапінг між різними типами об'єктів, включаючи вбудовані колекції, дати та інші;

– кастомізація мапінгу: можна використовувати кастомні методи або зовнішні класи для кастомізації мапінгу в тих випадках, коли потрібно впливати на стандартний процес генерації коду.

2.11 Доступ до бази даних з Java програми за допомогою Spring Data

Spring Data – це підпроєкт в екосистемі Spring Framework, який спрощує розробку доступу до даних у різних джерелах даних, таких як реляційні бази даних, NoSQL-сховища, Apache Hadoop та інші. Spring Data надає абстракції та інструменти для створення репозиторіїв даних, керування транзакціями та виконання різних операцій з базами даних.

Один із ключових компонентів Spring Data – це репозиторії (repositories). Репозиторій – це абстракція, яка надає простий інтерфейс для взаємодії з базою даних. Репозиторій визначає методи для збереження, оновлення, видалення та читання об'єктів даних без необхідності писати конкретний SQL-код або взаємодіяти з JDBC безпосередньо.

Spring Data використовує анотації та конвенції для створення реалізацій репозиторіїв автоматично. Основні реалізації репозиторіїв у Spring Data включають:

– JPA (Java Persistence API) репозиторії: для роботи з реляційними базами даних, такими як MySQL, PostgreSQL, Oracle тощо. Ці репозиторії використовують абстракції JPA для збереження та витягування об'єктів;

– MongoDB репозиторії: для роботи з NoSQL-базою даних MongoDB;

– Neo4j репозиторії: для роботи з графовими базами даних, такими як Neo4j;

– Elasticsearch репозиторії: для роботи з Elasticsearch.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ВЕБ-ДОДАТКІВ ІЗ ВИКОРИСТАННЯМ СИНХРОННОЇ ТА АСИНХРОННОЇ МОДЕЛІ ОБРОБКИ HTTP-ЗАПИТІВ

3.1 Функціональність та високорівнева архітектура

Мета цього розділу полягає в розробці, реалізації та тестуванні веб-додатків, використовуючи два різних архітектурних підходи: "thread-per-request" (потік на запит) та "event-driven" (подійно-орієнтований).

Головні цілі цього розділу включають:

– розробка веб-додатків: створення веб-додатків, які будуть мати функціональність яка відповідає потребам сучасних розподілених систем;

– реалізація архітектурних підходів: впровадження архітектурних підходів "thread-per-request" та "event-driven" у веб-додатки. Розробка має бути здійснена з урахуванням кращих практик та стандартів програмування;

– тестування функціональності: вивчення, аналіз та порівняння роботи двох архітектурних підходів щодо ефективності, швидкодії, відмовостійкості та інших ключових характеристик;

– оцінка результатів: проведення об'єктивної оцінки результатів реалізації та тестування, щоб зрозуміти переваги та недоліки кожного архітектурного підходу у конкретному контексті.

Цей розділ має на меті вивчення та порівняння двох різних архітектурних підходів для розробки веб-додатків, що може внести вагомий внесок у розуміння та вибір оптимального підходу при розробці веб-додатків у майбутньому.

Administrative Data Service (ADS) або сервіс надання адміністративних даних, може надавати дані про податки та комунальним платежі платника. ADS реалізований в стилі REST, а саме відповідає таким критеріям як:

– уніфікований ідентифікатор (Uniform Interface): REST використовує уніфікований інтерфейс для взаємодії з ресурсами. Це включає в себе чотири основні види команд для обміну даними, які взаємодіють з ресурсами: HTTP GET (отримання), POST (створення), PUT (оновлення) та DELETE (видалення);

– представлення (Representation): ресурси можуть бути представлені в різних форматах, таких як XML, JSON або HTML;

– відсутність стану (Stateless): кожен запит від клієнта до сервера повинен містити всю необхідну інформацію для розуміння та обробки запиту. Сервер не повинен зберігати стан клієнта між запитами.

Сервіс складається з трьох системних вузлів. Java веб-додаток administrative-data-service приймає HTTP-запити від користувача, та формує HTTP-відповіді в форматі JSON. Також, збирає дані про податки та комунальні платежі зі сторонніх систем PostgreSQL та ads-utility-bills-service.

Java веб-додаток ads-utility-bills-service зберігає дані про комунальні платежі в пам’яті програми та формує HTTP відповіді з таких даних на запит.

Реляційна база даних PostgreSQL зберігає дані про податки платників у таблиці tax (рис 3.1).

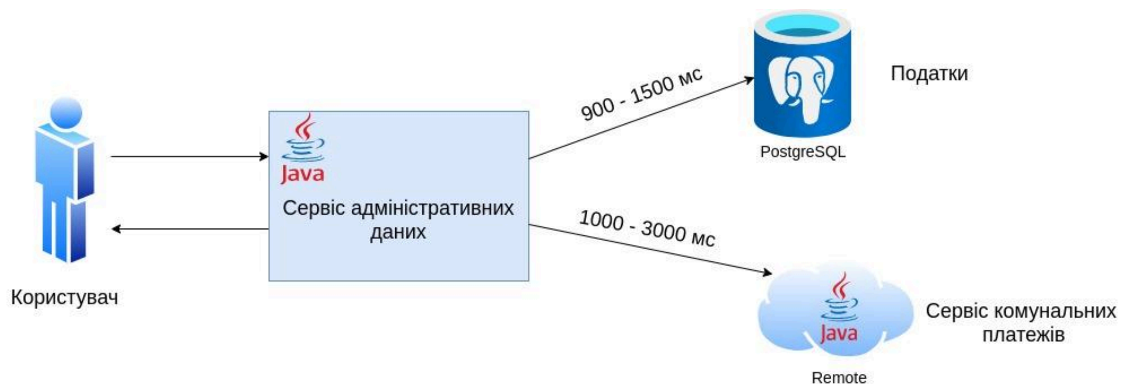


Рисунок 3.1 – Схема високорівневої архітектури системи

3.2 Налаштування та заповнення даними таблиці бази даних PostgreSQL

Для зручного та швидкого запуску PostgreSQL використовується технологія контейнеризації Docker та офіційний образ PostgreSQL з Docker Hub для запуску PostgreSQL в контейнері. Для запуску використовується команда “docker-compose up”.

Даний підхід дозволяє зробити базу даних ізольованою від програмного середовища комп’ютера де вона буде запускатися. Це дає гарантію того, що база даних буде працювати однаково на всіх комп’ютерах. Нижче показаний файл docker-compose.yml (рис. 3.2)

```
1  version: '3.3'
2  services:
3    postgress-db:
4      container_name: postgress-db
5      image: postgres:12-alpine
6      restart: on-failure
7      ports:
8        - 5432:5432
9      networks:
10       - default
11     environment:
12       POSTGRES_USER: postgres
13       POSTGRES_PASSWORD: password
14       POSTGRES_DB: taxdb
15
```

Рисунок 3.2 – Файл docker-compose.yml

Наступним кроком є підготовка файла з даними для заповнення таблиці tax. Для цього було написано міні програму, що складається з одного Java класу. Задача цієї програми генерувати файл у форматі csv з тестовими даними. Кількість рядків у файлі дорівнює один мільйон, що відповідає одному мільйону записів про неоплачені податки в базі даних. Така кількість записів потрібна щоб відтворити систему, схожу на реальну (додаток А). Приклад змісту сгенерованого файлу наведено на рисунку 3.3.

```
"id","type","name","person_name","payment_amount","create_time"
"1","Податок","Податок з продажу рухомого майна","Васильчук Олена Валентинівна","2382.59","2021-08-27 22:57:39"
"2","Податок","Воєнний збір","Дьяченко Михайло Костянтинович","353.84","2020-01-23 03:00:59"
"3","Податок","Воєнний збір","Дьяченко Михайло Костянтинович","2960.58","2019-09-15 22:15:47"
"4","Податок","Податок з продажу рухомого майна","Гнатюк Анна Олексіївна","1623.94","2020-01-02 08:56:00"
"5","Податок","Податок з доходу фізичних осіб","Дьяченко Михайло Костянтинович","2950.66","2019-08-26 19:58:52"
"6","Податок","Податок з доходу фізичних осіб","Лисенко Володимир Федорович","2204.34","2023-03-20 21:59:36"
"7","Податок","Воєнний збір","Гнатюк Анна Олексіївна","689.89","2019-09-21 22:54:03"
"8","Податок","Воєнний збір","Романченко В'ячеслав Анатолійович","2909.96","2020-01-12 17:43:54"
"9","Податок","Податок з доходу фізичних осіб","Пономаренко Наташа Олексіївна","2297.41","2023-01-09 11:23:41"
"10","Податок","Податок з доходу фізичних осіб","Лисенко Володимир Федорович","2874.07","2022-07-10 16:42:19"
```

Рисунок 3.3 – Зміт згенерованого csv файлу

Для завантаження згенерованих даних в PostgreSQL використовується Liquibase. Підключення коду Java бібліотеки Liquibase буде наведено в наступних підрозділах. На рисунку 3.4 зображена xml розмітка файлу конфігурації, на базі якого буде створена таблиця tax, після чого виконана трансляція даних з csv файлу до таблиці у базі даних.

```
<?xml version="1.1" encoding="UTF-8" standalone="no"?>
<databaseChangeLog xmlns="http://www.liquibase.org/xml/ns/dbchangelog"
                   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                   xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog
                   http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-3.5.xsd">
    <changeSet author="mdiachenko" id="create_table_tax">
        <comment>Taxes</comment>
        <createTable tableName="tax" remarks="Tax information table">
            <column name="id" type="SERIAL" remarks="Record id">
                <constraints nullable="false" primaryKey="true" primaryKeyName="pk_tax"/>
            </column>
            <column name="type" type="VARCHAR(255)" remarks="Tax type">
                <constraints nullable="false"/>
            </column>
            <column name="name" type="VARCHAR(255)" remarks="Tax record name">
                <constraints nullable="false"/>
            </column>
            <column name="person_name" type="VARCHAR(255)" remarks="Recipient name">
                <constraints nullable="false"/>
            </column>
            <column name="payment_amount" type="DECIMAL(12,2)" remarks="Payment amount">
                <constraints nullable="false"/>
            </column>
            <column name="create_time" type="TIMESTAMP WITHOUT TIME ZONE" remarks="Date of creation">
                <constraints nullable="false"/>
            </column>
        </createTable>
    </changeSet>
    <changeSet author="mdiachenko" id="bootstrap_tax_table">
        <loadData file="db/tax.csv" tableName="tax"/>
    </changeSet>
</databaseChangeLog>
```

Рисунок 3.4 – Конфігураційний файл Liquibase

На рисунку 3.5 показано стан таблиці tax в базі даних після загрузки з csv файла.

Рисунок 3.5 – Стан таблиці tax в базі даних після загрузки з csv файла

3.3 Розробка веб-додатку

3.3.1 Мікросервіс ads-utility-bills-service

Сервіс ads-utility-bills-service не є основним та представляє собою програмний вузол, з яким основний сервіс буде інтегруватися. Використовуючи найпопулярніші технології для створення веб-додатків на мові Java, а саме Spring WebMVC і Spring Boot, проводиться його реалізація.

Першим кроком є створення каркасу проекта з усіма необхідними залежностями. Для цього використано spring initializr – онлайн інструмент для генерації Java проектів на базі модулів фреймворку Spring (рис. 3.6).

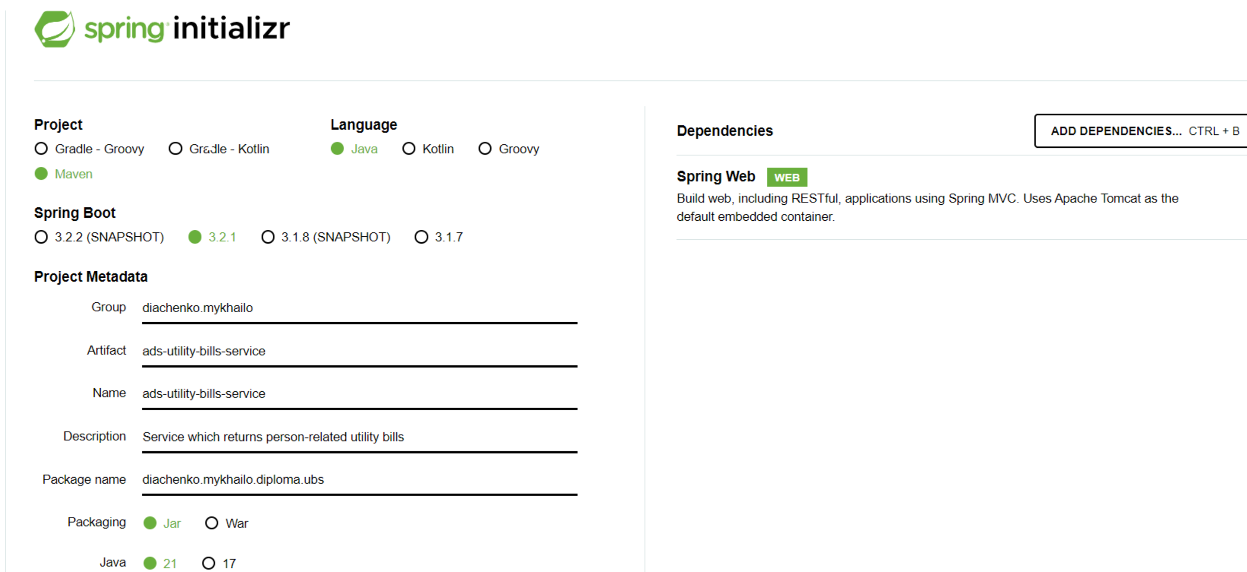


Рисунок 3.6 – Spring initializr

Після того як каркас проекту створений, додано потрібний функціонал. Цей мікросервіс складається з декількох класів та файлів конфігурації, структура проекту наведена на рисунку 3.7.

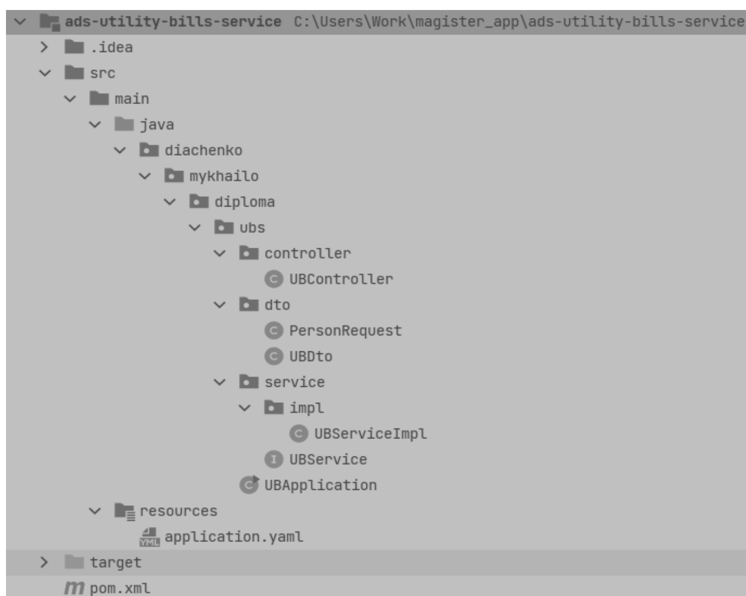


Рисунок 3.7 – Структура проекту ads-utility-bills-service

diachenko.mykhailo.diploma.ubs.controller – пакет, який містить класи контролери. Методи класів контролерів є точками входу HTTP-запита. В них виконуються такі операції як валідація вхідних даних, трансформація тіла запиту у DTO об’єкти та делегування виконання класам рівня сервісів;

diachenko.mykhailo.diploma.ubs.dto – пакет з класами DTO (data transfer object), це класи, які використовуються для передачі тієї чи іншої інформації всередині Java програми. Вони не містять логіки обробки даних, тільки самі дані;

diachenko.mykhailo.diploma.ubs.service – пакет, що містить класи з логікою. Тут виконується основний процес обробки та збору даних. Реалізація класу diachenko.mykhailo.diploma.ubs.service.impl.UBServiceImpl показана в додатку Б. Цей клас компонує дані про комунальні платежі особи, ім’я якої передається в параметрах HTTP-запиту. Далі контролер повертає ці дані в форматі JSON до споживача, яким виступає інший Java веб-додаток administrative-data-service.

Мікросервіс має один єдиний HTTP API для отримання даних про комунальні платежі особи в залежності від переданого ПІБ (рис. 3.8).

```
@GetMapping("/utility-bills")
public List<UBDto> getUtilityBills (@RequestParam String name) throws InterruptedException {
    Thread.sleep(random.nextInt( origin: 1000,  bound: 3000));
    return service.getUtilityBillsFines(name);
}
```

Рисунок 3.8 – Метод класу контролеру

Приклад HTTP-запиту на цю адресу виглядає наступним чином:

GET {protocol}://{host}:{port}/utility-bills?name=Дьяченко Михайло Костянтинович.

Тіло HTTP відповіді виглядає як показано на рисунку 3.9.

```
[
    {
        "id": 1,
        "type": "Комунальний платіж",
        "name": "Рахунок за вивіз сміття",
        "personName": "Дьяченко Михайло Костянтинович",
        "paymentAmount": 600.0645780212185,
        "createTime": "2022-09-28 23:11:33"
    },
    {
        "id": 2,
        "type": "Комунальний платіж",
        "name": "Рахунок за постачання газу",
        "personName": "Дьяченко Михайло Костянтинович",
        "paymentAmount": 191.3683879285978,
        "createTime": "2019-08-12 10:52:35"
    },
    {
        "id": 3,
        "type": "Комунальний платіж",
        "name": "Рахунок за постачання електроенергії",
        "personName": "Дьяченко Михайло Костянтинович",
        "paymentAmount": 364.863708889509,
        "createTime": "2020-07-18 22:10:48"
    },
```


Рисунок 3.9 – Тіло HTTP відповіді

3.3.2 Створення веб-додатку administrative-data-service

Спочатку реалізовано цей сервіс з використанням моделі “thread per request”, що описана в розділі 2, а саме на технології Spring WebMVC і Spring Boot. Spring WebMVC за замовчуванням поставляється с Java веб-сервером Apache Tomcat, який саме і реалізує модель “thread per request”.

Одним з кроків є створення каркасу проекту з усіма необхідними залежностями. Для цього використовується spring initializr – онлайн інструмент для гереації Java проектів на базі модулей фреймворка Spring. Зміст файлу керування залежностями та процесом збірки проекту pom.xml наведений у додатку В. Структура проекту показана на рисунку 3.10.

Рисунок 3.10 – Структура проекту administrative-data-service

У Spring Boot фреймворку, файл application.yaml – це файл конфігурації, який використовується для налаштування параметрів додатка на основі Spring Boot. Цей файл містить конфігурацію з’єднання з базою даних PostgreSQL, адресу стороннього HTTP-сервісу та налаштування логування (рис. 3.11).

```yaml
spring:
  application:
    name: administrative data service
  datasource:
    url: jdbc:postgresql://localhost:5432/taxdb
    username: postgres
    password: password
    hikari:
      minimumIdle: 2
      maximumPoolSize: 50
      idleTimeout: 60000
      connectionTimeout: 10000
  liquibase:
    change-log: classpath:db/db_bootstrap.xml
logging:
  level:
    diachenko:
      mykhailo:
        diploma: INFO
ads:
  utilityBills:
    uri: http://localhost:8090/utility-bills
```

Рисунок 3.11 – Конфігураційний файл Spring application.yaml

Наступним кроком буде правильне розташування файлів конфігурації Liquibase у директорії ресурсів проекту (рис. 3.12).

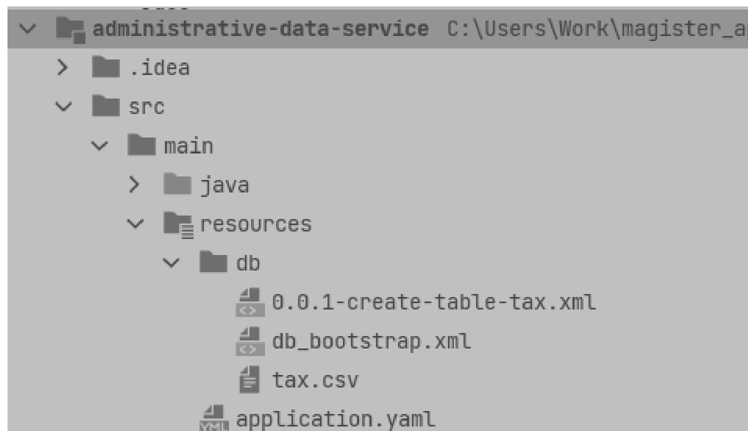


Рисунок 3.12 – Розташування файлів конфігурації Liquibase у директорії ресурсів проекту

Зміст файлів було розглянуто в розділі 3.2. Liquibase виконає інструкції описані в конфігураційному файлі 0.0.1-create-table-tax.xml і створить таблицю tax з даними в базі даних PostgreSQL (рис. 3.4, рис. 3.5). При наступному запуску програми операції бази даних, які вже були виконані раніше, повторно виконані не будуть. Liquibase підтримує механізм контролю версій і зрозуміє, що така версія таблиці вже існує.

Далі можемо починати описувати логіку програми за допомогою Java класів. Фінальна структура класів виглядає як показано на рисунку 3.13.

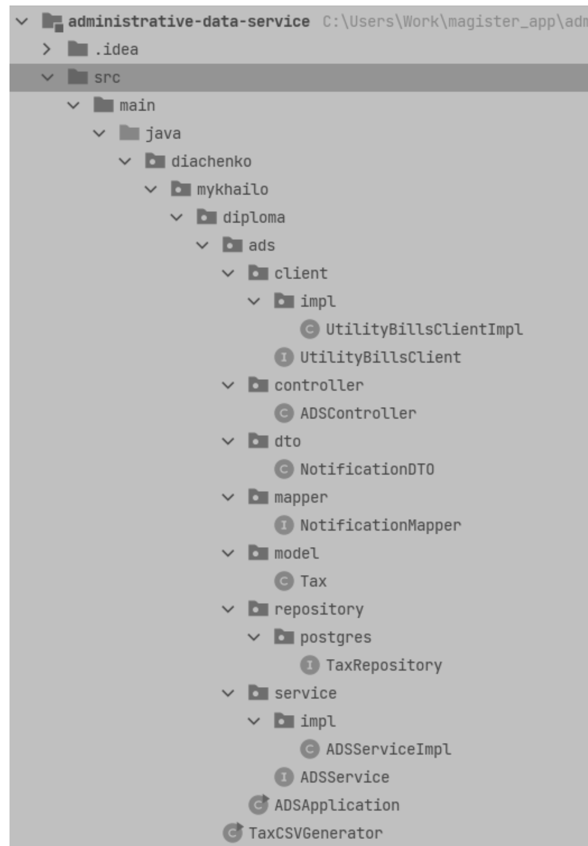


Рисунок 3.13 – Структура файлів проекту administrative-data-service

Пакет diachenko.mykhailo.diploma.ads.controller містить класи контролерів. Методи даних класів є точками входу HTTP-запита. В них виконуються такі операції як валідація вхідних даних, трансформація тіла запиту у DTO об'єкти та делегування виконання класам рівня сервісів.

Пакет diachenko.mykhailo.diploma.ads.dto містить класи DTO (data transfer object), які використовуються для передачі тієї чи іншої інформації всередині Java програми. Вони не містять логіки обробки даних, тільки самі дані.

Пакет diachenko.mykhailo.diploma.ads.service містить класи з логікою, де виконується основний процес обробки та збору даних. Реалізація класу diachenko.mykhailo.diploma.ads.service.impl.ADSServiceImpl показана в додатку Г. Цей клас компонує дані про комунальні платежі та податки особи, ім'я якої передається в параметрах HTTP-запиту. Далі контролер повертає ці дані в форматі JSON до споживача;

Пакет diachenko.mykhailo.diploma.ads.client містить класи та інтерфейси відповідні за комунікацію зі сторонніми програмами, в нашому випадку з ads-utility-bills-service. Спілкування реалізоване через org.springframework.web.client.RestTemplate Java клас, який надає зручний спосіб взаємодії з веб-сервісами за допомогою HTTP-запитів і є частиною модуля spring-web. Реалізацію показана на рисунку 3.14.

```
@Override
public List<UBDto> getUtilityBills (PersonRequest person) {
    String uri = UriComponentsBuilder.fromHttpUrl(utilityBillsGetURI) UriComponentsBuilder
            .queryParam( name: "name", person.getPersonName())
            .build( encoded: false) UriComponents
            .toUriString();
    log.info("utility bills service URI:{}", uri);
    String responseBody = new RestTemplate().getForEntity(uri, String.class)
            .getBody();
    try {
        return objectMapper.readValue(responseBody, new TypeReference<>() {
        });
    } catch (JsonProcessingException e) {
        log.error("Unable to parse utility-bills-service response due to {}", e.getMessage());
        return List.of();
    }
}
```

Рисунок 3.14 – Метод класу diachenko.mykhailo.diploma.ads.clientUtilityBillsClientImpl

Пакет diachenko.mykhailo.diploma.ads.repository містить Java інтерфейси, які виступають в ролі репозиторіїв – абстракцій для взаємодії з базою даних. Реалізацію цих абстракцій створює модуль Spring Data під час запуску програми.

Пакет diachenko.mykhailo.diploma.ads.mapper містить інтерфейси позначені анотацією з фреймворку MapStruct org.mapstruct.Mapper, з методами для перетворення одних DTO в інші.

Після запуску програми ми можемо бачити наступні логи (рис. 3.15), які свідчать про те, що програма успішно запущена на веб-сервері Apache Tomcat і доступна на мережевому порту під номером 8080.

```
00  INFO 5712 --- [administrative data service] [           main] liquibase.changelog                      : Reading from public.databasechangelog
00  INFO 5712 --- [administrative data service] [           main] liquibase.util                           : UPDATE SUMMARY
00  INFO 5712 --- [administrative data service] [           main] liquibase.util                           : Run:                          0
00  INFO 5712 --- [administrative data service] [           main] liquibase.util                           : Previously run:               2
00  INFO 5712 --- [administrative data service] [           main] liquibase.util                           : Filtered out:                 0
00  INFO 5712 --- [administrative data service] [           main] liquibase.util                           : -------------------------------
00  INFO 5712 --- [administrative data service] [           main] liquibase.util                           : Total change sets:            2
00  INFO 5712 --- [administrative data service] [           main] liquibase.util                           : Update summary generated
00  INFO 5712 --- [administrative data service] [           main] liquibase.lockservice                    : Successfully released change log lock
00  INFO 5712 --- [administrative data service] [           main] liquibase.command                        : Command execution complete
00  INFO 5712 --- [administrative data service] [           main] o.hibernate.jpa.internal.util.LogHelper  : HHH000204: Processing
 default]
00  INFO 5712 --- [administrative data service] [           main] org.hibernate.Version                    : HHH000412: Hibernate ORM core version

00  INFO 5712 --- [administrative data service] [           main] o.h.c.internal.RegionFactoryInitiator    : HHH000026: Second-level cache disabled
00  INFO 5712 --- [administrative data service] [           main] o.s.o.j.p.SpringPersistenceUnitInfo      : No LoadTimeWeaver setup: ignoring JPA

00  INFO 5712 --- [administrative data service] [           main] o.h.e.t.j.p.i.JtaPlatformInitiator       : HHH000489: No JTA platform available
n.jta.platform' to enable JTA platform integration)
00  INFO 5712 --- [administrative data service] [           main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory
ult'
00  WARN 5712 --- [administrative data service] [           main] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by
se queries may be performed during view rendering. Explicitly configure spring.jpa.open-in-view to disable this warning
00  INFO 5712 --- [administrative data service] [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port 8080 (http)

00  INFO 5712 --- [administrative data service] [           main] d.mykhailo.diploma.ads.ADSApplication    : Started ADSApplication in 22.716
or 25.196)
```

Рисунок 3.15 – Логи запуску administrative-data-service

3.3.3 Тестування додатку administrative-data-service працюючого на базі “thread per request” моделі

Спочатку виміряємо кількість програмних потоків, показники використання CPU та heap пам’яті без навантаження що наведено на рисунках 3.16, 3.17 та 3.18 відповідно.

Рисунок 3.16 – Кількість програмних потоків

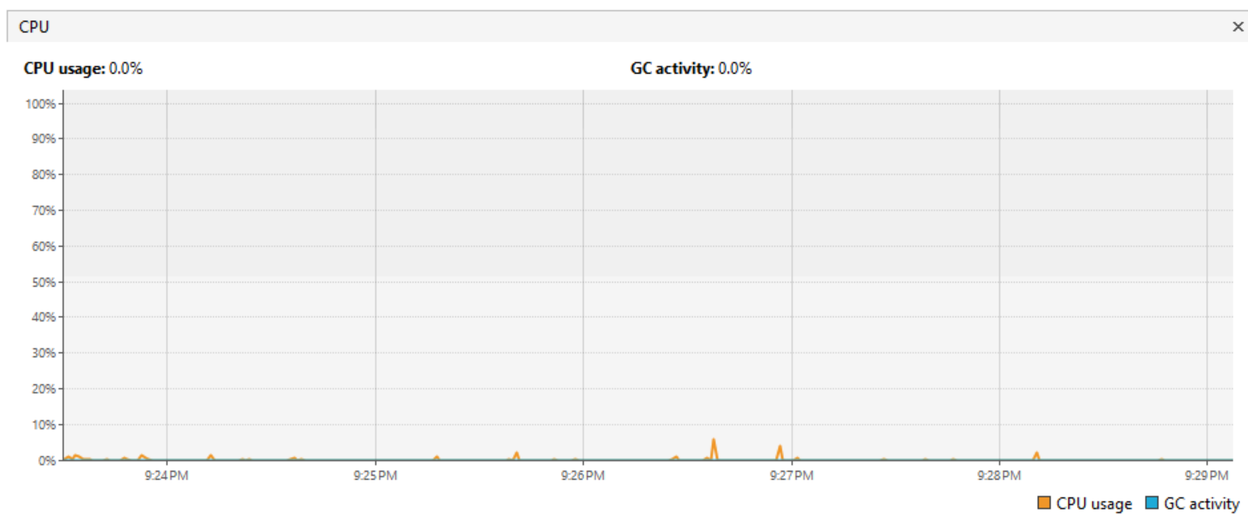


Рисунок 3.17 – Показники використання CPU

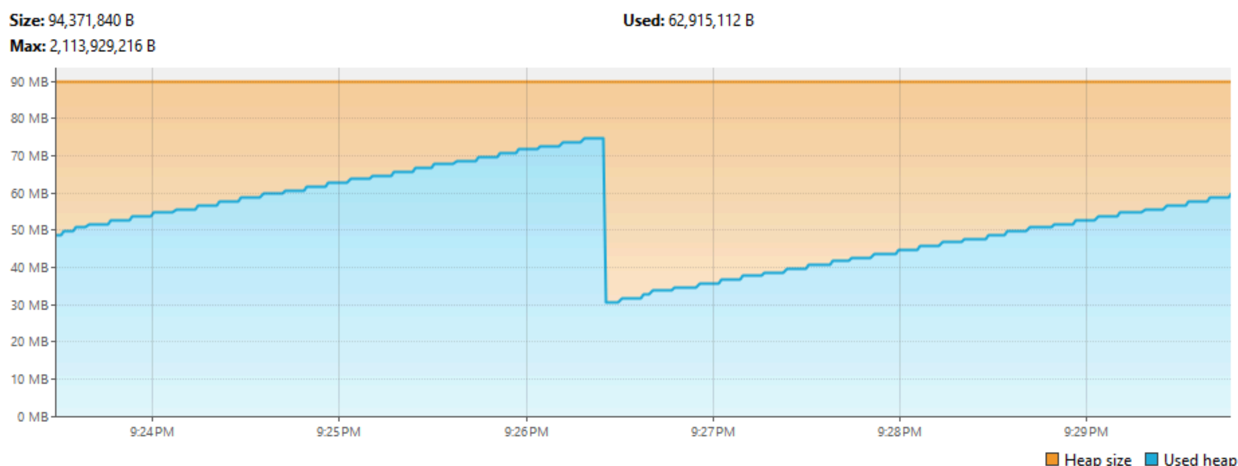


Рисунок 3.18 – Показники використання heap пам’яті

Тест №1 – це 30 HTTP-запитів у секунду на сервер впродовж 30 секунд. Всього 900 запитів. Тут бачимо, що Tomcat створив додаткові потоки, і тепер їх 30, що дорівнює кількості одночасних HTTP-запитів. Також видно, що час від часу кожний з потоків знаходиться в режимі очікування (помаранчевий колір на рис. 3.19).

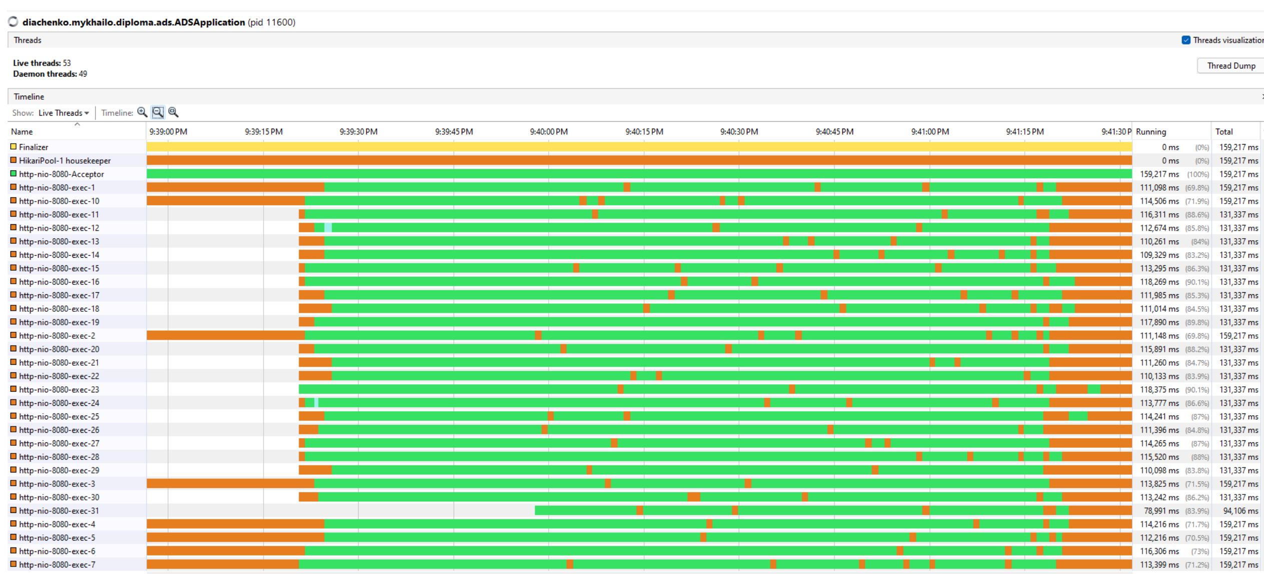


Рисунок 3.19 – Стан програмних потоків під час виконання тесту №1

Рівень використання CPU тримався в межах 20–40% переважну більшість часу, і максимальний був 68% (рис. 3.20).

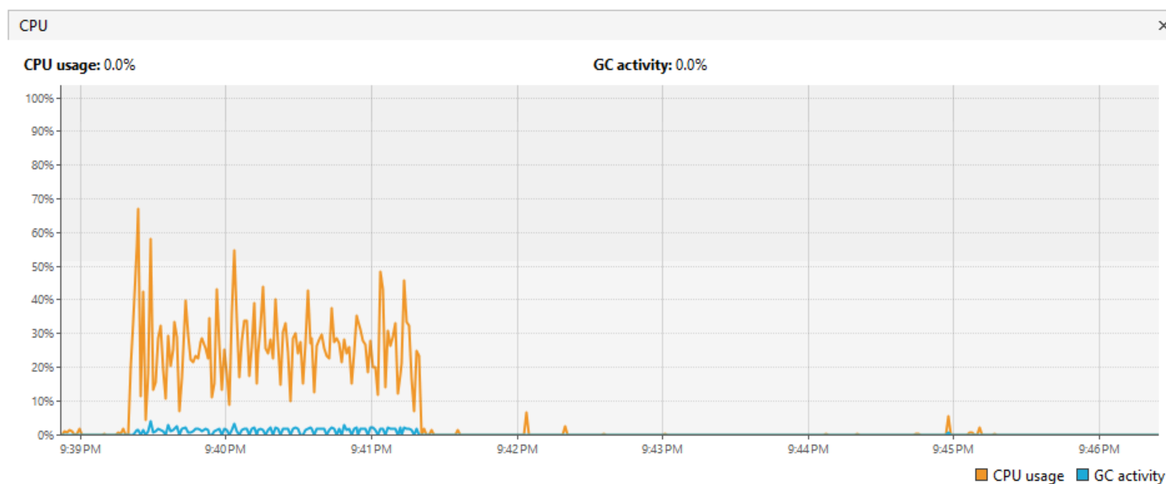


Рисунок 3.20 – Рівень використання CPU під час виконання тесту №1

Рівень пам’яті heap у використанні тримався на велечині 750–1480 мегабайт (рис. 3.21).

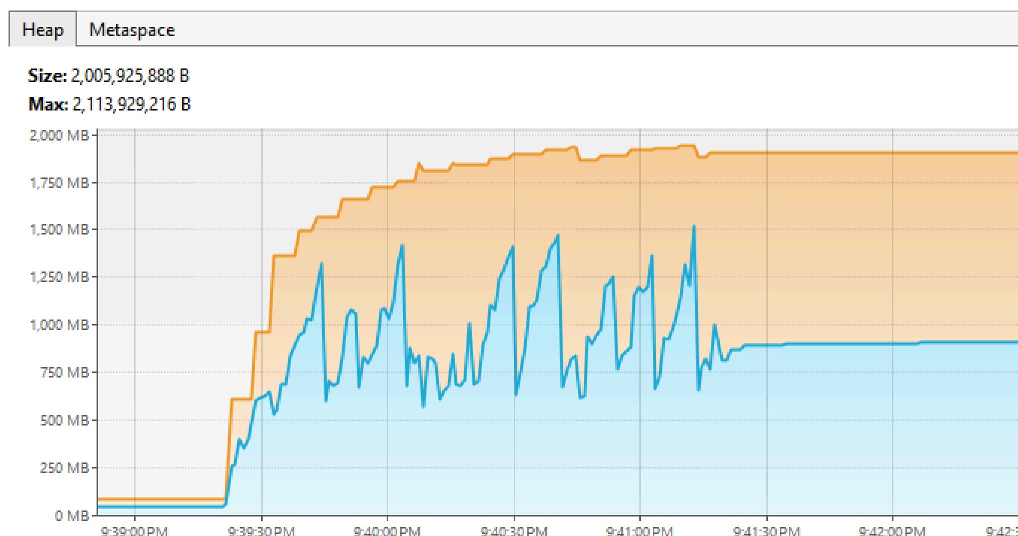


Рисунок 3.21 – Рівень пам’яті heap під час виконання тесту №1

Результати тесту №1 з Apache JMeter. Кількість запитів 900. Середня швидкість відповіді в мілісекундах 3959. Швидкість відповіді на 90% запитів менша ніж 5396 мілісекунд. Показник помилок 0%. Час виконання тесту 2 хв 6 сек (рис 3.22).

Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput
HTTP Request	900	3959	3842	5396	5918	6942	1293	10731	0.00%	7.1/sec
TOTAL	900	3959	3842	5396	5918	6942	1293	10731	0.00%	7.1/sec

Рисунок 3.22 – Результати з Apache JMeter , тест №1

Тест №2 – це 1000 одночасних запитів одноразово. З результатів профайлера видно, що Tomcat збільшив кількість потоків до 200. На цей раз час, який кожен потік очікує і не працює також значно збільшився (помаранчевий колір на рисунку 3.23).

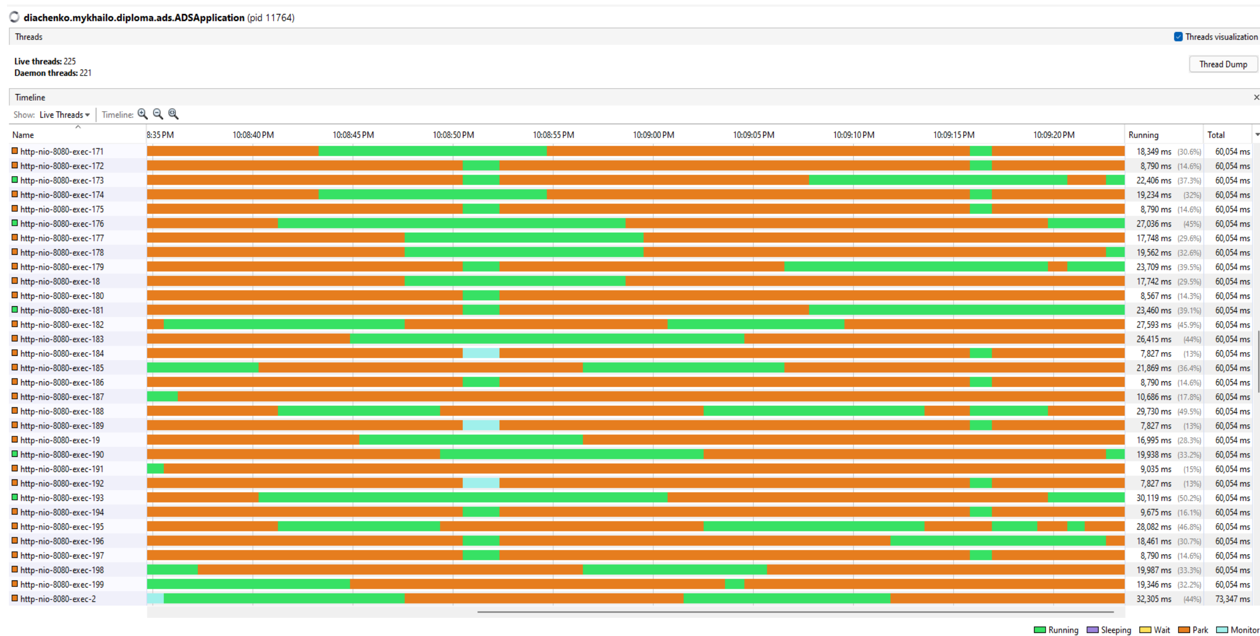


Рисунок 3.23 – Стан програмних потоків під час виконання тесту №2

Використання CPU залишилось без значних змін, адже потужності фізичного процесора і кількість ядер не змінилась (рис. 3.24).

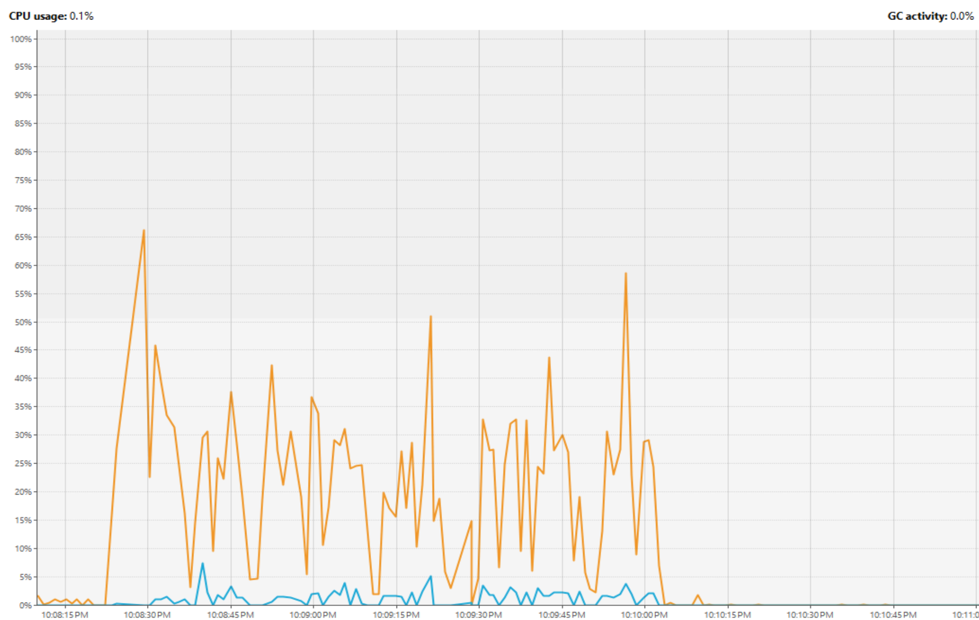


Рисунок 3.24 – Рівень використання CPU під час виконання тесту №2

Рівень пам’яті heap у використанні постійно зростав та дорівнював 800–1200 мегабайт більшу частину часу обробки запитів (рис. 3.25).

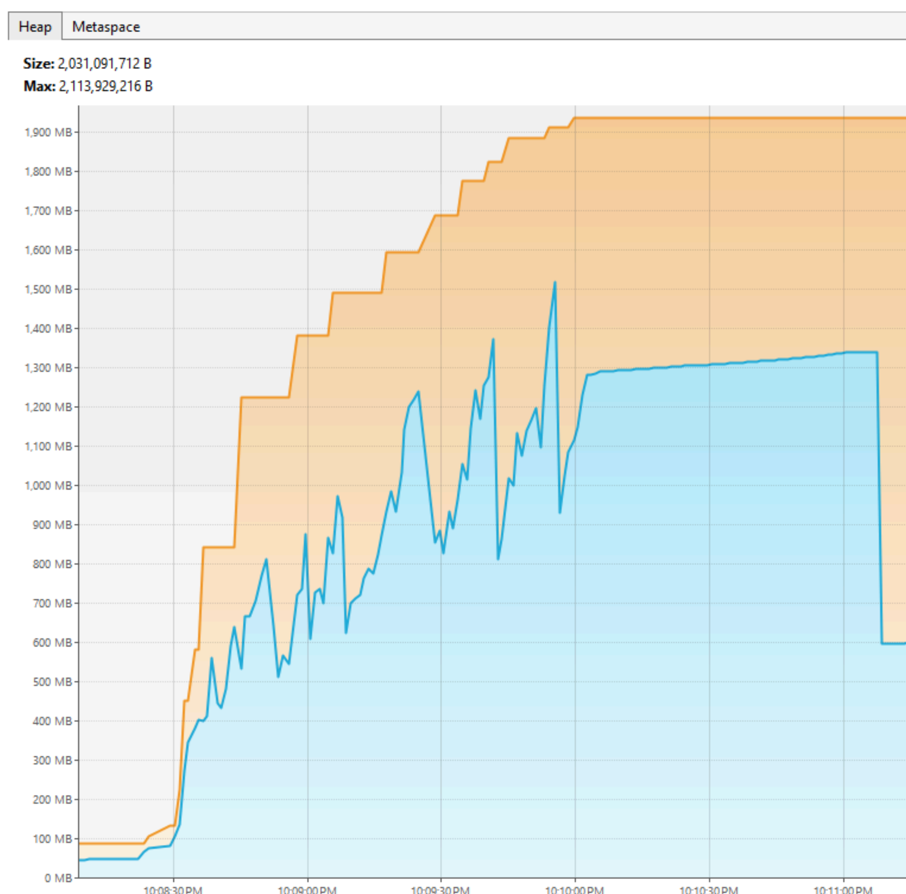


Рисунок 3.25 – Рівень пам’яті heap під час виконання тесту №2

Результати тесту №2 з Apache JMeter. Кількість запитів 1000. Середня швидкість відповіді в мілісекундах 41423. Швидкість відповіді на 90% запитів менша ніж 86989 мілісекунд. Показник помилок 59.3%. Час виконання тесту 1хв 42сек. Всі помилки пов’язані з перевищенням часу очікування з’єднання з базою даних, який становить 20000 мілісекунд (рис. 3.26)

Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput
HTTP Request	1000	41423	39409	86989	94031	100215	14	101258	59.30%	9.8/sec
TOTAL	1000	41423	39409	86989	94031	100215	14	101258	59.30%	9.8/sec

Рисунок 3.26 – Результати з Apache JMeter, тест №2

Враховуючи великий відсоток HTTP-запитів які були виконані з помилками, має сенс повторити тест №2 збільшивши максимальний час очікування з’єднання з

базою даних, для отримання більш чистих результатів часу виконання HTTP-запитів.

Результати другої спроби тесту №2 з Apache JMeter. Кількість запитів 1000. Середня швидкість відповіді в мілісекундах 80778. Швидкість відповіді на 90% запитів менша ніж 131049 мілісекунд. Показник помилок 0%. Час виконання тесту 2хв 26сек (рис. 3.27).

Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %
HTTP Request	1000	80778	83177	131049	135590	138391	7310	140394	0.00%
TOTAL	1000	80778	83177	131049	135590	138391	7310	140394	0.00%

Рисунок 3.27 – Результати з Apache JMeter, тест №2 повтор

3.3.4 Модификація веб-додатку administrative-data-service

Змінимо реалізацію програми але не її логіку. По-перше, потрібно підключити залежності на інші Java модулі. Видаляємо spring-boot-starter-web, додаємо замість нього spring-boot-starter-webflux. (рис. 3.28).

```
project > dependencies > dependency
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.postgresql</groupId>
        <artifactId>postgresql</artifactId>
    </dependency>
    <dependency>
```

```
28          <exclusions>
29              <exclusion>
30                  <groupId>org.springframework.boot</groupId>
31                  <artifactId>spring-boot-starter-web</artifactId>
32              </exclusion>
33          </exclusions>
34      </dependency>
35      <dependency>
36          <groupId>org.springframework.boot</groupId>
37          <artifactId>spring-boot-starter-webflux</artifactId>
38      </dependency>
39      <dependency>
40          <groupId>org.postgresql</groupId>
41          <artifactId>postgresql</artifactId>
42      </dependency>
```

Рисунок 3.28 – Зміни в файлі pom.xml

Далі необхідно змінити клас контролер, таким чином щоб він відповідав реактивному стилю та використовував класи з модуля Spring WebFlux. Тепер методи контролера мають повертати тип reactor.core.publisher.Mono або reactor.core.publisher.Flux в залежності від кількості асинхронно отриманих елементів у відповіді, Mono=0…1, Flux=0…n. Метод

reactor.core.publisher.Mono#map – це потоковий метод для перетворення даних цього потоку, лямбда функція, яка передається як аргумент цього метода буде визвана тільки тоді, коли такі дані з'являться в потоці. Це і є неблокуючий (асинхронний) підхід, програмний потік не блокується чекаючи на дані як це було в минулій реалізації (рис. 3.29).

```
@RestController
@RequiredArgsConstructor
public class ADSController {

    private final ADSService service;

    @GetMapping(value = "/notifications", produces = MediaType
     .APPLICATION_JSON_VALUE)
    @ResponseStatus(HttpStatus.OK)
    public Page<NotificationDTO> getNotifications (@RequestParam
      String name) {
        PersonRequest person = new PersonRequest(name);
        List<NotificationDTO> notifications = service
         .getNotifications(person);
        int pageSize = Math.min(notifications.size(), 20);
        if(pageSize == 0) {
            pageSize = 1;
        }
        return new PageImpl<>(notifications.subList(0,
         pageSize), Pageable.ofSize(pageSize), notifications
         .size());
    }
}
```

```
@RestController
@RequiredArgsConstructor
public class ADSAController {

    private final ADSAService service;

    @GetMapping(value = "/notifications", produces = MediaType
     .APPLICATION_JSON_VALUE)
    @ResponseStatus(HttpStatus.OK)
    public Mono<PageImpl<NotificationDTO>> getNotifications (@RequestParam
     String name) {
        PersonRequest person = new PersonRequest(name);
        return
                service.getNotifications(person)
                        .collectList()
                        .map(notifications -> {
                            int pageSize = Math.min(notifications.size(),
                             20);
                            if(pageSize == 0) {
                                pageSize = 1;
                            }
                            return new PageImpl<>(notifications.subList(0,
                             pageSize), Pageable.ofSize(pageSize),
                             notifications.size());
                        });
```

Рисунок 3.29 – Зміни в класі diachenko.mykhailo.diploma.ads.controller.ADSController

Схожим чином змінюємо класи в пакеті diachenko.mykhailo.diploma.ads.service (рис. 3.30).

```
    @Override
    public List<NotificationDTO> getNotifications (PersonRequest
     person) {
        List<NotificationDTO> taxes = getTaxes(person);
        log.info("Taxes obtained. Size={}", taxes.size());
        List<NotificationDTO> utilityBillsFines =
         getUtilityBillsFines(person);
        log.info("Utility bills fines obtained. Size={}",
         utilityBillsFines.size());
        ArrayList<NotificationDTO> resultList = new ArrayList<>();
        resultList.addAll(taxes);
        resultList.addAll(utilityBillsFines);
        return resultList;
```

```
    @Override
    public Flux<NotificationDTO> getNotifications (PersonRequest
     person) {
        return getTaxes(person)
                .flatMapIterable(resp -> resp)
                .mergeWith(getUtilityBills(person).flatMapIterable
                 (resp -> resp));
    }
```

Рисунок 3.30 – Зміни в класі diachenko.mykhailo.diploma.ads.service.impl.ADSServiceImpl

Змінюємо реалізацію класів в пакеті diachenko.mykhailo.diploma.ads.client на реактивну (рис. 3.31).

```java
    @Value("${ads.utilityBills.uri}")
    private String utilityBillsGetURI;
    private final ObjectMapper objectMapper;

    @Override
    public List<UBDto> getUtilityBills (PersonRequest person) {
        String uri = UriComponentsBuilder.fromHttpUrl
         (utilityBillsGetURI)
                .queryParam("name", person.getPersonName())
                .build(false)
                .toUriString();
        log.info("utility bills service URI:{}", uri);
        String responseBody = new RestTemplate().getForEntity(uri,
         String.class)
                .getBody();
        try {
            return objectMapper.readValue(responseBody, new
             TypeReference<>() {
            });
        } catch (JsonProcessingException e) {
            log.error("Unable to parse utility-bills-service
             response due to {}", e.getMessage());
            return List.of();
        }
    }
}
```

```java
    @Value("${ads.utilityBills.uri}")
    private String utilityBillsGetURI;
    private final WebClient webClient;

    @Override
    public Mono<List<UBDto>> getUtilityBills (PersonRequest person) {
        String uri = UriComponentsBuilder.fromHttpUrl
         (utilityBillsGetURI)
                .queryParam("name", person.getPersonName())
                .build(false)
                .toUriString();
        log.info("utility bills service URI:{}", uri);

        return webClient.get()
                .uri(uri)
                .exchangeToMono(resp -> {
                    if(resp.statusCode().is2xxSuccessful()) {
                        return resp.bodyToMono(new
                         ParameterizedTypeReference<>() {
                        });
                    }
                    return resp.bodyToMono(String.class)
                            .doOnNext(log::error)
                            .then(Mono.error(new RuntimeException
                             (String.valueOf(resp.statusCode().value
                             ()))));
                });
    }
}
```

Рисунок 3.31 – Зміни в класі diachenko.mykhailo.diploma.ads.client.impl.UtilityBillsClientImpl

Для спілкування веб-сервісів через HTTP-запити будемо використовувати org.springframework.web.reactive.function.client.WebClient, який входить до складу модуля Spring WebFlux і є реактивною альтернативою традиційного RestTemplate для виконання HTTP-запитів у Spring-додатках. Використання WebClient дозволяє розробникам взаємодіяти з веб-сервісами за допомогою реактивного підходу, який дозволяє взаємодіяти з асинхронними та неблокуючими операціями.

З нових логів запуску програми видно, що тепер веб-сервером виступає Netty, який надає асинхронне середовище виконання на моделі “event driven design”, і є веб-сервером за замовчуванням у модулі Spring WebFlux (рис. 3.32).

```
2023-12-30T19:29:04.987+02:00  INFO 796 --- [administrative data service ASYNC] [           main] dministrativeDataServiceAsyncApplication : Starting
 AdministrativeDataServiceAsyncApplication using Java 21.0.1 with PID 796
 (C:\Users\Work\magister_app\administrative-data-service-async\administrative-data-service-async\target\classes started by Work in
 C:\Users\Work\magister_app\administrative-data-service-async\administrative-data-service-async)
2023-12-30T19:29:04.987+02:00  INFO 796 --- [administrative data service ASYNC] [           main] dministrativeDataServiceAsyncApplication : No active profile set, falling
 back to 1 default profile: "default"
2023-12-30T19:29:06.673+02:00  INFO 796 --- [administrative data service ASYNC] [           main] .s.d.r.c.RepositoryConfigurationDelegate : Multiple Spring Data modules
 found, entering strict repository configuration mode
2023-12-30T19:29:06.689+02:00  INFO 796 --- [administrative data service ASYNC] [           main] .s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring Data R2DBC
 repositories in DEFAULT mode.
2023-12-30T19:29:07.082+02:00  INFO 796 --- [administrative data service ASYNC] [           main] .s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repository
 scanning in 381 ms. Found 1 R2DBC repository interface.
2023-12-30T19:29:10.294+02:00  INFO 796 --- [administrative data service ASYNC] [           main] o.s.b.web.embedded.netty.NettyWebServer  : Netty started on port 8080
2023-12-30T19:29:10.332+02:00  INFO 796 --- [administrative data service ASYNC] [           main] dministrativeDataServiceAsyncApplication : Started
 AdministrativeDataServiceAsyncApplication in 6.346 seconds (process running for 7.762)
```

Рисунок 3.32 – Логи запуску веб-додатка на технології Spring WebFlux

3.3.5 Тестування додатку administrative-data-service працюючого на базі “event driven design” моделі

Виміряємо показники використання CPU, heap пам’яті та кількість програмних потоків без навантаження (рис. 3.33–3.35).

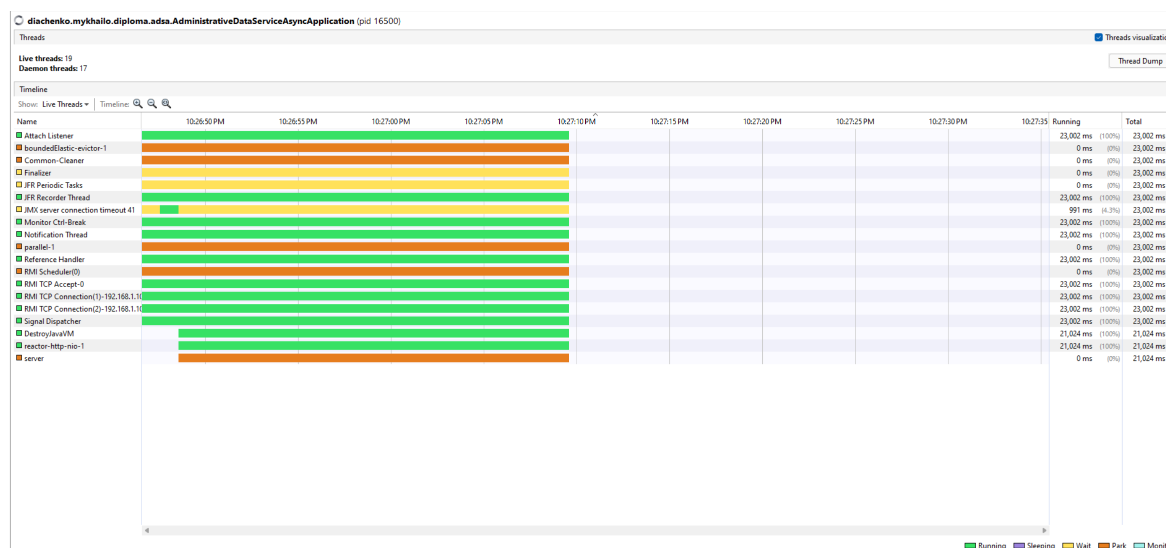


Рисунок 3.33 – Кількість програмних потоків без навантаження

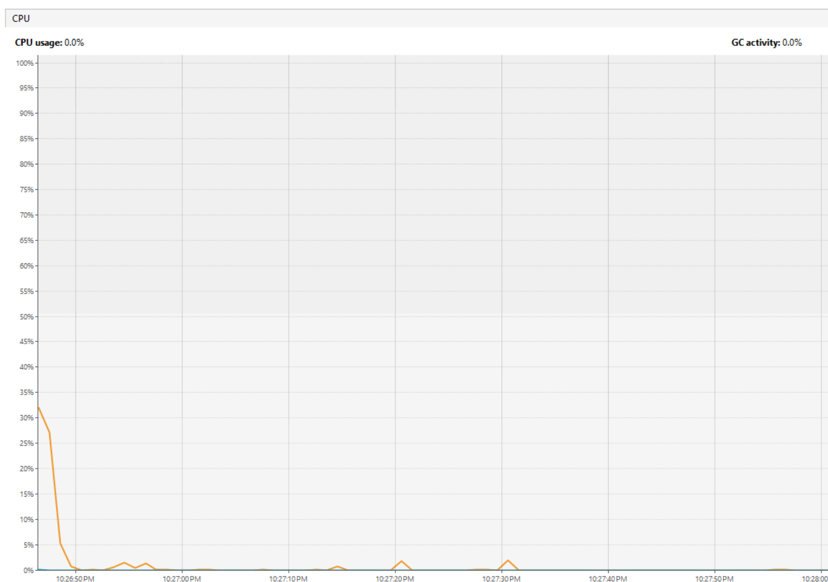


Рисунок 3.34 – Показники використання CPU без навантаження

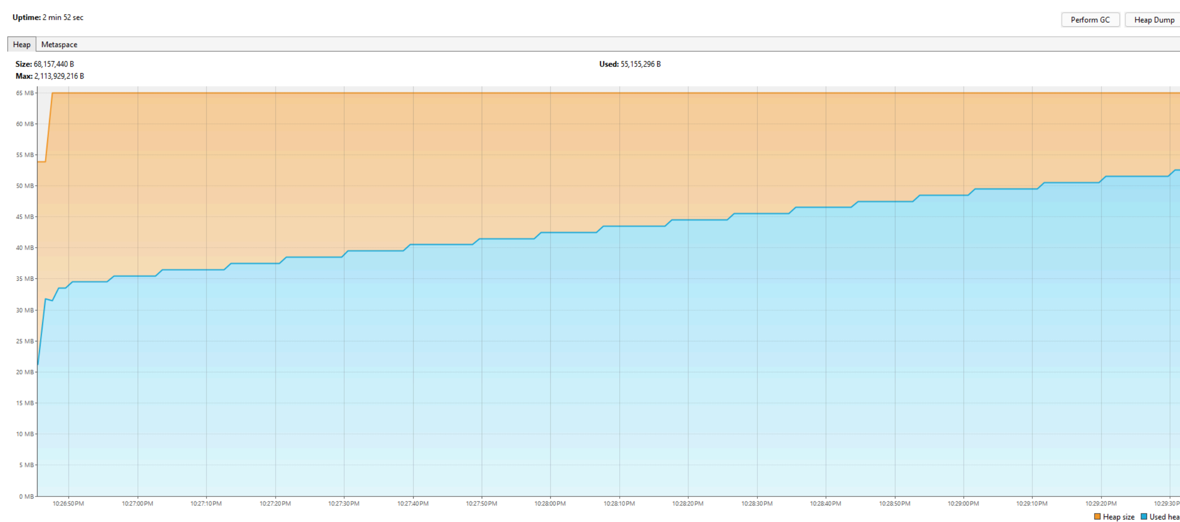


Рисунок 3.35 – Показники використання heap пам’яті без навантаження

Тест №1 – це 30 HTTP-запитів у секунду на сервер впродовж 30 секунд. Всього 900 запитів. Reactor модуль, який є серцем Spring WebFlux має декілька пулів потоків. db-pool – це окремий пул, який створено для задачі відправки запитів до бази даних. Це означає, що потоки зі стандартних пулів, які займаються обробкою HTTP-запитів від користувача, не блокуються під час очікування відповіді PostgreSQL, а продовжують свою роботу. В поточному тесті

навантаження не є достатньо вагомим для того, щоб завантажити усі 30 потоків з db-pool на весь час проведення тесту (помаранчевий колір на рис. 3.36).

Рисунок 3.36 – Стан програмних потоків під час виконання тесту №1

Рівень використання CPU тримався в межах 15–30% переважну більшість часу, і максимальний був 37% (рис. 3.37).

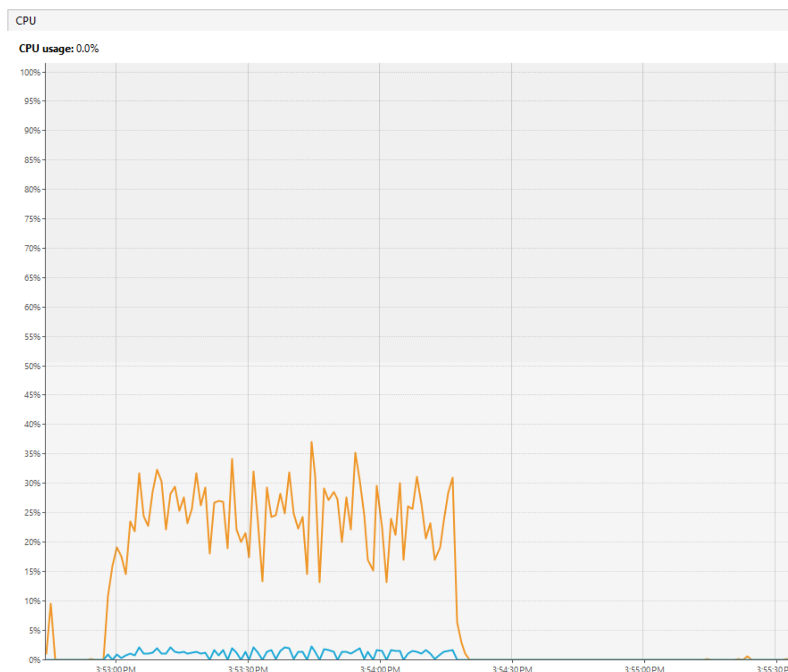


Рисунок 3.37 – Рівень використання CPU під час виконання тесту №1

Рівень пам’яті heap у використанні коливався у діапазоні 500–1000 мегабайт більшу частину часу обробки запитів (рис. 3.38).

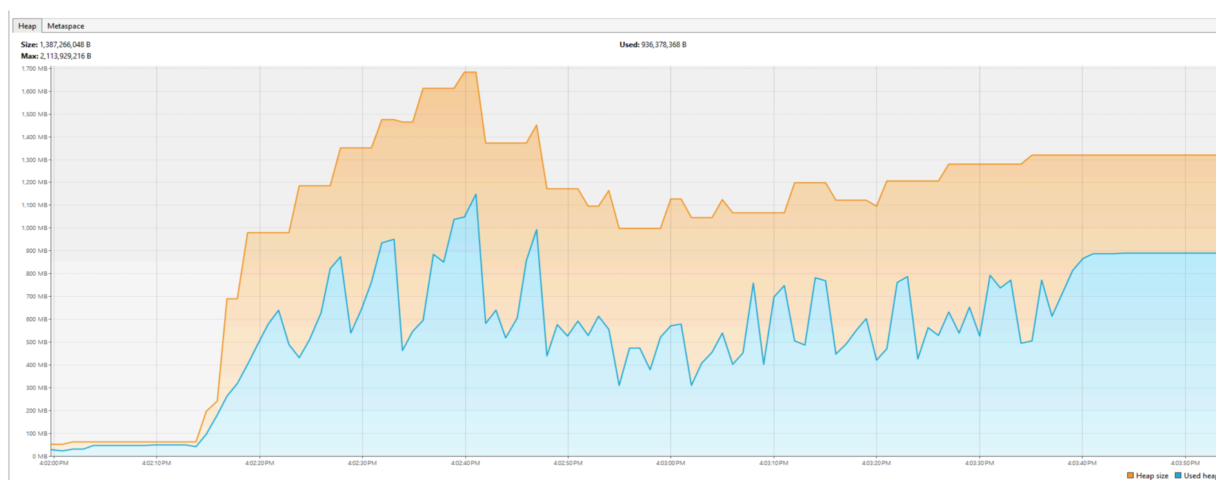


Рисунок 3.38 – Рівень пам’яті heap під час виконання тесту №1

Результати тесту №1 з Apache JMeter. Кількість запитів 900. Середня швидкість відповіді в мілісекундах 3181. Швидкість відповіді на 90% запитів менша ніж 4790 мілісекунд. Показник помилок 0%. Час виконання тесту 1 хв. 52 сек (рис. 3.39).

Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput
HTTP Request	900	3181	2847	4790	5585	8565	1042	10218	0.00%	8.0/sec
TOTAL	900	3181	2847	4790	5585	8565	1042	10218	0.00%	8.0/sec

Рисунок 3.39 – Результати з Apache JMeter, тест №1

Тест №2 – це 1000 одночасних запитів одноразово. З результатів профайлера бачимо, що кількість потоків залишилась незмінна, але db-pool став більш навантажений. У цей раз кожен потік зайнятий майже на 100% протягом усього часу проведення тесту (помаранчевий колір на рис. 3.40).

Рисунок 3.40 – Стан програмних потоків під час виконання тесту №2

Використання CPU залишилось без значних змін, адже потужності фізичного процесора і кількість ядер не змінилась (рис. 3.41).

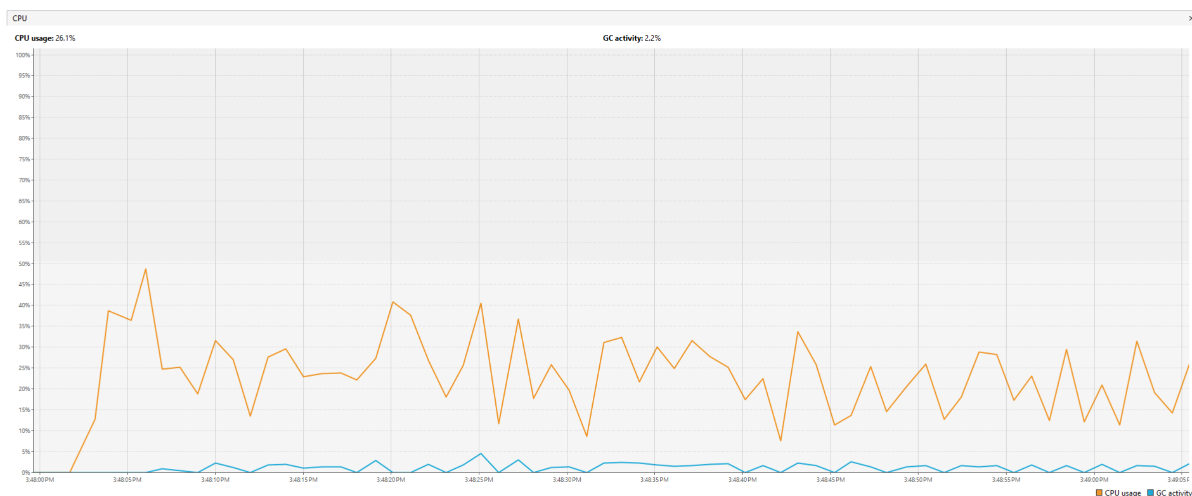


Рисунок 3.41 – Рівень використання CPU під час виконання тесту №2

Рівень пам’яті heap у використанні тримався в рамках 600–1100 мегабайт (рис. 3.42).

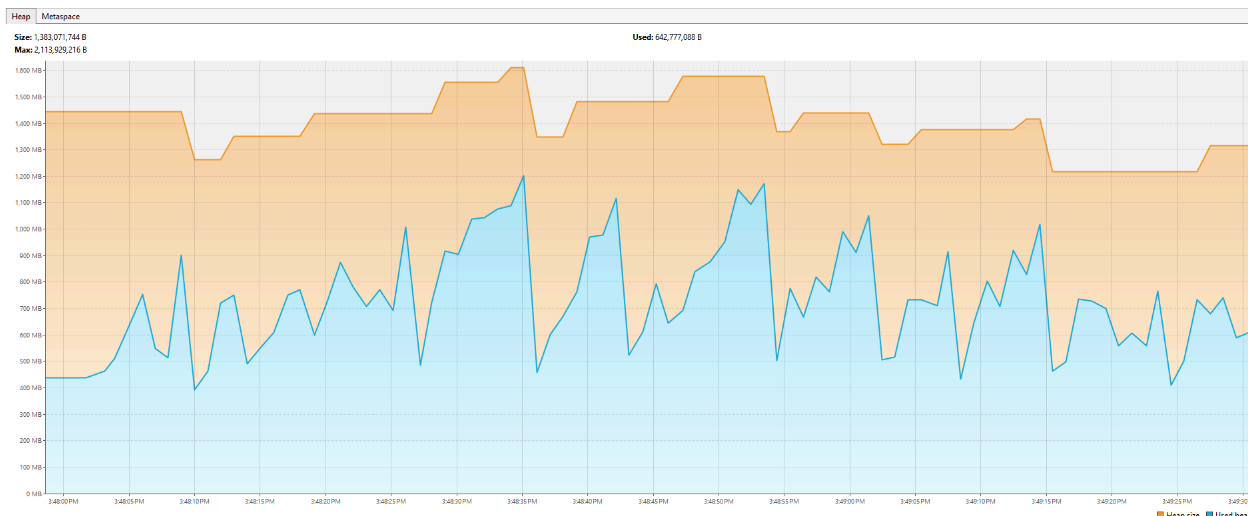


Рисунок 3.42 – Рівень пам’яті heap під час виконання тесту №2

Результати тесту №2 з Apache JMeter. Кількість запитів 1000. Середня швидкість відповіді в мілісекундах 48884. Швидкість відповіді на 90% запитів менша ніж 83847 мілісекунд. Показник помилок 0.00%. Час виконання тесту 1 хв 51 сек. Максимальний час очікування з'єднання з базою даних становить 20000 мілісекунд (рис. 3.43).

Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput
HTTP Request	1000	48884	49376	83847	88301	91470	3379	92165	0.00%	10.5/sec
TOTAL	1000	48884	49376	83847	88301	91470	3379	92165	0.00%	10.5/sec

Рисунок 3.43 – Результати з Apache JMeter, тест №2

3.4 Порівняння результатів тестів

Отримані результати тестів наведені в таблиці 3.1.

Таблиця 3.1 – порівняння результатів тестів

Модель реалізації програми	Назва тесту	Швидкість HTTP-відповідей на 90% запитів, мілісекунд	Помилки, %	Час обробки всіх HTTP-запитів, хв:сек	Використання CPU, %	Використання heap пам'яті, мегабайт	Кількість робочих програмних потоків
Spring WebMVC	Тест №1	5396	0	2:06	20-40	750-1480	30
Spring WebFlux		4790	0	1:52	15-30	500-1000	30
Spring WebMVC	Тест №2	86989	59.3	1:42	20-40	800 - 1500	200
Spring WebMVC (спроба 2)		131049	0	2:26	20-40	800 - 1500	200
Spring WebFlux		83847	0	1:51	20-40	600-1100	30

Для відсоткового порівняння використаємо формулу відсоткового зменшення (3.1).

$$\left(\frac{Nmax - Nmin}{Nmax}\right) \times 100 \qquad (3.1)$$

де $Nmax$ – це більше з двох чисел, $Nmin$ – меньше з двох чисел.

Тест №1:

Швидкість HTTP-відповідей на 90% запитів:

$$(\frac{5396 - 4790}{5396}) \times 100 = 11.23\%$$

на користь Spring WebFlux.

Помилки: зменшення відсотку помилок не обчислюється, оскільки обидві технології показали нульовий відсоток помилок.

Час обробки всіх HTTP-запитів:

$$(\frac{(2\times60+6)-(1\times60+52)}{2\times60+6}) \times 100 = 11.11\%$$

на користь Spring WebFlux.

Використання CPU:

$$(\frac{(\frac{40+20}{2})-(\frac{30+15}{2})}{(\frac{40+20}{2})}) \times 100 = 25\%$$

на користь Spring WebFlux.

Використання heap пам'яті:

$$(\frac{(\frac{800+1150}{2})-(\frac{600-1100}{2})}{(\frac{800+1150}{2})}) \times 100 = 12.82\%$$

на користь Spring WebFlux.

Кількість робочих програмних потоків: Зменшення відсотку кількості робочих програмних потоків не обчислюється, оскільки обидві технології показали однаковий результат.

Тест №2:

Швидкість HTTP-відповідей на 90% запитів:

$$(\frac{131049-86989}{131049}) \times 100 = 33.62\%$$

на користь Spring WebFlux.

Помилки:

$$(\frac{59.3 - 0}{59.3}) \times 100 = 100\%$$

на користь Spring WebFlux. З повторного результату Тесту №2 для Spring WebMVC видно, що відсоток помилок можна скоригувати конфігурацією бази

даних та іншим, але застосування, наприклад, нелімітованого часу очікування з'єднання за базою даних не є хорошою практикою і може ще більше погіршити продуктивність програми. Тому беремо до розрахунків результати першої спроби тесту №2, тобто 59.3%.

Час обробки всіх HTTP-запитів:

$$(\frac{(2\times60+26)-(1\times60+42)}{2\times60+26})\times 100 = 30.1\%$$

на користь Spring WebFlux.

Використання CPU: Зменшення відсотку використання CPU не обчислюється, оскільки обидві технології показали однаковий результат.

Використання heap пам'яті:

$$(\frac{(\frac{1480+750}{2})-(\frac{1000+500}{2})}{(\frac{1480+750}{2})})\times 100 = 32.7\%$$

на користь Spring WebFlux.

Кількість робочих програмних потоків:

$$(\frac{200-30}{200})\times 100 = 85\%$$

на користь Spring WebFlux.

ВИСНОВКИ

Дослідивши стан ринку Java веб-додатків стосовно їх реалізації та відповідності сучасним вимогам, а саме швидкості обробки HTTP-запитів від користувачів в умовах високого трафіку з одного боку, та надання багатої функціональності з іншого, що може включати інтеграції зі сторонніми веб-додатками, можна зробити наступний висновок: більшість з таких додатків можуть досягти відповідного рівня продуктивності тільки за рахунок масштабування систем (додавання обчислювальних потужностей та кількості обчислювальних машин), що збільшує складність системи та ціну її утримання.

З проведеного аналізу видно, що однією з основних причин є модель обробки HTTP-запитів “thread-per-request”, яка була та залишається основним вибором розробників. Порівнявши результати тестів, навантаження такої системи з іншою, реалізованою на базі моделі обробки HTTP-запитів “event-driven”, можемо зробити наступні висновки: При відносно не великому навантаженні (30 запитів в секунду) результати відрізняються на значущі, і навіть тут “event-driven” модель показує себе краще на 11-12%. При навантаженні 1000 запитів в секунду, “event-driven” модель показує значно кращі результати в швидкості, споживанні ресурсів та відмовостійкості на приблизно 30%, а по деяких критеріям до 60% .

Таким чином, зміна підходу до реалізації веб-додатків разом з можливістю масштабування систем, показують більші перспективи для застосування в сучасних високонавантажених системах.

ПЕРЕЛІК ПОСИЛАНЬ

1. “Асинхронність в програмуванні” [https://habr.com/ru/companies/jugru/articles/446562/].
2. “The performance challenge in Java microservices” [https://oskar-uit-de-bos.medium.com/the-performance-challenge-in-java-microservices-e51cce3977e9].
3. “Performance and scalability analysis of Java IO and NIO based server models, their implementation and comparison” [https://s3-eu-central-1.amazonaws.com/ucu.edu.ua/wp-content/uploads/sites/8/2019/12/Petro-Karabyn.pdf].
4. “Java NIO vs. IO” [https://jenkov.com/tutorials/java-nio/nio-vs-io.html].
5. “Difference between Java IO and Java NIO” [https://www.geeksforgeeks.org/difference-between-java-io-and-java-nio/].
6. “TIOBE Index for December 2023” [https://www.tiobe.com/tiobe-index/].
7. “Server-side programming languages market position report” [https://w3techs.com/technologies/market/programming_language].
8. “What is a Servlet Container?” [https://dzone.com/articles/what-servlet-container].
9. “The State of Microservices” [https://www.redhat.com/en/blog/state-microservices]
10. “Akamai Online Retail Performance Report: Milliseconds Are Critical” [https://www.akamai.com/newsroom/press-release/akamai-releases-spring-2017-state-of-online-retail-performance-report]

Додаток А

Програмний код класу diachenko.mykhailo.diploma.TaxCSVGenerator

```
package diachenko.mykhailo.diploma;

import com.opencsv.CSVWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.time.Instant;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;
import java.util.Random;

public class TaxCSVGenerator {

    private static final List<String> PERSONS = List.of(
            "Пономаренко Наташа Олексіївна",
            "Мельниченко Віра Михайлівна",
            "Гнатюк Анна Олексіївна",
            "Крамарчук Валерій Васильович",
            "Васильчук Олена Валентинівна",
            "Лисенко Володимир Федорович",
            "Романченко В'ячеслав Анатолійович",
            "Дьяченко Михайло Костянтинович");

    private static final List<String> TAX_NAMES = List.of(
```

```
            "Податок з доходу фізичних осіб",
            "Податок з продажу рухомого майна",
            "Воєнний збір",
            "Єдиний соціальний внесок"
    );

    private static final DateFormat formatter = new SimpleDateFormat("yyyy-MM-dd
HH:mm:ss");

    public static void main (String[] args) throws IOException {
        List<String[]> csvData = createCsvDataSimple();
        // default all fields are enclosed in double quotes
        // default separator is a comma
        try (CSVWriter writer = new CSVWriter(new
FileWriter("C:\\Users\\Work\\magister_app\\administrative-data-service\\administrative-
data-service\\tax.csv"))) {
            writer.writeAll(csvData);
        }
    }

    private static List<String[]> createCsvDataSimple () {
        Random rand = new Random();
        List<String[]> list = new ArrayList<>();
        String[] header = {"id", "type", "name", "person_name", "payment_amount",
"create_time"};
        list.add(header);
        for (int i = 1; i < 1000000; i++) {
            String[] record = {
                    String.valueOf(i),
```

```
                "Податок",
                TAX_NAMES.get(rand.nextInt(TAX_NAMES.size())),
                PERSONS.get(rand.nextInt(PERSONS.size())),
                String.format("%.2f", rand.nextFloat(1f, 3000f)),
                getRandomDate(rand)
            };
            list.add(record);
        }
        return list;
    }

    private static String getRandomDate (Random rand) {
        long day = 86400;
        long now = Instant.now().getEpochSecond();
        long minDateMills = new Date((now - day * 365 * 5) * 1000).getTime();
        long maxDateMills = now * 1000;
        long randomDateMills = rand.nextLong(minDateMills, maxDateMills);
        return formatter.format(new Date(randomDateMills));
    }
}
```

Додаток Б

Програмний код класу

diachenko.mykhailo.diploma.ubs.service.impl.UBServiceImpl

```
package diachenko.mykhailo.diploma.ubs.service.impl;

import diachenko.mykhailo.diploma.ubs.dto.UBDto;
import diachenko.mykhailo.diploma.ubs.service.UBService;
import org.springframework.stereotype.Service;

import java.math.BigDecimal;
import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.time.Instant;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;
import java.util.Random;

@Service
public class UBServiceImpl implements UBService {

    private final DateFormat formatter = new SimpleDateFormat("yyyy-MM-dd
HH:mm:ss");

    private static final List<String> BILL_NAMES = List.of(
            "Рахунок за постачання газу",
            "Рахунок за постачання електроенергії",
            "Рахунок за вивіз сміття",
```

```
        "Рахунок за посточання води"
);

@Override
public List<UBDto> getUtilityBillsFines (String personName) {
    Random random = new Random();
    int loopSize = random.nextInt(1, 20);
    List<UBDto> utilityBills = new ArrayList<>();
    for (int i = 0; i < loopSize; i++) {
        UBDto bill = new UBDto();
        bill.setId((long) i);
        bill.setName(BILL_NAMES.get(random.nextInt(BILL_NAMES.size())));
        bill.setPersonName(personName);
        bill.setType("Комунальний платіж");
        bill.setCreateTime(getRandomDate(random));
        bill.setPaymentAmount(BigDecimal.valueOf(random.nextDouble(10, 1000)));
        utilityBills.add(bill);
    }
    return utilityBills;
}

private String getRandomDate (Random rand) {
    long day = 86400;
    long now = Instant.now().getEpochSecond();
    long minDateMills = (now - day * 365 * 5) * 1000;
    long maxDateMills = now * 1000;
    long randomDateMills = rand.nextLong(minDateMills, maxDateMills);
    return formatter.format(new Date(randomDateMills));
}}
```

Додаток В

Зміст файлу pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>3.2.1</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>diachenko.mykhailo</groupId>
    <artifactId>ads</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>Administrative data service</name>
    <description>Service which returns person-related administrative data</description>
    <properties>
        <java.version>21</java.version>
        <mapstruct.version>1.5.5.Final</mapstruct.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
        </dependency>
```

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
</dependency>
<dependency>
    <groupId>org.liquibase</groupId>
    <artifactId>liquibase-core</artifactId>
</dependency>
<dependency>
    <groupId>org.mapstruct</groupId>
    <artifactId>mapstruct</artifactId>
    <version>${mapstruct.version}</version>
</dependency>
<dependency>
    <groupId>org.mapstruct</groupId>
    <artifactId>mapstruct-processor</artifactId>
    <version>${mapstruct.version}</version>
</dependency>
<dependency>
    <groupId>diachenko.mykhailo</groupId>
    <artifactId>ads-utility-bills-service</artifactId>
```

```xml
            <version>0.0.1-SNAPSHOT</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.11.0</version>
                <configuration>
                    <annotationProcessorPaths>
                        <path>
                            <groupId>org.projectlombok</groupId>
                            <artifactId>lombok</artifactId>
                            <version>1.18.30</version>
                        </path>
                        <path>
                            <groupId>org.mapstruct</groupId>
                            <artifactId>mapstruct-processor</artifactId>
```

```
                        <version>${mapstruct.version}</version>
                    </path>
                </annotationProcessorPaths>
            </configuration>
        </plugin>
    </plugins>
</build>
</project>
```

Додаток Г

Програмний код класу

diachenko.mykhailo.diploma.ads.service.impl.ADSServiceImpl

```
package diachenko.mykhailo.diploma.ads.service.impl;

import diachenko.mykhailo.diploma.ads.client.UtilityBillsClient;
import diachenko.mykhailo.diploma.ads.dto.NotificationDTO;
import diachenko.mykhailo.diploma.ads.mapper.NotificationMapper;
import diachenko.mykhailo.diploma.ads.model.Tax;
import diachenko.mykhailo.diploma.ads.repository.postgres.TaxRepository;
import diachenko.mykhailo.diploma.ads.service.ADSService;
import diachenko.mykhailo.diploma.ubs.dto.PersonRequest;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.stereotype.Service;
import org.springframework.util.StopWatch;

import java.util.ArrayList;
import java.util.List;

@Slf4j
@Service
@RequiredArgsConstructor
public class ADSServiceImpl implements ADSService {

    private final TaxRepository taxRepository;
    private final UtilityBillsClient utilityBillsClient;
    private final NotificationMapper mapper;
```

```
@Override
public List<NotificationDTO> getNotifications (PersonRequest person) {
    List<NotificationDTO> taxes = getTaxes(person);
    log.info("Taxes obtained. Size={}", taxes.size());
    List<NotificationDTO> utilityBillsFines = getUtilityBillsFines(person);
    log.info("Utility bills fines obtained. Size={}", utilityBillsFines.size());
    ArrayList<NotificationDTO> resultList = new ArrayList<>();
    resultList.addAll(taxes);
    resultList.addAll(utilityBillsFines);
    return resultList;
}

private List<NotificationDTO> getUtilityBillsFines (PersonRequest person) {
    try {
        return utilityBillsClient.getUtilityBills(person)
                .stream()
                .map(mapper::toNotification)
                .toList();
    } catch (Exception e) {
        log.error("unable to get utilityBillsFines due to {}", e.getMessage(), e);
        throw e;
    }
}

private List<NotificationDTO> getTaxes (PersonRequest person) {
    try {
        StopWatch stopWatch = new StopWatch();
        stopWatch.start();
```

```
            List<Tax> taxes =
taxRepository.findAllByPersonName(person.getPersonName());
            stopWatch.stop();
            log.info("Call for taxes took {}ms", stopWatch.getTotalTimeMillis());
            return taxes
                    .stream()
                    .map(mapper::toNotification)
                    .toList();
        } catch (Exception e) {
            log.error("unable to get taxes due to {}", e.getMessage(), e);
            throw e;
        }
    }
}
```

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

«ДОСЛІДЖЕННЯ ПРОДУКТИВНОСТІ ВИСОКОНАВАНТАЖЕНИХ HTTP-СЕРВЕРІВ, ПРАЦЮЮЧИХ НА БАЗІ АСИНХРОННОГО СЕРЕДОВИЩА ВИКОНАННЯ ТА РОЗРОБЛЕНИХ З ВИКОРИСТАННЯМ РЕАКТИВНОЇ МОДЕЛІ ПРОГРАМУВАННЯ НА МОВІ JAVA»

Виконав: студент групи КНДМ-63
Дьяченко Михайло Костянтинович
Керівник: доктор філософії
Березовська Юлія Володимирівна

ЗАГАЛЬНА ХАРАКТЕРИСТИКА КВАЛІФІКАЦІЙНОЇ РОБОТИ

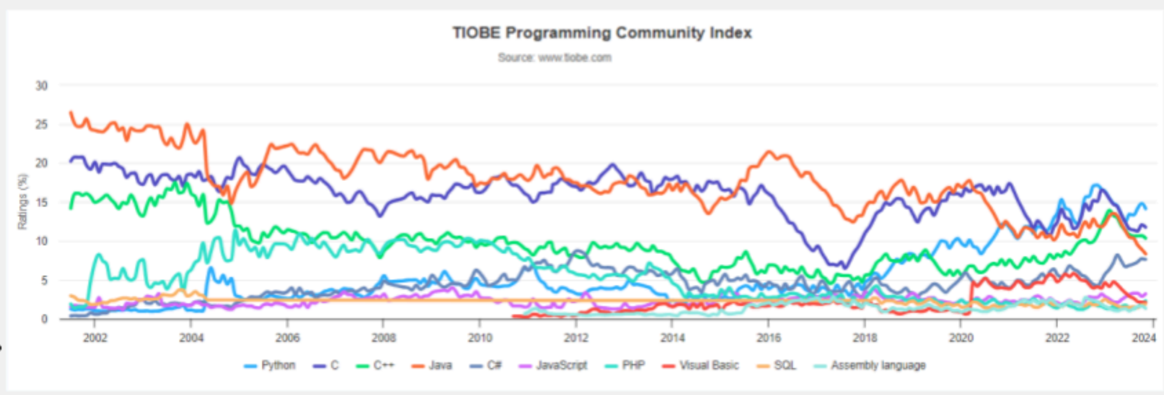
Мета дослідження	підвищення продуктивності та відмовостійкості високонавантажених веб-додатків шляхом використання асинхронного підходу обробки HTTP-запитів.
Об'єкт дослідження:	процес обробки HTTP-запитів у високонавантажених веб-додатках.
Предмет дослідження:	модель обробки HTTP-запитів у високонавантажених веб-додатках.

АКТУАЛЬНІСТЬ ТЕМИ

- Статистика та тренди вказують на те, що швидкість обробки інформації є критично фактором для забезпечення успіху веб-додатків.
- Інтеграція зовнішніх веб-ресурсів є не менш важливої для покращення функціональності веб-додатка
- Мікросервісна архітектура веб-додатків набуває популярності, що призводить до більшої кількості взаємодій між ними.
- У світлі зростаючої складності веб-систем та вимог до миттєвої відповіді, синхронна модель обробки HTTP-запитів на серверній стороні виявляється недостатньою для забезпечення високої продуктивності

ПОПУЛЯРНІСТЬ МОВИ JAVA

Згідно джерелу статистичної інформації TIOBE [6], мова програмування JAVA була і залишається лідером на всесвітньому ринку протягом останніх двадцяти років як зображено на рисунку



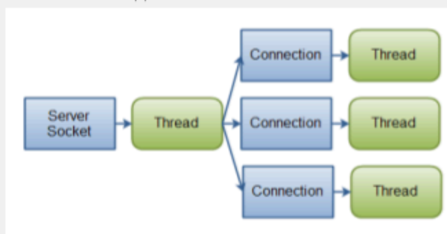
ПОПУЛЯРНІСТЬ МОВИ JAVA В ЗАЛЕЖНОСТІ ВІД НАВАНТАЖЕНОСТІ СИСТЕМИ

Опитування проведене w3techs [7], показує позицію мови програмування Java на ринку з точки зору популярності та трафіку порівняно з найпопулярнішими серверними мовами програмування . Технологія в нижньому правому куті використовується багатьма сайтами, але переважно сайтами із середнім рейтингом трафіку. Технологія у верхньому лівому куті використовується меншою кількістю сайтів, але переважно сайтами з високим трафіком.

ВАРІАНТИ РОБОТИ З ПОТОКАМИ ВВОДУ/ВИВОДУ ДАНИХ НА МОВІ JAVA

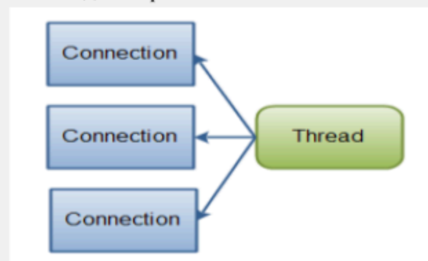
JAVA I/O

- Потоки Java I/O блокуються до закінчення операції.
- Для підтримання паралельних операцій необхідно створення нових Java потоків.
- Використовується у веб-серверах написаних з традиційною моделлю обробки HTTP-запитів “thread per request”.
- Обробка кожного запиту ізольована від інших, порядок виконання коду зазвичай є послідовним і не залежить від кількості клієнтів



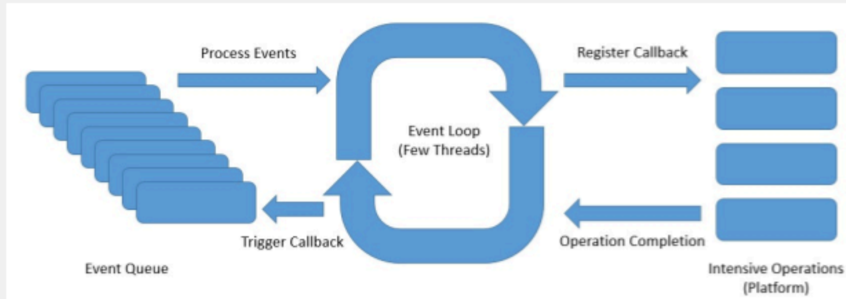
JAVA NIO

- Потоки Java NIO не блокуються на весь час операції.
- Запитує дані з каналу та отримує лише те, що доступно, або нічого.
- Java потік може виконувати інші операції доки очікує дані.
- Лежить в основі веб рішень працюючих на моделі обробки HTTP-запитів “event driven”.

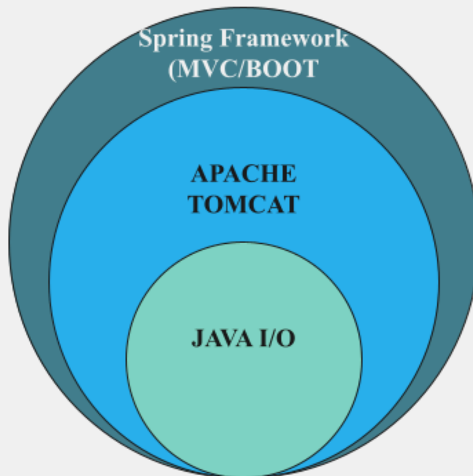


Модель “event driven” з циклом подій

- Сигнал, який створює компонент після досягнення заданого стану є подією (event)
- Цикл подій (event loop) забезпечує координацію одного програмного потоку з декількома I/O з'єднаннями, різко зменшуючи кількість потоків, необхідних для роботи системи
- Цикл подій обробляє події з черги подій послідовно та повертається одразу після реєстрації обробника такої події
- Система може ініціювати події завершення операцій, такі як виклик бази даних або виклик стороннього сервісу.
- Цикл подій може визвати зареєстрований обробник такої події та повернути результат клієнту.



ТЕХНОЛОГІЇ ДЛЯ РОЗРОБКИ ВЕБ-ДОДАТКА НА СИНХРОННІЙ МОДЕЛІ



SPRING FRAMEWORK

програмний каркас (фреймворк) з відкритим кодом та контейнери з підтримкою інверсії керування для платформи Java.
SPRING MVC модуль для веб розробки.
SPRING BOOT модуль для швидкої розробки з різними авто конфігураціями

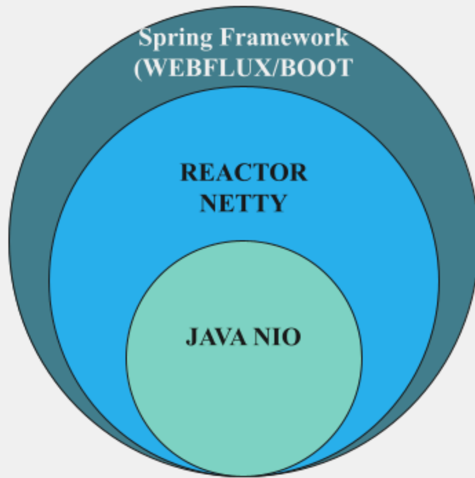
APACHE TOMCAT

Веб-сервер та контайнер сервлетів Java, працюючих на моделі “thread per request”

JAVA I/O

Стандартний програмний інтерфейс JAVA
Пакет java.io містить усі класи, необхідні для операцій вводу та виводу в синхронному (блокуючому) стилі

ТЕХНОЛОГІЇ ДЛЯ РОЗРОБКИ ВЕБ-ДОДАТКА НА АСИНХРОННІЙ МОДЕЛІ



SPRING FRAMEWORK

програмний каркас (фреймворк) з відкритим кодом та контейнери з підтримкою інверсії керування для платформи Java.
SPRING WEBFLUX модуль для веб розробки асинхронних веб-додатків.
SPRING BOOT модуль для швидкої розробки з різними авто конфігураціями

REACTOR NETTY

Асинхронний побудований на моделі “event driven” програмний фреймворк для роботи з мережею

JAVA NIO

Стандартний програмний інтерфейс JAVA
Містить усі класи, необхідні для операцій вводу та виводу в асинхронному (неблокуючому) стілі

ІНШІ ВИКОРИСТАНІ ТЕХНОЛОГІЇ

Java - високорівнева, кроссплатформенна, об’єктно-орієнтована мова програмування.
Apache Maven - інструмент для управління проектами та залежностями в області розробки програмного забезпечення.
PostgreSQL - об'єктно-реляційна система керування базами даних (СКБД), яка відповідає стандартам SQL та визначається як "вільне та відкрите програмне забезпечення"
Docker - платформа для розробки, доставки та експлуатації програмного забезпечення в контейнерах.
Liquibase - вільний та відкритий інструмент для керування версіями схеми бази даних

Intelij Idea - інтегроване середовище розробки призначене для розробки програмного забезпечення.
VisualVM - інструмент для профілювання та аналізу. продуктивності Java додатків
Apache JMEter - інструмент для тестування продуктивності та навантаження веб-додатків.

Liquibase

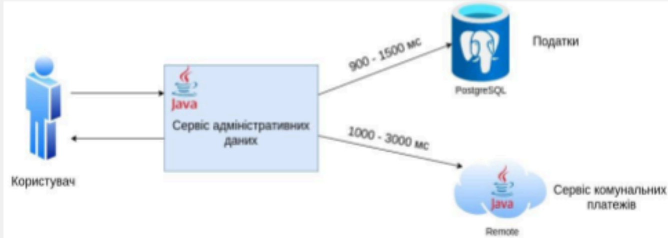
Maven™

VisualVM

APACHE JMeter™

Функціональність та високорівнева архітектура

Administrative Data Service (ADS) або сервіс надання адміністративних даних, може надавати дані про податки та комунальним платежі платника. ADS реалізований в стилі REST. Дані завантажуються з двох сторонніх систем, PostgreSQL і Utility Bills Service

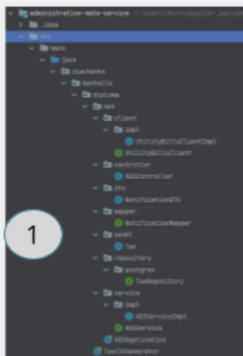


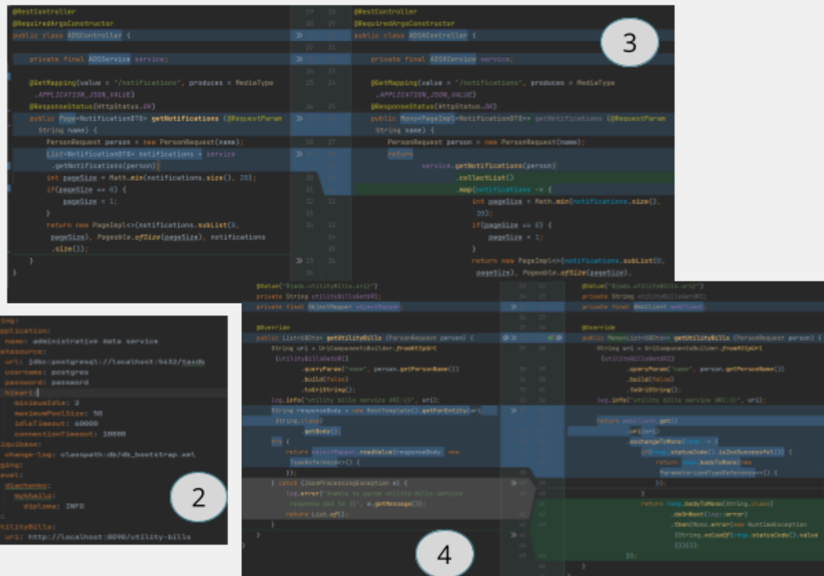
Utility Bills Service (UBS) або сервіс комунальних платежів, зберігає дані про комунальні платежі платника в пам'яті програми та формує HTTP відповіді в форматі JSON. HTTP-запит обробляється за час між 1 та 3 секундами.

PostgreSQL реляційна база даних, зберігає дані про податки платників у таблиці tax. Таблиця заповнена тестовими даними в кількості 1 мільйон рядків. Запит в базу даних через JDBC дарйвер займає між 0.9 - 1.5 секунди, в залежності від кількості знайдених результатів.


```
[
    {
        "id": 1,
        "type": "Комунальний платіж",
        "name": "Рахунок за вивіз сміття",
        "personName": "Дьяченко Михайло Костянтинович",
        "paymentAmount": 600.0645780212185,
        "createTime": "2022-09-28 23:11:32"
    },
    {
        "id": 2,
        "type": "Комунальний платіж",
        "name": "Рахунок за постачання газу",
        "personName": "Дьяченко Михайло Костянтинович",
        "paymentAmount": 191.3683879285978,
        "createTime": "2019-08-12 10:52:35"
    },
    {
        "id": 3,
        "type": "Комунальний платіж",
        "name": "Рахунок за постачання електроенергії",
        "personName": "Дьяченко Михайло Костянтинович",
        "paymentAmount": 364.8637088809509,
        "createTime": "2020-07-18 22:18:48"
    },
```


Структура файлів проекту





1. Структура файлів проекту.
2. Файл конфігурації application.yaml.
3. Класс контроллер.
4. Класс клієнт сервісу UBS.

Зведена таблиця результатів тестування

Технологія реалізації програми	**Назва тесту**	**Швидкість HTTP-відповідей на 90% запитів, мс**	**Помилки, %**	**Час обробки всіх HTTP-запитів, хв:сек**	**Використання CPU, %**	**Використання heap пам'яті, мегабайт**	**Кількість робочих програмних потоків**
Spring WebMVC	Тест №1 30 HTTP-запитів у секунду на сервер впродовж 30 секунд. Всього 900 запитів	5396	0	2:06	20-40	750-1480	30
Spring WebFlux		4790	0	1:52	15-30	500-1000	30
Spring WebMVC	Тест №2 1000 одночасних запитів одноразово	86989	59.3	1:42	20-40	800 - 1500	200
		131049	0	2:26	20-40	800 - 1500	200
Spring WebFlux		83847	0	1:51	20-40	600-1100	30

РОЗРАХУНОК ВІДСОТКОВОГО ЗМЕНШЕННЯ ПО КОЖНОМУ ПОКАЗНИКУ ОТРИМАНИХ З РЕЗУЛЬТАТУ ТЕСТУ №2

Швидкість HTTP-відповідей на 90% запитів:
(131049 – 86989) / 131049 * 100 = 33.62% на користь Spring WebFlux.
Помилки: (59.3 – 0) / 59.3 * 100 = 100% на користь Spring WebFlux. З повторного результату Тесту №2 для Spring WebMVC видно, що відсоток помилок можна скоригувати конфігурацією бази даних та іншим, але застосування, наприклад, нелімітованого часу очікування з'єднання за базою даних не є хорошою практикою і може ще більше погіршити продуктивність програми. Тому беремо до розрахунків результати першої спроби тесту №2, тобто 59.3%.
Час обробки всіх HTTP-запитів: ((2×60+26)−(1×60+42))/(2×60+26)*100=30.1% на користь Spring WebFlux.
Використання CPU: Зменшення відсотку використання CPU не обчислюється, оскільки обидві технології показали однаковий результат.
Використання heap пам'яті:
((1480 + 750) / 2 − (1000 + 500) /2) / (1480 + 750) / 2 * 100 = 32.7% на користь Spring WebFlux.
Кількість робочих програмних потоків: (200 – 30) / 200 * 100 = 85% на користь Spring WebFlux.

Висновки

Дослідивши стан ринку Java веб-додатків стосовно їх реалізації та відповідності сучасним вимогам, а саме швидкості обробки HTTP-запитів від користувачів в умовах високого трафіку з одного боку, та надання багатої функціональності з іншого, що може включати інтеграції зі сторонніми веб-додатками, можна зробити наступний висновок:

- більшість з таких додатків можуть досягти відповідного рівня продуктивності тільки за рахунок масштабування систем (додавання обчислювальних потужностей та кількості обчислювальних машин), що збільшує складність системи та ціну її утримання.
- з проведеного аналізу видно, що однією з основних причин є синхронна модель обробки HTTP-запитів, яка була та залишається основним вибором розробників.
- при відносно невеликому навантаженні (30 запитів в секунду) результати відрізняються не критично, і навіть тут асинхронна модель показує себе краще на 11-12%.
- при навантаженні 1000 запитів в секунду, “event-driven” модель показує значно кращі результати в швидкості, споживанні ресурсів та відмовостійкості на приблизно 30%, а по деяких критеріям до 60% .
- зміна підходу до реалізації веб-додатків разом з можливістю масштабування систем, показують більші перспективи для застосування в сучасних високонавантажених системах.