НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
Інститут прикладного системного аналізу
Кафедра математичних методів системного аналізу

До захисту допущено:
Завідувача кафедри
_____Оксана ТИМОЩУК
«___»_____20__ р.

Дипломна робота
на здобуття ступеня бакалавра
за освітньо-професійною програмою «Системи та методи штучного інтелекту»
спеціальності 122 «Комп'ютерні науки»
на тему: «Створення програмного забезпечення для проведення аналізу ефективності веб-серверів на базі віртуальної машини Java»

Виконав:
студент IV курсу, групи КА-75
Савенко Ілля Михайлович _____

Керівник:
асистент кафедри ММСА
Насиров Дмитро Євгенович _____

Консультант з економічного розділу:
Доцент кафедри теоретичної і прикладної економіки ФММ
Рощина Надія Василівна _____

Консультант з нормоконтролю:
Доцент к.т.н.
Коваленко Анатолій Єпіфанович _____

Рецензент:
Замісник директора ТОВ "ФАРОС ПРОДАКШН"
Баранець Анастасія Вікторівна _____

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

Київ – 2021 року

Інститут прикладного системного аналізу

Кафедра математичних методів системного аналізу

Рівень вищої освіти – перший (бакалаврський)

Спеціальність – 122 «Комп'ютерні науки»

Освітньо-професійна програма «Системи та методи штучного інтелекту»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Оксана ТИМОЩУК

«26» травня 2021 р.

ЗАВДАННЯ

на дипломну роботу студенту

Савенку Іллі Михайловичу

1. Тема роботи «Створення програмного забезпечення для проведення аналізу ефективності веб-серверів на базі віртуальної машини Java», керівник роботи Насиров Дмитро Євгенович, асистент кафедри ММСА, затверджені наказом по університету № 1344-с від 26.05.2021 р.
2. Термін подання студентом роботи 30.05.21
3. Вихідні дані до роботи: метрики аналітики серверного програмного забезпечення.
4. Зміст роботи: дослідження предметної середи, специфіки веб-серверів та інструментів дослідження на можливість тестування, розробка програмного продукту.
5. Перелік ілюстративного матеріалу архітектура серверного тестувального ПЗ, коефіцієнт порівняння протестованих систем, метрики та результати, висновки.

6. Консультанти розділів роботи[1*]

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

7. Дата видачі завдання 27 березня 2021

Календарний план

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітка
1.	Вивчення літератури за темою	15.04.2021	
2.	Підготовка першого розділу	21.04.2021	
3.	Підготовка другого розділу	30.04.2021	
4.	Розробка веб додатку	20.04.2021	
5.	Підготовка третього розділу	30.04.2021	
6.	Підготовка економічної частини	15.05.2021	
7.	Оформлення дипломної роботи	30.05.2021	
8.	Підготовка презентації доповіді	7.06.2021	
9.	Оформлення розділів відповідно до нормоконтролю		

Студент _____ Савенко Ілля

Керівник_____ Насиров Дмитро

1

РЕФЕРАТ

Дипломна робота містить: 103 с., 6 табл., 53 рис., 2 дод., 11 джер.

СТВОРЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ ПРОВЕДЕННЯ АНАЛІЗУ ЕФЕКТИВНОСТІ ВЕБ-СЕРВЕРІВ НА БАЗІ ВІРТУАЛЬНОЇ МАШИНИ JAVA

В якості методів тестування було обрано методи тестування продуктивності, стресс-тестування, та тестування навантаженням програмного продукту, як такі, що реалізують оптимальні методи для аналізу серверної архітектури.

Скрипти аналізу написано за допомогою такої технології як Gatling, для зняття метрик аналізу.

В цій роботі мені також довелося написати серверне ПЗ, яке імітує роботу бізнес-логіки, звернення до БД. Для отримання та аналізу метрик було використано інтерактивний режим механізму тестування. Для розгортання серверних технологій було використано апаратну архітектуру EC2 від AWS з архітектурою потужностей m5.2xlarge з 8ма ядрами процессора, 32гб оперативної пам’яті та операційною системою Debian 10.

Були розглянуті декілька технологій серверного ПЗ такі як Tomcat, Jetty, Wildfly й інші. Серверне ПЗ Jetty та Quarkus показали найкращі результати на час відгуку запитів. Tomcat показав найкращу роботи при стрес-тестуванні. Розвивати роботу можна у багатьох напрямках. Перший – розгорнути на іншій апартній архітектурі з меншими або більшими потужностями. Другий – запускати серверне ПЗ на інших операційних системах. Третій – запускати серверне ПЗ на інших Java-подібних віртуальних машинах. Четвертий – додати можливість масштабування ПЗ.

ABSTRACT

Thesis:103 p., 6 tabl., 53 fig., 2 adds., 11 references.

DEVELOPMENT OF SOFTWARE FOR ANALYSIS OF THE EFFICIENCY OF WEB SERVERS BASED ON THE VIRTUAL MACHINE JAVA

Methods of performance testing, stress testing, and software product load testing were selected as testing methods as those that implement optimal methods for server architecture analysis.

Analysis scripts are written using technology such as Gatling, to get analysis metrics.

In this work, I also had to write server software that simulates the work of business logic, access to the database. An interactive mode of the testing mechanism was used to obtain and analyze the metrics. AWS EC2 hardware architecture with m5.2xlarge power architecture with 8 processor cores, 32GB of RAM and Debian 10 operating system was used to deploy the server technologies.

Several server software technologies such as Tomcat, Jetty, Wildfly and others have been considered. Jetty and Quarkus server software showed the best results at the time of response. Tomcat showed the best performance in stress testing. You can develop your work in many ways. The first is to deploy to another hardware architecture with less or more capacity. The second is to run the server software on other operating systems. The third is to run the server software on other Java-like virtual machines. Fourth - add the ability to scale the software.

ЗМІСТ

ВСТУП

В наш час майже вся життєдіяльність людини пов’язана з IT-технологіями. Все більше людей в наш час починає користуватися онлайн сервісами, а вони ж в той час заміщують десктопні додатки. Онлайн-сервіси в наш час використовуються від простих підрахунків шкільної математики до складних систем банківських установ.

Розвиток людства в області бізнесу надання послуг спонукає ставити складніші задачі з системами, що націлені на їх обслуговування, для покращення добробуту звичайного користувача, і разом з тим підвищення доходу в цій сфері. Конкурентоспроможність – це один з ключових навичок в наш час, що просто перетворився у базову необхідність. Для обслуговуючих систем можна вивести декілька основних факторів, що наразі грають найбільшу роль: кількісна характеристика комп’ютерних ресурсів, необхідна для оптимальної роботи системи на апаратному рівні та результат інженерного підходу проектування системи, що визначає оптимальність її роботи на рівні програмного забезпечення. Ці два фактори визначають більш очевидні для користувача характеристики: стійкість системи до навантаження, швидкість відгуку, базову працездатність, різноманіття функцій та менш очевидну сторону: здатність системи до вдосконалення, перенесення її на іншу апаратну архітектуру тощо. У зв’язку с цим зростає навантаження на серверні додатки та виникає необхідність у постійному покращенні та моніторингу вже працюючих базових серверних програмних продуктів.

РОЗДІЛ 1 ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Опис розвитку високонавантаженої серверної архітектури

За останнє десятиліття ми побачили чимало цікавих нововведень і вдосконалень в сферах баз даних (БД), розподілених систем, а також в способах створення працюючих з ними додатків. Ось деякі чинники, що призвели до цих удосконалень.

- Такі інтернет-компанії, як Google, Yahoo!, Amazon, Facebook, LinkedIn, і Twitter, обробляють колосальні обсяги даних і трафіку, змушує їх створювати нові інструменти, які підходять для ефективної в подібних масштабах.
- Комерційним компаніям доводиться адаптуватися, перевіряти гіпотези мінімальними витратами і швидко реагувати на зміни ринкової обстановки шляхом скорочення циклів розробки і забезпечення гнучкості моделей даних.
- Вільне програмне забезпечення стало надзвичайно популярним і у багатьох випадках є кращим у порівнянні з комерційним ПО і ПО для внутрішнього використання.
- Тактові частоти процесорів не дуже зросли, але багатоядерні процесори стали стандартом, плюс збільшилися швидкості передачі даних по мережі. Це означає подальше зростання паралелізму.
- Навіть невелика команда розробників може створювати системи, розподілені по безлічі машин і географічних регіонів, завдяки такій IaaS – Infrastructure as a service - «інфраструктура як сервіс»), як Amazon Web Services.
- Багато сервісів стали високодоступних; тривалі простої через перебої в обслуговуванні або поточних робіт вважаються всі менш прийнятними.

Високонавантажені даними додатка (data-intensive applications, DIA) відкривають нові горизонти можливостей завдяки використанню цих технологічних удосконалень. Ми говоримо, що додаток є високонавантажених даними (data-intensive), якщо ті становлять основну проблему, з якої воно стикається, - якість даних, ступінь їх складності або швидкість змін, - на відміну від високонавантаженого обчислення (compute-intensive), де вузьким місцем є цикли CPU.

Інструменти і технології, що забезпечують зберігання і обробку даних за допомогою DIA, швидко адаптувалися до цих змін. У центр уваги потрапили нові типи систем баз даних (NoSQL), черги повідомлень, кеші, пошукові індекси, фреймворки для пакетної і потокової обробки і тому подібні технології теж дуже важливі. Багато додатків інколи використовують їх поєднання.

Сама архітектура серверної системи не стоїть на місці. Зі збільшенням вимог до цієї системи зростає складність розробки, з’являються задачі масштабування та децентралізації системи від одного працюючого додатку як єдиного елементу системи до розбиття його на менш комплексні серверні додатки, кожен з яких виконує певну підзадачу основної системи. Для кожної частини архітектури – по хронології розвитку та в залежності від складності проблеми та поставленої задачі спочатку “сервісу”, а згодом “мікросервісу”, виникає необхідність у перевірці коректності роботи того чи іншого сервісу – тестуванні. В залежності від аспекту, що необхідно перевірити, область тестування ПЗ поділена на різні типи.

1.2 Типи тестування високонавантаженого серверного ПЗ

Тестування високонавантаженої системи зводиться до тестування певних її частин та аспектів(коректність роботи функціональної частини, перевірка взаємодії між частинами, швидкість відгуку, тестування роботи при навантаженні тощо) а також їх взаємодії між собою. Кажучи простими словами, для тестування мікро-сервісної архітектури чи сервіс-орієнтованої архітектури будуть використовуватися ті ж самі методи та підходи що і при тестуванні монолітної гексагональної архітектури серверного ПЗ, але з певними доповненнями. Можемо виділити два основних класи тестування: функціональне та нефункціональне. Розглянемо кожне з них. До функціонального тестування належить: Unit-тестування, Integration-тестування, System-тестування, Sanity-тестування, Smoke-тестування, Interface-тестування, Regression-тестування. В основі функціонального типу тестування лежить перевірка коректності роботи бізнес-функцій, тому всі підтипи данного типу тестування будуть реалізовувати ці функції перевірки в тому чи іншому вигляді. Розглянемо декілька основних видів функціонального тестування:

1. Unit-тестування: базовий вид тестування компонент ПЗ, в основі якого лежить перевірка роботи базових функцій, які не можна поділити на інші, підфункції.
2. Integration-тестування: тестування в основі якого лежить перевірка на коректну взаємодію між собою програмних компонентів.
3. System-тестування: тестування програмної системи по принципу “чорної коробки”, де на вхід ми відправляємо певний набір даних та очікуємо отримати точно-визначений результат.
4. Sanity-тестування: вид тестування програмних компонент, що вже були протестовані але згодом були змінені.

5. Smoke-тестування: тестування мінімальної кількості функцій для оптимальної роботи системи.

Для отримання висновку, що система, яка тестується буде мати мінімальну кількість помилок та працювати оптимально, гарною практикою є слідування відношенню тестувальної піраміди на рис. 1.

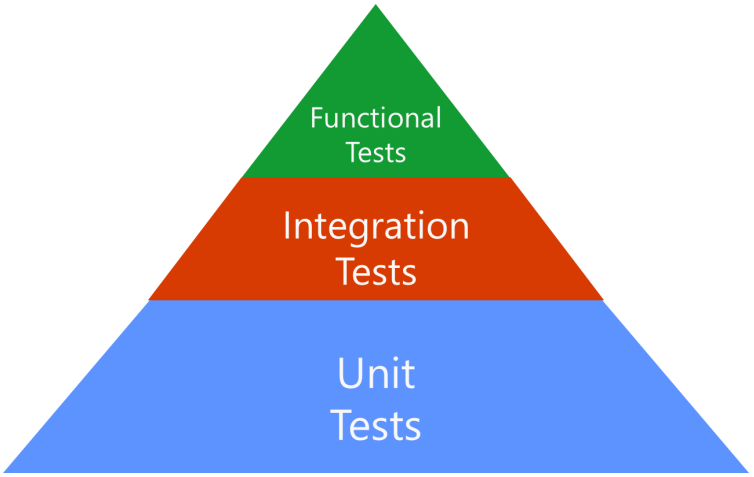


Рисунок 1.1 — Піраміда тестування

До нефункціонального тестування належать такі види: Performance-тестування, Load-тестування, Stress-тестування, Volume-тестування, Security-тестування тощо. Основна ідея нефункціонального тестування – перевірити та описати тести, що необхідні для визначення характеристичних властивостей програмного забезпечення, що можуть бути виміряні різними величинами. В більшості випадків Performance-тестування. Load-тестування та Stress-тестування є спорідненими областями, адже визначають працездатність програмних компонент, хоча і в різній мірі.

1.3 Висновки

Для написання дипломної роботи, була досліджена область тестування програмного забезпечення. Також необхідним стало ознайомитися із класифікацією тестування для подальшого написання програмного продукту роботи.

РОЗДІЛ 2 СПЕЦИФІКА СЕРЕДИ ВЕБ-СЕРВЕРІВ ТА ІНСТРУМЕНТИ ДОСЛІДЖЕННЯ ЇХ НА МОЖЛИВІСТЬ ТЕСТУВАННЯ

2.1 Огляд серверного програмного забезпечення

У випадку користування чи розробки веб-сервісів перша мета – розмістити додаток, що буде виконувати певний бізнес-функціонал. Друга мета – це оптимально налаштувати розгорнутий додаток для коректної роботи для обслуговування користувачів по мережі. Виконання цих двох задач забезпечується спеціальним серверним програмним забезпеченням, що дає можливість не тільки розгорнути налаштовані веб-додатки а і забезпечує механізмами моніторингу, логування, балансування, створення проксі, налаштування з'єднання та оптимального розгортання на певній апаратній архітектурі, забезпечення різними протоколами API тощо. Коротко кажучи створює середу для роботи веб-додатків.

2.2 Огляд віртуальної середи виконання серверного забезпечення JVM

Для оптимальної та швидкої роботи серверів необхідна певна середа, яка буде адаптуватися під апаратну архітектуру машини, на якій працює данна серверна система. Ця середа – певне низькорівневе програмне забезпечення, що проводить адаптацію працюючих в цій середі додатків, уніфікує виконання програм, забезпечує кросплатформність – результат виконання програм в середі на різних апаратних архітектурах буде мати однаковий результат. Можемо виділити деякі поширені віртуальні машини та платформи розробки: JVM(Java virtual machine), .NET тощо.

JVM – специфікація програмного забезпечення, яке виконує код та надає середу виконання для цього коду. Це віртуальна машина, яка

дозволяє запускати Java-програми в портативному режимі. Основний огляд архітектури Java Virtual Machine можна побачити на рис. 2.1.

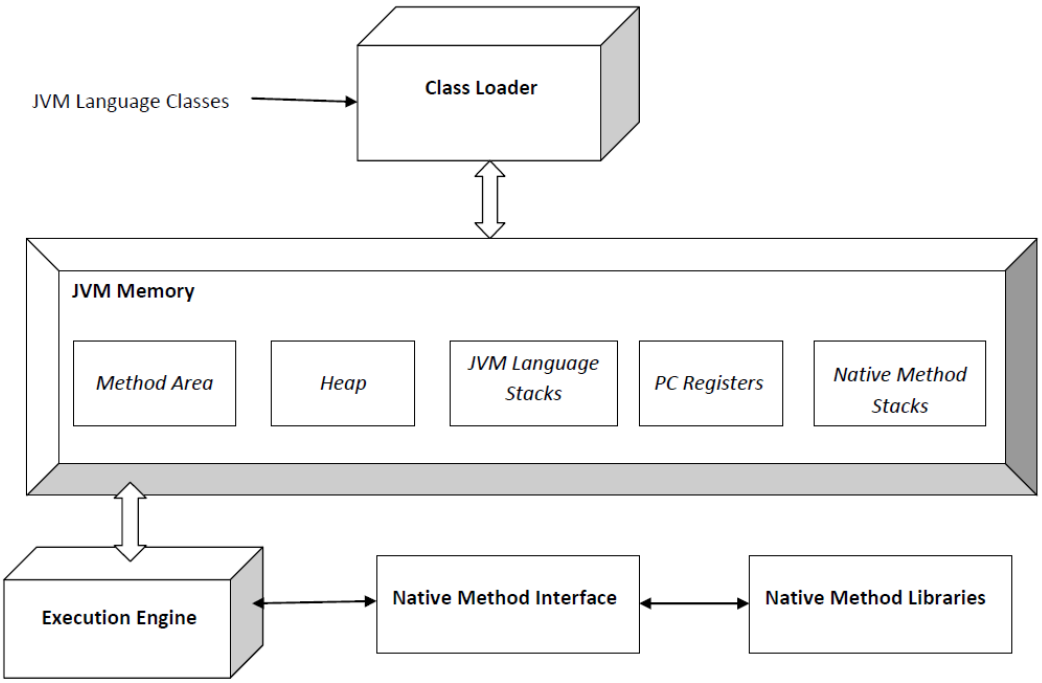


Рисунок 2.1 — Архітектура JVM

Можемо побачити, що віртуальна машина реалізує механізми роботи з пам’яттю, інтерпритації раніше скомпільованого коду, виконавчі механізми, розділи роботи з архітектурою мови програмування Java. Механізм виконання в JVM складається з завантажувача класів та виконання коду. Останнє включає в себе керування доступом до системних ресурсів. Механізм виконання JVM стоїть між роботою програми, з її запитами на файлові, мережеві ресурси та ресурси пам’яті, та операційною системою, яка забезпечує ці ресурси.

2.3 Класифікація серверів

Не зважаючи на те, що серверне програмне забезпечення сфокусовано на такій області використання як обслуговування

користувачів по мережі, можемо виділити певну класифікацію цих серверів. Загалом можемо поділити на такі підтипи:

1. Файлові сервери
2. Print сервери
3. Сервери веб-додатків
4. DNS сервери
5. Поштові сервери
6. Веб-сервери
7. Сервери баз-даних
8. Віртуальні сервери
9. Proxy сервери
10. Сервери моніторингу та менеджменту

Файлові сервери зберігають та розподіляють файли. Кілька клієнтів або користувачів можуть обмінюватися файлами, що зберігаються на сервері. Крім того, централізоване зберігання файлів пропонує простіші рішення для резервного копіювання та відмовостійкості, ніж спроби забезпечити безпеку та цілісність файлів на кожному пристрої в організації. Апаратне забезпечення файлового сервера може бути розроблено для максимізації швидкості читання та запису для підвищення продуктивності.

Сервери друку дозволяють керувати та розподіляти функції друку. Замість того, щоб приєднувати принтер до кожної робочої станції, єдиний сервер друку може відповідати на запити друку від численних клієнтів. Сьогодні деякі великі та високоякісні принтери мають власний вбудований сервер друку, що позбавляє потреби в додатковому комп'ютерному сервері друку. Цей внутрішній сервер друку також функціонує, відповідаючи на запити друку від клієнта.

Сервери програм запускають програми замість клієнтських комп'ютерів, що запускають програми локально. Сервери додатків часто запускають ресурсоємні програми, якими користується велика кількість

користувачів. Це знімає необхідність у кожного клієнта мати достатньо ресурсів для запуску програм. Це також позбавляє від необхідності встановлювати та підтримувати програмне забезпечення на багатьох машинах, на відміну лише від одного.

Сервери системи доменних імен (DNS) - це сервери додатків, які забезпечують роздільну здатність імен для клієнтських комп'ютерів шляхом перетворення легко зрозумілих людям імен у машиночитані IP-адреси. Система DNS - це широко розповсюджена база даних імен та інших серверів DNS, кожен з яких може бути використаний для запиту невідомого імені комп'ютера. Коли клієнту потрібна адреса системи, він відправляє DNS-запит з ім'ям потрібного ресурсу на DNS-сервер. DNS-сервер відповідає необхідною IP-адресою зі своєї таблиці імен.

Поштові сервери - це дуже поширений тип сервера додатків. Поштові сервери отримують електронні листи, надіслані користувачеві, і зберігають їх до запиту клієнта від імені зазначеного користувача. Наявність сервера електронної пошти дозволяє постійно правильно налаштовувати та підключати одну мережу до мережі. Тоді він готовий надсилати та отримувати повідомлення, а не вимагати, щоб кожна клієнтська машина мала власну підсистему електронної пошти, яка постійно працює.

Одним з найпоширеніших типів серверів на сучасному ринку є веб-сервер. Веб-сервер - це особливий вид сервера додатків, на якому розміщуються програми та дані, запитувані користувачами через Інтернет або інтрамережу. Веб-сервери реагують на запити браузерів, що працюють на клієнтських комп'ютерах, щодо веб-сторінок або інших веб-служб. Поширені веб-сервери включають веб-сервери Apache, сервери Microsoft Internet Information Services (IIS) та сервери Nginx.

Обсяг даних, що використовуються компаніями, користувачами та іншими службами, приголомшує. Значна частина цих даних зберігається в базах даних. Бази даних повинні бути доступними для кількох клієнтів у

будь-який момент часу і можуть вимагати надзвичайної кількості дискового простору. Обидві ці потреби добре підходять для пошуку таких баз даних на серверах. Сервери баз даних запускають програми баз даних і реагують на численні запити клієнтів. Поширені серверні програми баз даних включають Oracle, Microsoft SQL Server, DB2 та Informix.

На відміну від традиційних серверів, які встановлюються як операційна система на апаратному забезпеченні машини, віртуальні сервери існують лише як визначено в рамках спеціалізованого програмного забезпечення, яке називається гіпервізор. Кожен гіпервізор може одночасно запускати сотні або навіть тисячі віртуальних серверів. Гіпервізор представляє серверу віртуальне обладнання, ніби це справжнє фізичне обладнання. Віртуальний сервер використовує віртуальне обладнання, як зазвичай, а гіпервізор передає фактичні потреби в обчисленнях і сховищі на реальне апаратне забезпечення, яке спільно використовується між усіма іншими віртуальними серверами.

Проксі-сервер виступає посередником між клієнтом і сервером. Проксі-сервер, який часто використовується для ізоляції або клієнтів, або серверів з метою безпеки, приймає запит від клієнта. Замість того, щоб відповісти клієнту, він передає запит іншому серверу або процесу. Проксі-сервер отримує відповідь від другого сервера, а потім відповідає початковому клієнту, ніби він відповідає самостійно. Таким чином, ні клієнту, ні відповідаючому серверу не потрібно безпосередньо підключатися один до одного.

Деякі сервери існують для моніторингу та управління іншими системами та клієнтами. Існує багато типів серверів моніторингу. Деякі з них слухають мережу і отримують кожен запит клієнта та відповідь сервера, але деякі не запитують і не відповідають на дані самі. Таким чином, сервер моніторингу може відстежувати весь трафік у мережі, а також запити та відповіді клієнтів та серверів, не втручаючись у ці операції. Сервер моніторингу реагуватиме на запити клієнтів моніторингу,

таких як ті, що запускаються адміністраторами мережі, які стежать за станом мережі.

2.4 Структура серверного програмного забезпечення Tomcat

Розглянемо архітектуру серверного програмного забезпечення на прикладі одного з найпопулярніших серверів – Tomcat. Він може виконувати роль не тільки сервлетного контейнеру – програми, що виконує певну бізнес логіку, але і сервер веб-додатків. В мікросервісній архітектурі, він може використовуватися з можливістю зменшення витрати ресурсів розгортання та виконання, що забезпечується його характеристиками легкості та стабільності.

Оскільки, як було сказано вище, Tomcat – сервер розгортання сервлетних додатків та раз з тим веб-сервер, він виконує роль налаштування з'єднань з мережею та розгортання сервлетів. Структуру можемо побачити на рисунку 2.2.

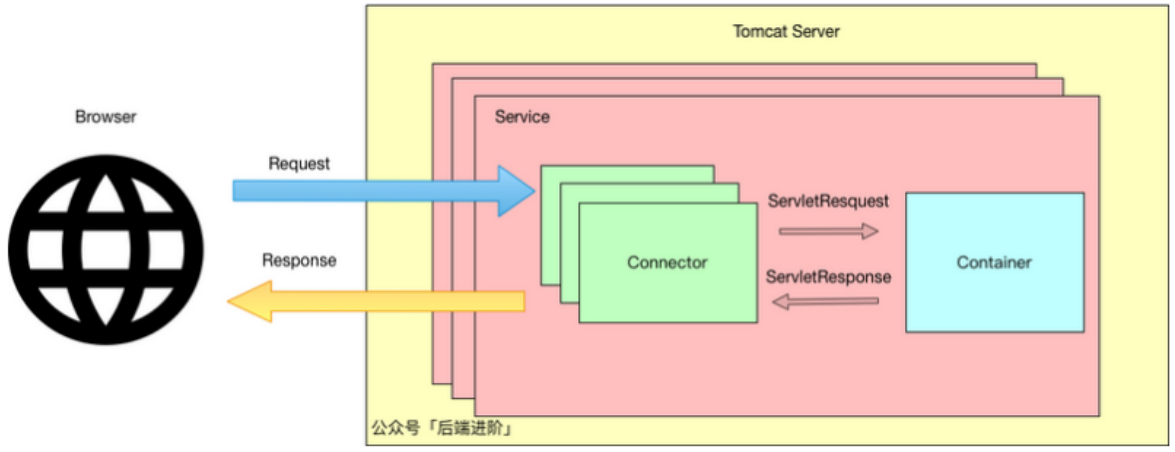


Рисунок 2.2 — Структура серверу Tomcat

Сервер складається з компонент - сервісів, що дозволяють розділити відповідальність роботи додатків, що на них розгортаються, та апаратні ресурси. За-замовчуванням, основний сервіс для розгортання веб-додатків

– Catalina. В той час, сервіс може зберігати багато елементів з’єднання. Це зумовлюється підтримкою данною серверною технологією різних мережевих проколів, таких як HTTP/1.1, HTTP/2, AJP тощо. В структурі сервісу також присутній контейнер – логічний розділ налаштований на розгортання веб-додатків. З’єднувачі комунікують з контейнером використовуючи ServletRequest та ServletResponse об’єкти. Вигляд файлу конфігурування серверу зображений на рис. 2.3.

```
<Server port="8005" shutdown="SHUTDOWN">

  <Service name="Catalina">

    <Connector connectionTimeout="20000" port="8080" protocol="HTTP/1.1" redirectPort="8443" U
RIEncoding="UTF-8"/>

    <Connector port="8009" protocol="AJP/1.3" redirectPort="8443"/>

    <Engine defaultHost="localhost" name="Catalina">

      <Host appBase="webapps" autoDeploy="true" name="localhost" unpackWARs="true">

        <Context docBase="handler-api" path="/handler" reloadable="true" source="org.eclipse.j
st.jee.server:handler-api"/>
      </Host>
    </Engine>
  </Service>
</Server>
```

Рисунок 2.3 — Вигляд файлу налаштування веб-серверу Tomcat

Поєднувані компоненти веб-серверу зберігають всі основні види мережевих протоколів, інкапсулюючи деталі мережевого підключення та обробки вводу-виводу. Далі запит передається на обробку до контейнеру. Клас, що реалізує даний функціонал – ProtocolHandler. ProtocolHandler є типом інтерфейсу та реалізує обробку різних протоколів шляхом реалізації ProtocolHandler, таких як Http11AprProtocol. Структура спадкування класів зображена на рис. 2.4.

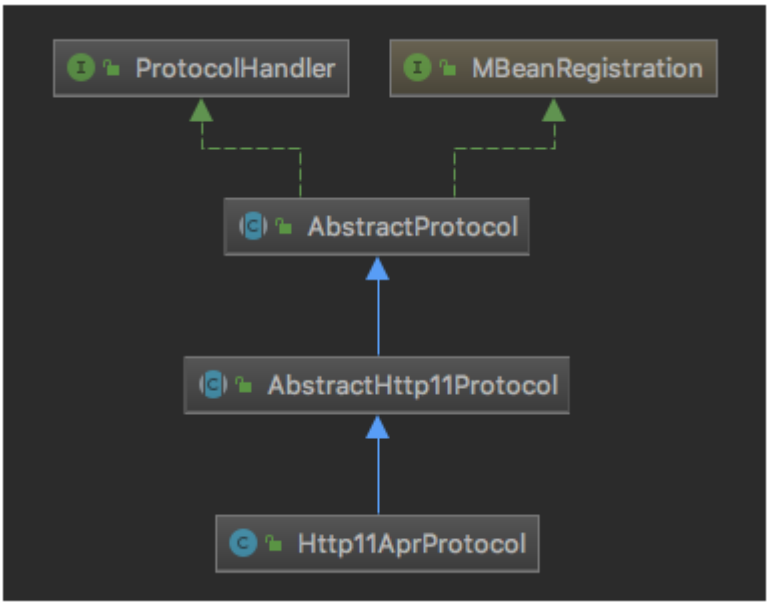


Рисунок 2.4 — Структура спадкування інтерфейсу з’єднання

ProtocolHandler інкапсулює обробку мережевого з’єднання, перетворює байтову послідовність запиту мережевого протоколу у модель серверного запиту, що реалізовується класом Request. В подальшому новостворений екземпляр вищезазначеного класу проходить процес адаптації до використання в рамках сервлетів. ProtocolHandler включає такі три компоненти, реалізовані класами: Endpoint, Processor, Adapter. Endpoint компоненти використовуються для обробки запитів низькорівневих протоколів. Компоненти кінцевої точки використовуються для обробки базових мережевих з'єднань SocketAprEndpoint має внутрішній клас SocketProcessor, який відповідає за перетворення отриманих запитів Socket в об'єкти Request для AprEndpoint. SocketProcessor реалізує інтерфейс Runnable. Для обробки він матиме спеціальний пул потоків. Наразі Tomcat має лише один клас реалізації адаптера, CoyoteAdapter. Основна функція адаптера - адаптувати об'єкт Request до об'єкта Request, який контейнер може розпізнати. Дана схема описана на рис. 2.5.

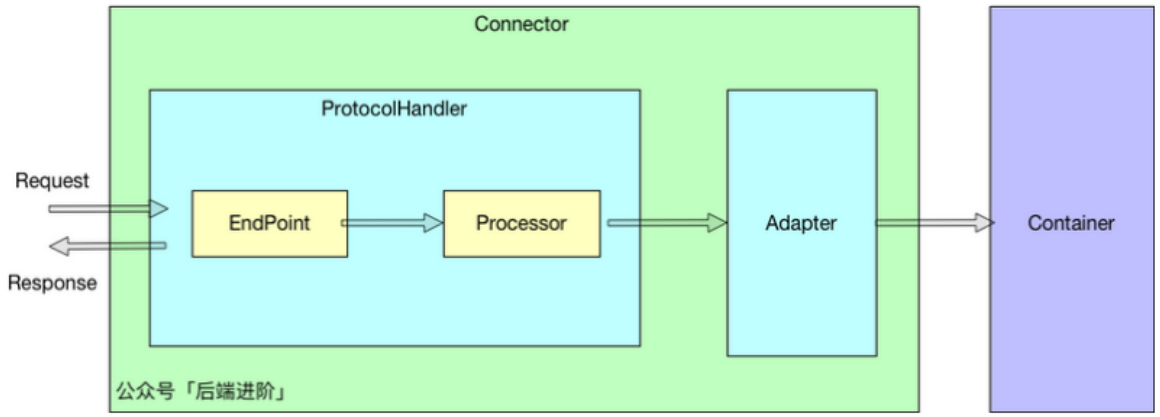


Рисунок 2.5 — Структура з’єднувального компоненту серверної архітектури

Як було зазначено вище, для розгортання та виконання веб-додатків реалізованої бізнес-моделі відповідають контейнери. Чотири контейнери розроблені в Tomcat, це Engine, Host, Context та Wrapper. Їх взаємодія показана на рис. 2.6.

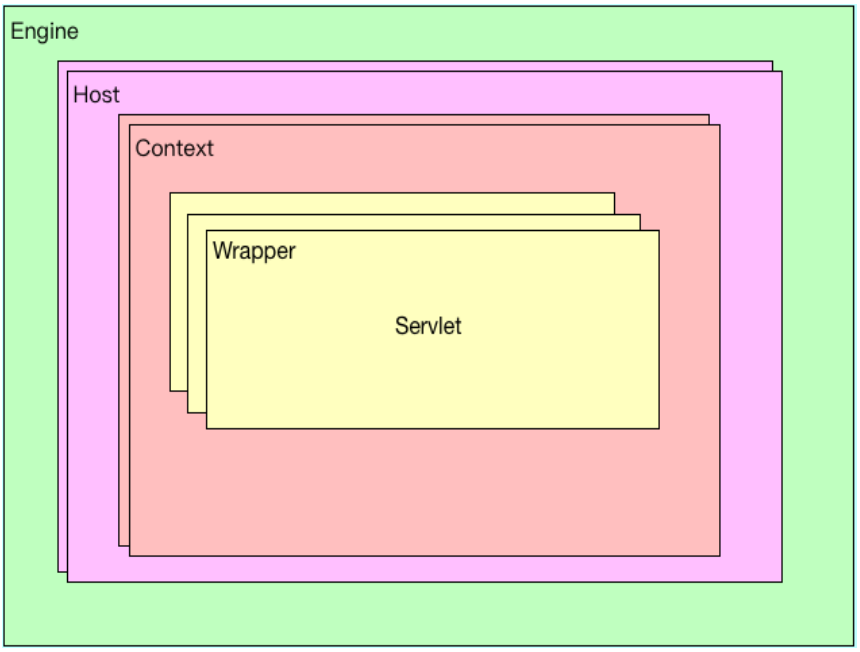


Рисунок 2.6 — Структура компонентів контейнера

- Engine: представляє механізм віртуального хоста. Сервер Tomcat має лише один механізм. Всі запити з'єднувача передаються в цей компонент, а механізм передається відповідному виртуальному хосту для обробки запитів.
- Host: представляє віртуальний хост. Контейнер може мати більше одного віртуального хоста. Кожен хост має своє власне доменне ім’я. У Tomcat веб-додаток представляє віртуальний хост.
- Context: Представляє контейнер додатків, віртуальний хост може мати кілька програм, кожен каталог у веб-додатках представляє контекст, і кожна програма може налаштувати кілька сервлетів.

Як видно на рис. 2.6 вище, взаємозв'язок між компонентами контейнера є від великого до малого, тобто відносини батько-родич, які утворюють деревоподібну структуру. Їхні класи реалізації реалізують інтерфейс Container, який контролює взаємозв'язок між компонентами контейнера.

У Tomcat багато компонентів. Компоненти реалізують інтерфейс життєвого циклу, а Tomcat управляє життєвим циклом цих компонентів за допомогою механізму подій.

Ідея дизайну контейнера Tomcat насправді базується на ідеї комбінованого дизайну. Найбільшою перевагою комбінованого дизайну є те, що він може вільно додавати вузли, що робить компоненти контейнера Tomcat дуже простими для розширення та відповідності принципу open-closed шаблону дизайну.

Щодо взаємодії контейнерних компонентів виникає питання: Коли надходить запит, як Tomcat ідентифікує запит і передає його певному сервлету для обробки? З набору контейнерів видно, що порядок їх виклику можна описати схемою, зображеною на рис. 2.7.

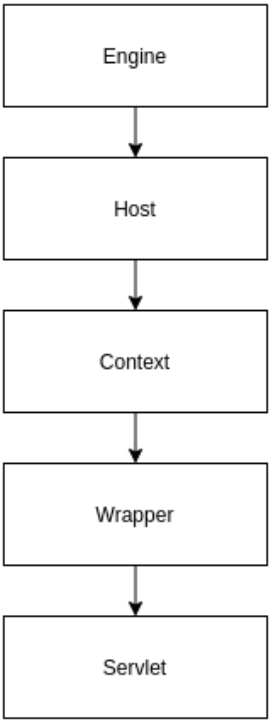


Рисунок 2.7 — Порядок виклику компонент

Щодо механізму знаходження сервлетів технології tomcat алгоритм полягає у використанні компонентів Mapper для позиціонування. Основною основною функцією Mapper є збереження взаємозв'язку шляхів доступу між компонентами контейнера. Загальна схема позначена на рис. 2.8.

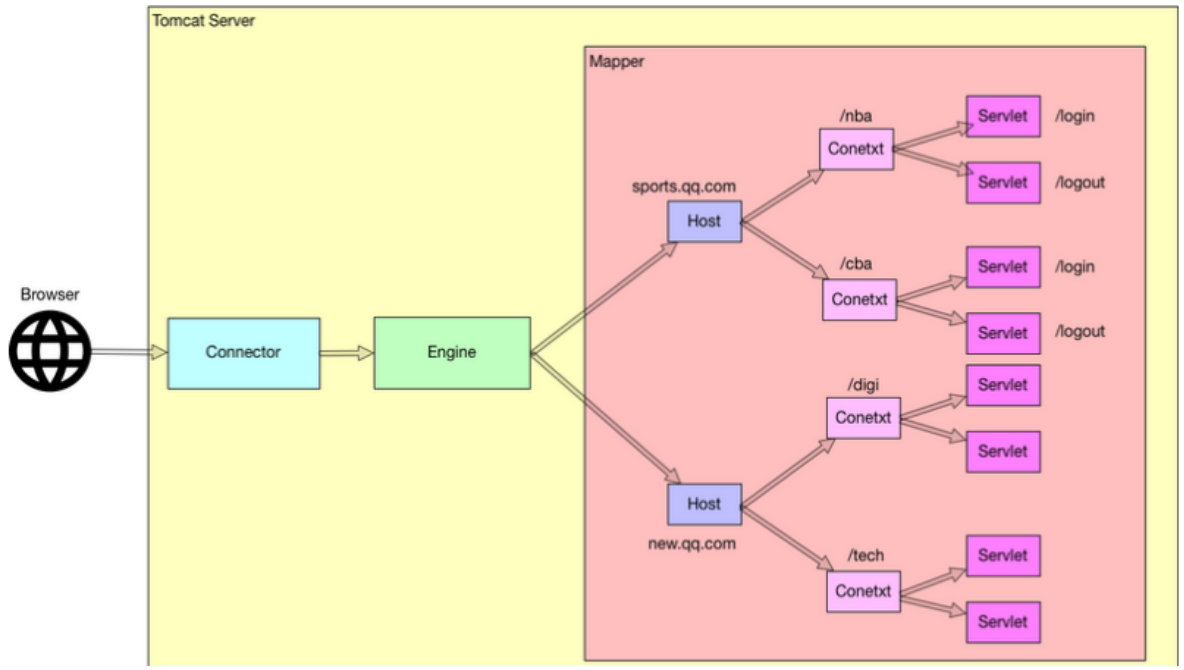


Рисунок 2.8 — Загальна схема роботи серверної технології Tomcat

2.5 Структура серверного програмного забезпечення Jetty

Jetty – сервлетний двигун. Його архітектура порівняно проста, це масштабований та дуже гнучкий сервер. Він має базову модель даних. Ця модель даних є обробником. Всі компоненти, які можна розширити, можна додати на сервер як обробник. Jetty допоможе вам керувати цими обробниками.

На рисунку 2.9 представлена основна архітектурна схема Jetty. Ядро Jetty складається з сервера та коннектора. Весь компонент сервера базується на контейнері Handler. Це схоже на контейнер Tomcat's Container. Іншим важливим компонентом Jetty є Connector, який відповідає за прийняття запитів на підключення клієнта та присвоєння запитів черзі обробки для виконання.

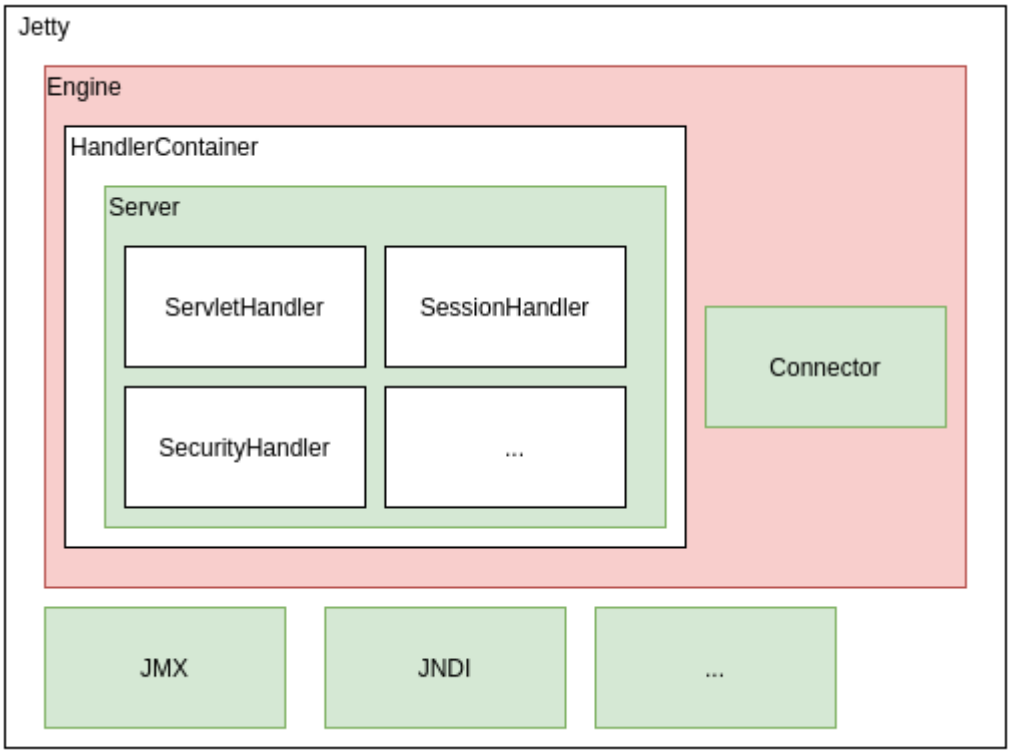


Рисунок 2.9 — Структура серверного ПЗ Jetty

У Jetty є кілька компонентів, які можна розширити поверх нього. Як і JMX, ми можемо визначити деякі MBeans для додавання на сервер, ці Beans будуть працювати разом, коли сервер запускається.

Ядро Jetty побудовано навколо класу Server. Клас Server успадковує Handler і поєднує Connector і Container. Container - це контейнер, який управляє Mbean технологією. Розширення сервера Jetty's в основному полягає в реалізації обробника та додаванні обробника на сервер. Правила доступу для виклику цих обробників містяться на сервері.

Управління життєвим циклом усіх компонентів Jetty базується на шаблоні спостерігача, подібному до управління Tomcat. Кожен компонент міститиме спостерігача. Коли ініціюються такі події, як запуск, збій або зупинка, ці слухачі будуть викликані. Це найпростіший спосіб проектування, набагато простіший, ніж LifeCycle Tomcat.

Jetty - веб-сервер та механізм контейнеризації сервлетних додатків, що підтримує механізми HHTP/2, WebSocket, OSGi, JMX, JNDI, JAAS

тощо. Ці компоненти програмного забезпечення є open source та відкриті для комерційного використання.

Особливості технології:

- Повнофункціональна та стандартизована
- Технологія має відкритий код
- Гнучка
- Має можливість вбудови
- Асинхронна
- Має можливість масштабування

Таким чином, можемо зробити висновок, що данна серверна технологія реалізує основну частину функціональності роботи сучасних веб-серверних архітектур.

2.6 Опис серверних технологій на інших мовах програмування

Розглянемо серверні технології, що реалізуються іншими мовами програмування. Одна з найпопулярніших з них є Node.js. Ця серверна технологія побудована для розробки мережевих додатків, проте її архітектура не використовує модель обробки паралельних потоків операційної системи. Даний аспект спрощує її використання. Також важливою характеристикою даної технології є фокус на відсутності блокування процесів. Уникнення блокування обумовлюється присутністю проміжних механізмів між роботою сервера та потоком вводу-виводу. Це дозволяє розробляти маштабовані системи. Основою розробки додатків бізнес-моделі є реалізація моделі подій з використанням циклу подій (event-loop) як основу оточення. Використовуючи механізми мови JavaScript такі як функції зворотного виклику це дає змогу приховати сам подієвий цикл від користувача. Можна виділити такі плюси використання даної серверної технології: реалізація механізму роботи асинхронного

потоку вводу-виводу, модель подій, простота у використанні завдяки мові JavaScript, велика спільнота користувачів, зручний пакетний менеджер. Із мінусів виділимо такі аспекти: Неможливість реалізації маштабованості, складність роботи з реляційними базами даних.

Підсумовуючи, технологія Node.js дозволяє використовувати модель однопроцесорного серверного програмування, що дозволяє створювати невеликі але потужні, з можливістю реалізації обслуговування у реальному часі, веб-додатки.

Розглянемо інший не менш популярний фреймворк для розробки архітектури веб-додатків Ruby on Rails. Цей фреймворк працює на мові програмування Ruby. Розробка та реалізація веб-додатків відбувається по моделі MVC - модель, вигляд, контроллер. Із гарних сторін можна виділити: ефективність часу, велика кількість корисних інструментів та бібліотек, величезна та активна спільнота, чітке дотримання стандартів.
Із негативних сторін виділяють: нестачу гнучкості, низька швидкість виконання та низька популярність.

2.7 Гексагональна архітектура

Оптимальним способом для створення веб-додатків та впровадження серверного програмного забезпечення буде впровадження гексагональної архітектури як способу розробки серверного програмного забезпечення бізнес логіки. Можемо виділити такі три принципи гексагональної архітектури:

1. Розділення функціоналу клієнтської частини, бізнес-логіки та серверної частини.
2. Залежності переходять від сторони користувача та сервера до бізнес-логіки.

3. Ізоляція зв’язків відбувається шляхом використання механізмів портів та адаптерів.

Розглянемо перший принцип розділення функціоналу клієнтської частини, бізнес-логіки, серверної частини. Цей принцип впроваджує розподілення коду на три великі частини, що зображені на рисунку 2.10.

Перша частина, зліва на рис. 2.10 – це клієнтська частина. Це сторона, через яку користувач або зовнішні програми будуть взаємодіяти з додатком. Він містить код, який дозволяє ці взаємодії. Як правило, тут знаходиться код інтерфейсу користувача, HTTP-маршрути для API, серіалізації JSON для програм, які надсилають запити до додатку.

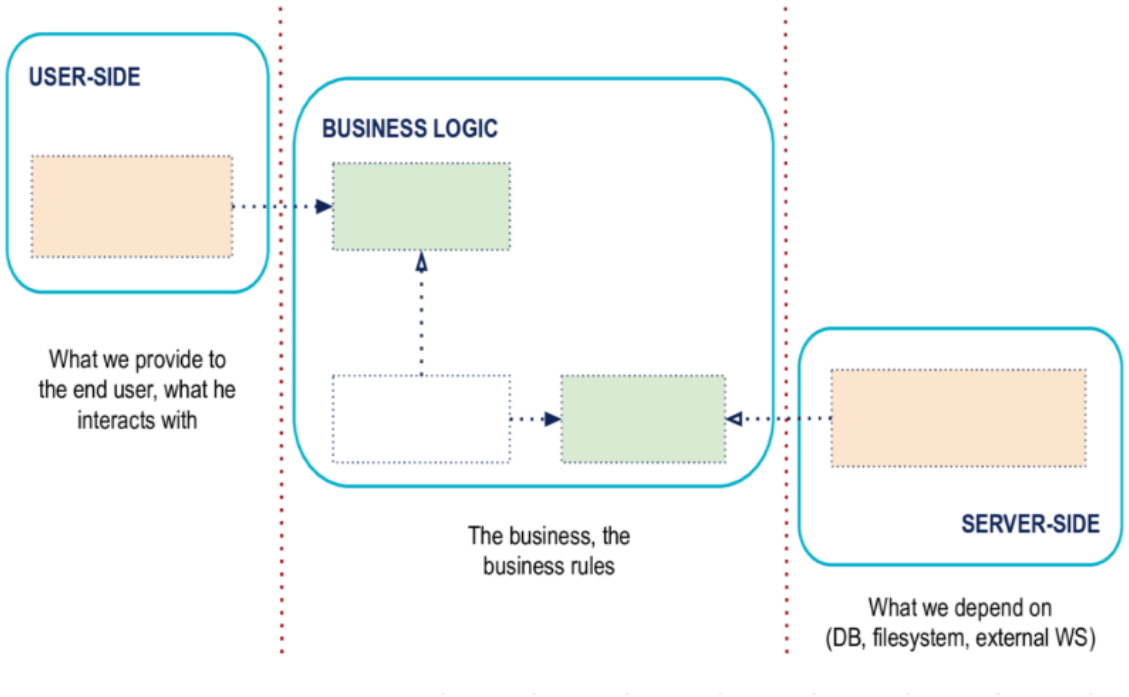


Рисунок 2.10 — Перший принцип гексагональної архітектури

Друга частина, що розташована в центрі на рис. 2.10 – програмна частина бізнес-логіки. Це та частина, яку ми хочемо відокремити як від серверної частини, так і від клієнтської частини. Він містить весь код, який відноситься та реалізує бізнес-логіку. Бізнес-лексика та чиста ділова логіка, яка стосується конкретної проблеми, яка вирішує проблему. В

ідеалі, експерт домену, який не знає, як кодувати, міг би прочитати фрагмент коду в цій частині та вказати вам на невідповідність.

Третя частина – суто технічна. Ця частина містить важливі деталі інфраструктури, такі як код, який взаємодіє з базою даних, здійснює виклики до файлової системи або код, який обробляє виклики HTTP до інших програм.

Першою важливою особливістю цього поділу є те, що воно розділяє проблеми. У будь-який час ви можете зосередитись на одній логіці, майже незалежно від інших двох: логіці на стороні користувача, бізнес-логіці або логіці на стороні сервера. Їх легше зрозуміти, не змішуючи їх, і обмеження кожної логіки мають менший вплив на інші.

Інша характеристика полягає в тому, що ми ставимо бізнес-логіку на перший план нашого коду. Його можна виділити в каталог або модуль, щоб зробити це явним для всіх розробників. Його можна визначити, вдосконалити та протестувати. Це важливо, оскільки врешті-решт саме реалізація бізнес-логіки розробниками йде на виробництво.

З точки зору автоматизованих тестів, ми досягнемо успіху в тестуванні з розумними зусиллями:

- Вся бізнес-логіка окремо.
- Інтеграція між користувацькою та бізнес-логікою незалежно від серверної частини
- Інтеграція між бізнес-логікою та стороною сервера незалежно на стороні користувача

Щоб проілюструвати ці принципи більш конкретно, ми використаємо невеликий приклад, використаний під час заходу «Алістер у шестикутнику», запропонований у 2017 році Томасом П'єрреном та самим Алістером Кокберном.

Мета цього невеликого додатка - забезпечити програму командного рядка, яка записує вірші у стандартний вихід консолі.

Щоб правильно проілюструвати три зони (на стороні користувача, бізнес-логіка, на стороні сервера), ця програма буде шукати вірші у зовнішній системі: у файлі. Її архітектура зображена на рисунку 2.11. Ми також могли б підключити цю програму до бази даних, принцип був би однаковим.

У цьому контексті, як ми можемо застосувати цей перший принцип, а саме поділ на три зони? Як розподілити ліворуч (що рухає), в центрі (основний бізнес) та праворуч (чим керується)?

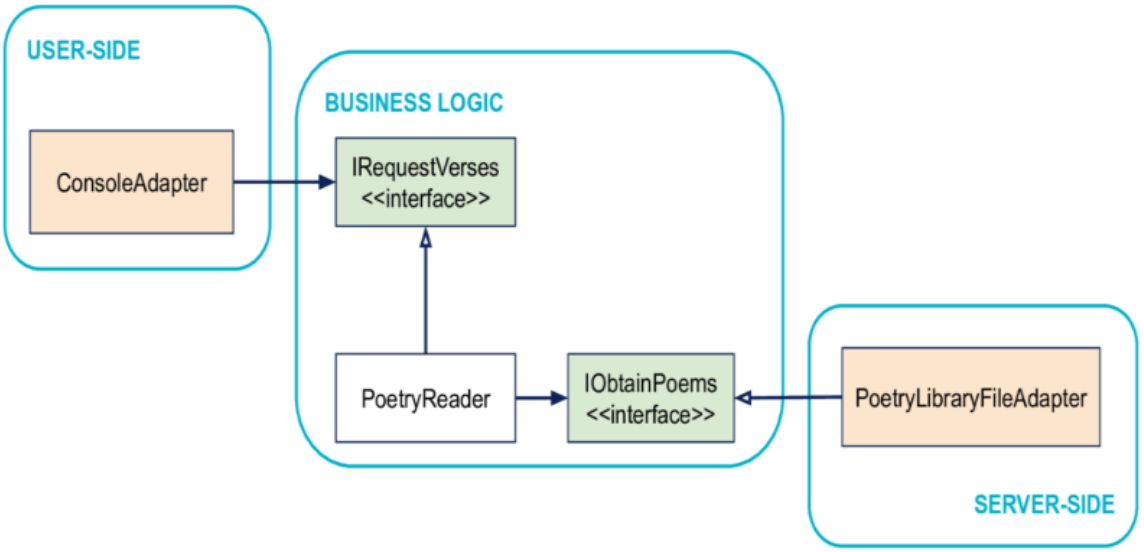


Рисунок 2.11 — Оглядовий приклад принципів гексагональної архітектури

З точки зору користувача, програма представлена у вигляді консольного додатка. Тож концепція консолі буде зліва, на стороні користувача. Через консоль користувач керуватиме доменом.

Технічно в нашому випадку вірші зберігаються у файлі. Це поняття файлу можна знайти праворуч, на стороні сервера. Сторона бізнес-логіки зробить запит своїх віршів, виконуючи запит до серверної частини, конкретно реалізований PoetryLibraryFileAdapter.

Тут, як уже згадувалося вище, ми можемо легко обмінятися джерелом віршів (файлом, базою даних, веб-сервісом). Отже, фактична реалізація джерела як файлу - це технічна деталь технічної реалізації.

Нашим основним бізнес-функціоналом у цьому випадку, що має цінність для користувача, є поняття читання віршів. Наприклад, ми можемо матеріалізувати це поняття в коді за допомогою класу PoetryReader.

З точки зору бізнес-логіки, неважливо, надходить запит із консольного додатка чи іншого, це технічна деталь, яку ми хочемо мати, щоб абстрагуватися. Це якраз одне з початкових намірів: «керуватися користувачем так само, як і тестами». Тому в Business Logic немає поняття консолі. Однак те, що дозволяє наша програма, з точки зору користувача – це запитувати вірші. Саме це поняття ми знайдемо в бізнес-логіці – IRequestVerses, і це дозволить стороні користувача взаємодіяти з логікою бізнесу.

Подібним чином, з точки зору бізнес-логіки, не має значення, чи вірші походять з файлу чи бази даних, ми хочемо мати можливість протестувати наш додаток незалежно від зовнішніх систем. Відсутність поняття файлу в бізнес-логіці. Для роботи домену все ще потрібно отримати вірші. Ми знаходимо це поняття отримання віршів у логіці бізнесу у формі інтерфейсу IObtainPoems. Саме це поняття отримання віршів дозволить домену взаємодіяти із стороною сервера.

Принцип, коли залежності заходять всередину, що зображено на рис. 2.12 – важливий принцип для досягнення мети. Ми вже почали бачити це в попередньому принципі.

Програмою можна керувати як за допомогою консолі, так і за допомогою тестів, у Business Logic немає поняття консолі. Бізнес-логіка не залежить від сторони користувача, це сторона користувача, яка залежить від бізнес-логіки. Сторона користувача (ConsoleAdapter) залежить від поняття

запиту на вірш IRequestVerses (що визначає загальний механізм “запиту на вірш” з боку користувача).

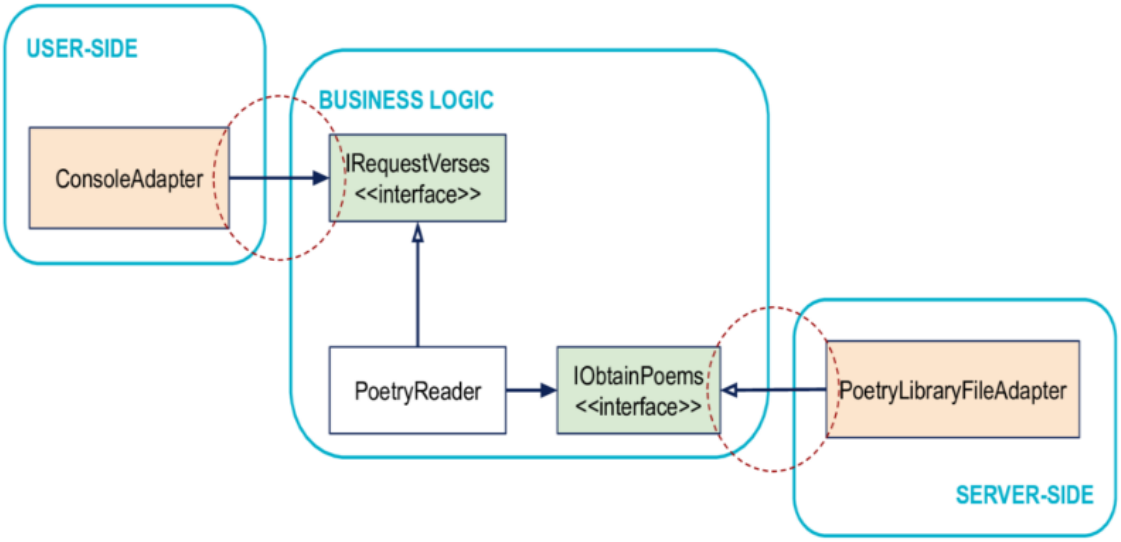


Рисунок 2.12 — Ілюстрація принципу, коли залежності заходять всередину

Подібним чином програму можна протестувати незалежно від її зовнішніх систем, бізнес-логіка не залежить від сторони сервера, вона навпаки. Серверна сторона залежить від бізнес-логіки, через поняття отримання віршів, IObtainPoems. Технічно клас на стороні сервера успадкує інтерфейс, визначений у Business Logic, та реалізує його, ми детально побачимо його нижче, щоб поговорити про інверсію залежностей.

Якщо ми бачимо відносини залежності (<< залежить від ... >>) як стрілки, то цей принцип визначає центральну бізнес-логіку як всередині, а все інше як зовні (рис. 2.13). Ми регулярно знаходимо ці поняття всередині та зовні, коли ми обговорюємо гексагональну архітектуру. Це навіть може бути фундаментальним моментом для запам’ятовування та передавання: залежності входять всередину.

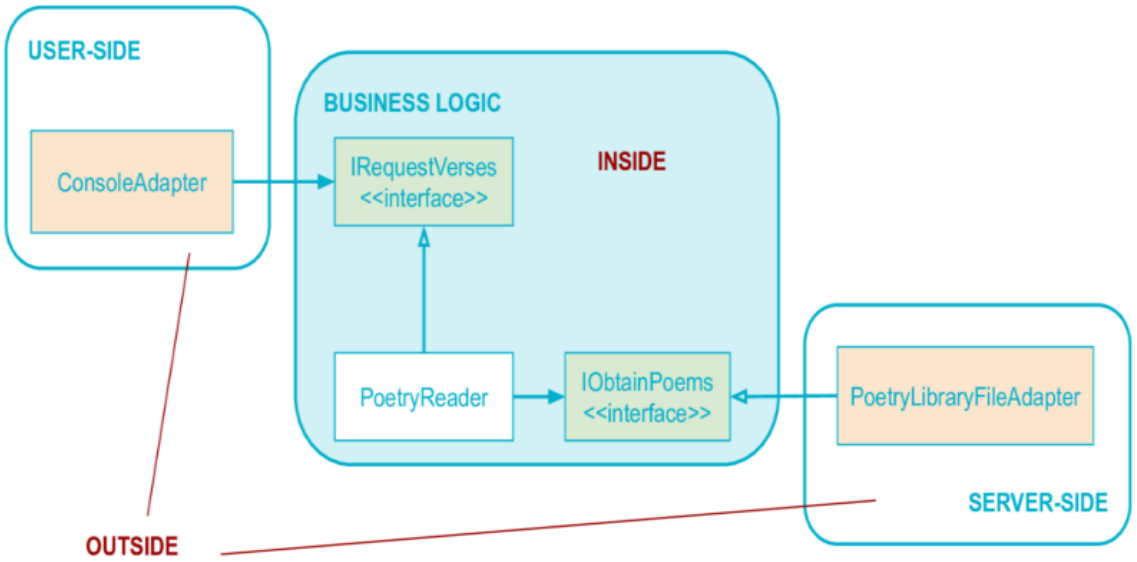


Рисунок 2.13 — Розділення на центральну бізнес логіку та зовнішні компоненти

Іншими словами, все залежить від бізнес-логіки, бізнес-логіка не залежить ні від чого. Алістер Кокберн наполягає на цьому розмежуванні між внутрішньою та зовнішньою сторонами, яка є більш структурованою, ніж різниця між стороною користувача та стороною сервера для вирішення початкової проблеми. Зв’язок відбувається шляхом додавання до бізнес-логіки інтерфесів комунікації(рис. 2.14).

Гексагональна архітектура використовує метафору портів та адаптерів, щоб відобразити взаємодію всередині та зовні. Зображення полягає в тому, що бізнес-логіка визначає порти, на яких усі типи адаптерів можуть бути взаємозамінними, якщо вони відповідають специфікації, визначеній портом.

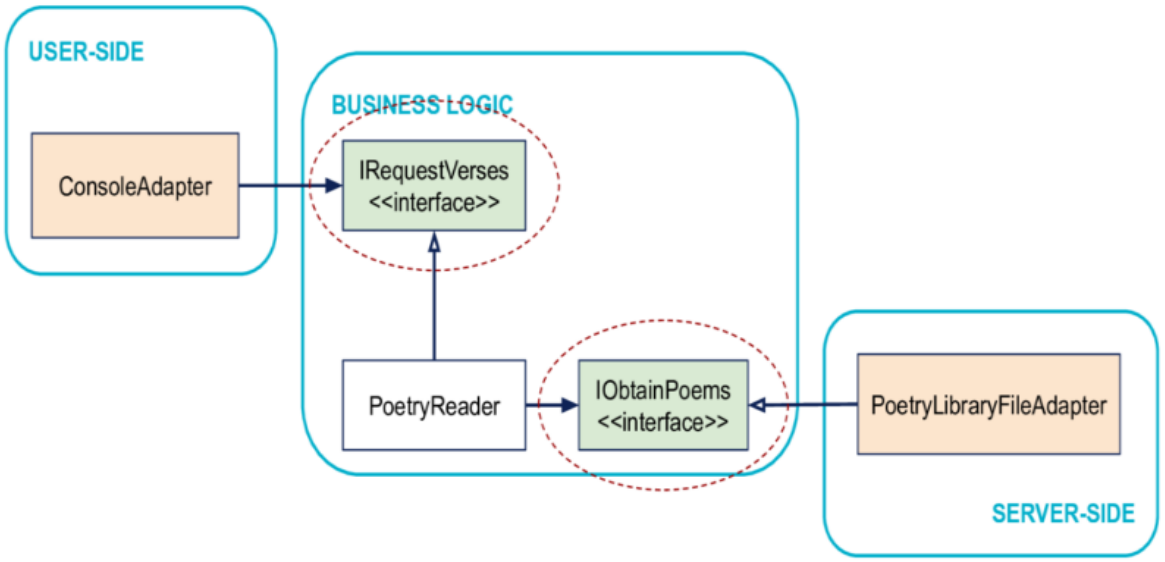


Рисунок 2.14 — Фокус на інтерфейсах бізнес-логіки

Наприклад, ми можемо уявити порт бізнес-логіки, до якого підключатимемо жорстко закодоване джерело даних під час модульного тестування або реальну базу даних в тесті інтеграції. Основна структура розташування портів та адаптерів позначена на рис. 2.15.

Розглядаючи вищезазначену архітектуру та її взаємодію з додатковими компонентами цілої системи можемо назвати її гексагональною, тому що спосіб з’єднання зручніше всього буде позначити у вигляді многокутника.

Окрім принципів, розглянутих вище, ми абсолютно вільно впорядковуємо код у кожній зоні саме так, як хочемо. Що стосується бізнес-коду, всередині, гарною ідеєю є вибрати організацію його модулів (або каталогів) відповідно до бізнес-логіки.

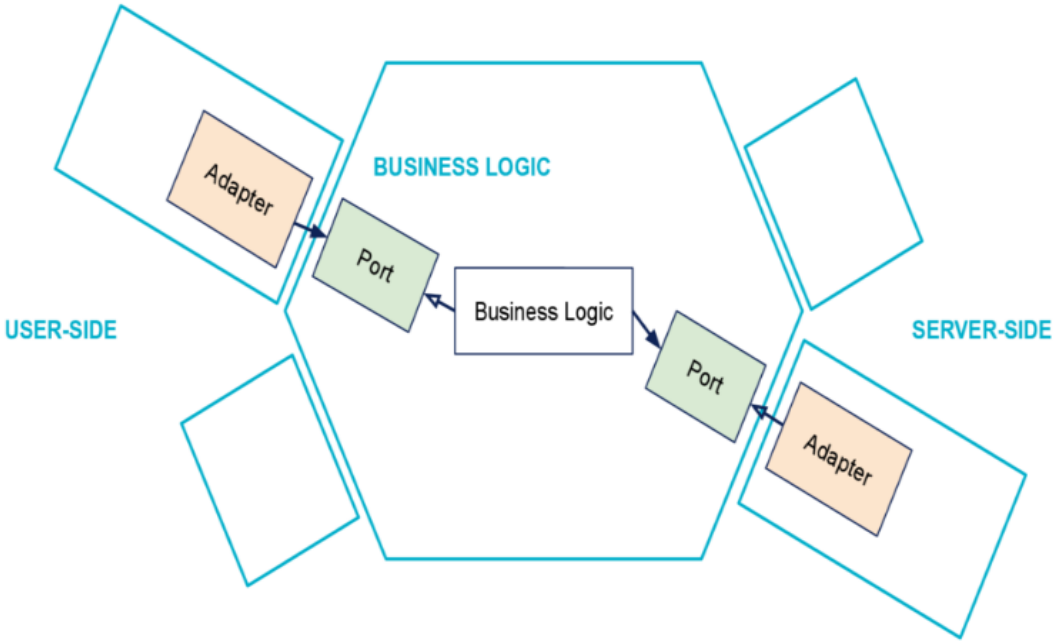


Рисунок 2.15 — Основна структура розташування портів та адаптерів

Однієї організації, якої слід уникати, є групування класів за типами. Наприклад, каталог “порти”, каталог “сховища” (якщо ви використовуєте цей шаблон), або каталог “послуги”. Ідеальний випадок - це можливість відкрити каталог або модуль бізнес-логіки та негайно зрозуміти бізнес-проблеми, які вирішує ваша програма; замість того, щоб бачити лише "сховища", "служби" чи інші каталоги "менеджерів".

2.8 Тестування у гексагональній архітектурі

У загальному випадку роль клієнтського коду може відігравати безпосередньо тестова структура. Дійсно, тестовий код може безпосередньо керувати кодом бізнес-логіки як показано на рис. 2.16.

Код серверної частини повинен керуватися бізнесом. Загалом, якщо ґ мета написати модульний тест, відбувається заміна його на макет чи будь-яку іншу форму тестового модулю, залежно від того, що перевірити на меті. Схема на рис. 2.17.

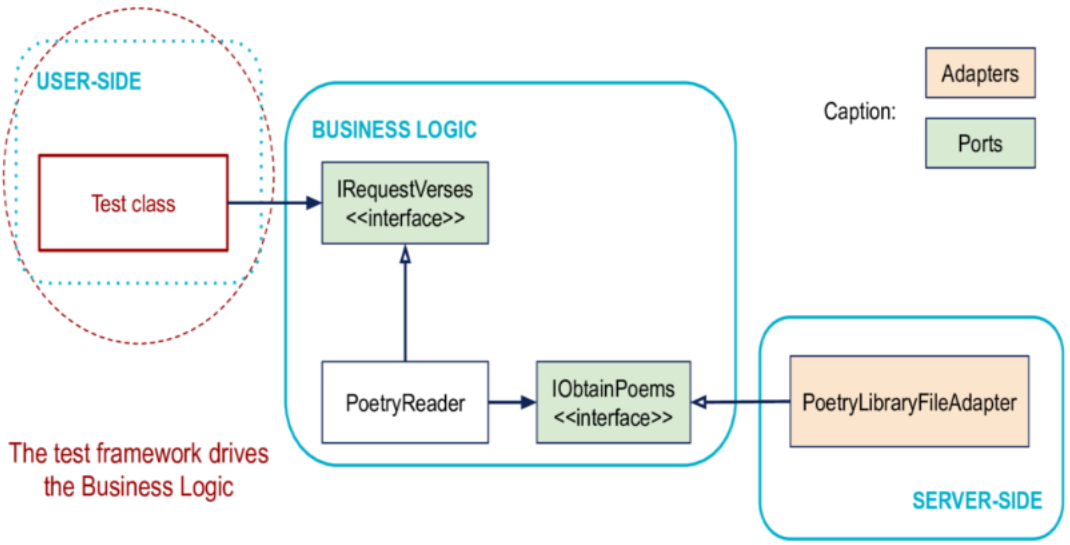


Рисунок 2.16 — Заміна компоненту клієнтської частини на тестувальну

Гексагональна архітектура є гарним компромісом між складністю та потужністю. Але це лише одне рішення серед інших. Для простих випадків це може бути занадто складно, а для складних випадків - занадто просто. Є й інші архітектури програмного забезпечення. Наприклад, Clean Architecture йде далі у формалізації та ізоляції. Або в іншій, але сумісній осі, CQRS дає змогу краще розділяти записи та зчитування.

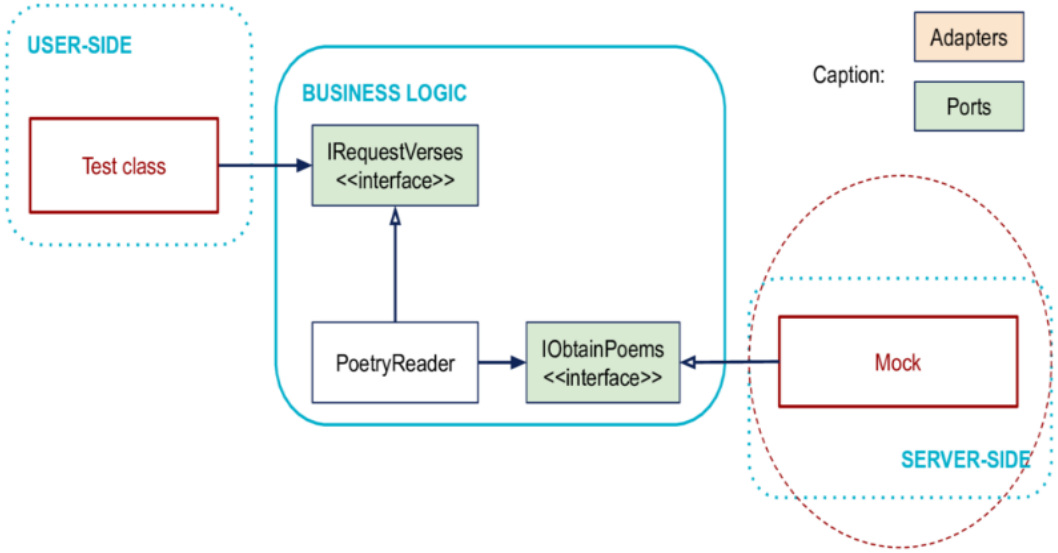


Рисунок 2.17 — Заміна компоненту серверної частини на тестувальну

2.9 Performance-тестування (тестування на ефективність)

Тестування продуктивності - загальна назва тестів, які перевіряють поведінку та ефективність системи. Тестування продуктивності визначає здатність системи до обробки запитів, стабільність, масштабованість, надійність, швидкість та використання ресурсів вашого програмного забезпечення та інфраструктури. Різні типи тестів продуктивності надають вам різні дані.

Перед тестуванням продуктивності важливо визначити бізнес-цілі системи, щоб зрозуміти, чи система поводиться задовільно чи ні відповідно до потреб клієнтів.

Після запуску тестів продуктивності присутня можливість аналізувати різні KPI, такі як кількість віртуальних користувачів, звернення в секунду, помилки в секунду, час відгуку, затримка та байти в секунду (пропускна здатність), а також співвідношення між ними. За допомогою звітів ви можете виявити вузькі місця, помилки, а потім зробити висновки відповідно до необхідності.

Необхідність в тестах на продуктивность присутня, коли на меті перевірка ефективності веб-сайту чи додатків, які можуть поширюватися на тестування серверів, баз даних, мереж тощо. При дотриманні методології водоспаду, необхідність в тестування присутня принаймні кожного разу, коли випускається версія додатку.

2.10 Load-тестування (тестування навантаженням)

Тестування навантаженням - це тип перевірки продуктивності, який перевіряє, як системи функціонують при великій кількості одночасних

віртуальних користувачів, які виконують транзакції протягом певного періоду часу. Іншими словами, тест вимірює, як системи обробляють великі обсяги навантаження.

Тестування навантаженням використвується, коли потрібно визначити, скільки користувачів насправді може обробити серверна система. Налаштування тестів для моделювання різних сценаріїв користувача, які можуть зосередитися на різних частинах вашої системи (наприклад, на сторінці оформлення замовлення). Є можливість визначити, як поводиться навантаження, коли надходить з різних географічних місць, або як навантаження може наростати, а потім вирівняти до стійкого рівня. Тести навантаження слід проводити постійно, щоб забезпечити постійну роботу системи, саме тому її слід інтегрувати у безперервні цикли інтеграції.

2.11 Stress-тестування

Стрес-тест - це тип перевірки продуктивності, який перевіряє верхні межі системи, перевіряючи її при екстремальних навантаженнях. Стрес-тести вивчають, як система поводиться при інтенсивних навантаженнях і як вона відновлюється при поверненні до звичайного використання. Чи такі показники ефективності, як пропускна здатність та час відгуку, такі самі, як до стрибка навантаження? Стрес-тести також виявляють витоки пам'яті, уповільнення, проблеми із безпекою та пошкодження даних.

Стрес-тестування можна провести за допомогою інструментів тестування навантаження, визначивши тестовий приклад з дуже великою кількістю одночасних віртуальних користувачів.

2.10 Моделі відкритого та закритого робочого навантаження

Що стосується моделі навантаження, системи поводяться двома різними способами:

1. Закриті системи: контроль одночасної кількості користувачів.
2. Відкриті системи: контроль рівня прибуття користувачів

Закрита система - це система, де обмежується кількість одночасних користувачів. На повну потужність новий користувач може ефективно увійти в систему лише після виходу іншого. Типовими системами, які поводяться таким чином є:

- кол-центр, де зайняті всі оператори
- продаж веб-сайтів, де користувачі потрапляють у чергу, коли система працює на повну потужність.

Навпаки, відкриті системи не мають контролю над кількістю одночасних користувачів: користувачі постійно надходять, навіть незважаючи на те, що додатки мають проблеми з обслуговуванням. Більшість веб-сайтів поводяться так.

2.12 Висновки

Дослідивши серверне програмне забезпечення, середу виконання, та архітектурні рішення можемо зробити висновки, що всі розглянуті серверні програмні продукти підходять для розгортання сучасних веб-додатків та побудови оптимальної архітектури обслуговування

користувацьких запитів бізнес-моделі. Для дослідження роботи використаних серверних продуктів, було прийнято рішення використовувати стрес-тестування та тестування на ефективність. Ці два методи покажуть, який серверний програмний продукт покаже найкращі результати, при невеликому та стрімкому збільшенні запитів до серверу.

РОЗДІЛ 3 РОЗРОБКА ПРОГРАМНОГО ПРОДУКТУ

3.1 Вступ

Загальну задачу ми можемо описати таким чином: “Розробити програмний продукт, який буде реалізовувати стрес-тестування та тестування на ефективність серверних програмних продуктів”. Наразі можемо спостерігати за появою, оптимізацією нових серверних продуктів, кожен з яких пропонує краще використання пам’яті віртуальної машини, на яких працюють ці програмні продукти, швидкий відгук на користувацькі запити, швидку здатність розробки та впровадження бізнес-логіки веб-додатків. Тому здається доречним розробити програмний продукт, що дозволить проводити аналітику серверної архітектури, що в подальшому дозволить обирати рішення для розгортання користувацького додатку.

При розробці серверної архітектури, часто бувають випадки, коли є невизначеність у виборі серверної частини додатку. Може статися таке, що розробники чи архітектори обирають застару технологію, що в майбутньому приводить до неоптимальної роботи веб-додатку, необхідності написання legacy-коду, що є “застарілим”, низької кількості розробників на ринку праці тощо.

Підсумовуючи, можна сказати, що програмний продукт має бути у змозі коректно зробити запити до веб-серверу, прийняти результат та показати часові ряди запитів, час їх обробки та затримку.

3.2 Постановка задачі

Підійдемо до задачі методом декомпозиції. По-перше необхідно мати апаратну середу, достатню для того, щоб робити висновки, які будуть

максимально приближені до наразі працюючих веб-додатків. В подальшому необхідно буде обрати серверне програмне забезпечення, для якого буде проводитися тестування. Після обрання серверу, необхідно налаштувати Application Programming Interface, що дозволить серверу приймати запити з мережі та створити імітацію роботи бізнес-логіки для цього сервера.

Після створення серверної частини проекту необхідно налаштувати середу тестування. Бажано використовувати ту ж саму архітектуру та використовувати того ж самого провайдера послуг, адже він буде мати спільну локальну мережу. Це дозволить не враховувати перешкоди в мережевому з'єднанні з сервером, адже вони будуть мінімальними. Наприклад у випадку розташування в одному глобальному регіоні чи навіть в одному датацентрі.

Коли середа тестування буде підготовлена до початку тестувальної процедури, можна запускати програмний продукт тестування та робити аналітику щойноотриманих метрик.

3.3 Налаштування апаратної архітектури

Як зазначено вище, для розгортання веб-додатку першопочатково необхідно налаштувати апаратну архітектуру наближену до реальних серверних архітектур. Для цієї мети буде використано послуги веб-сервісу Amazon Web Services EC2. Для повного програмного налаштування оберемо модель хмарних обчислень Platform as a service(послуги надання платформи). Налаштування апаратної архітектури включає в себе обрання певного стеку ресурсів та ряду налаштувань платформи, на якому буде працювати сервер. В якості операційної системи оберемо 64-бітну, на

процесорній архітектурі AMD(x86), unix-подібну Debian(рис. 3.1).

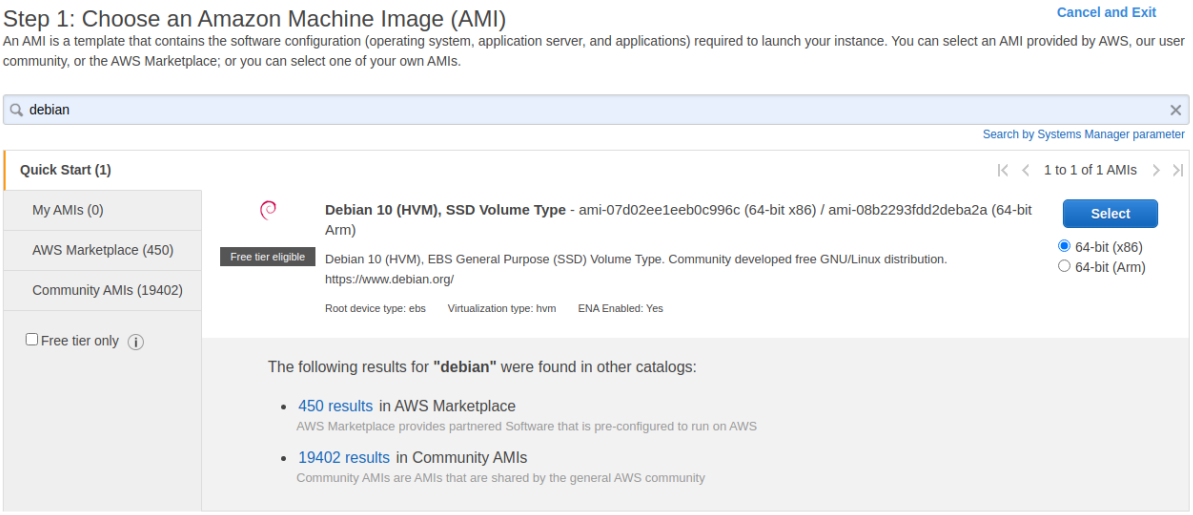


Рисунок 3.1 — Обрана ОС та машина для розгортання веб-серверу

Щодо ресурсної архітектури оберемо m5.2xlarge з 8-ма процесорними ядрами та 32 Гб оперативної пам’яті. Обмеження на швидкість мережевого з’єднання складає 10 Гігабіт/сек(рис. 3.2).

☐	m5	m5.16xlarge	64	256	EBS only	Yes	20 Gigabit	Yes
☐	m5	m5.24xlarge	96	384	EBS only	Yes	25 Gigabit	Yes
☑	m5	m5.2xlarge	8	32	EBS only	Yes	Up to 10 Gigabit	Yes
☐	m5	m5.4xlarge	16	64	EBS only	Yes	Up to 10 Gigabit	Yes
☐	m5	m5.8xlarge	32	128	EBS only	Yes	10 Gigabit	Yes

Рисунок 3.2 — Обрана ресурсна архітектура

Більш детальні налаштування машини виконання залишимо запропонованими самим сервісом та оберемо розгортання однієї машини, оскільки при розгортанні на декількох машинах однієї серверної архітектури довелося б також впроваджувати балансувальник навантаження(рис. 3.3).

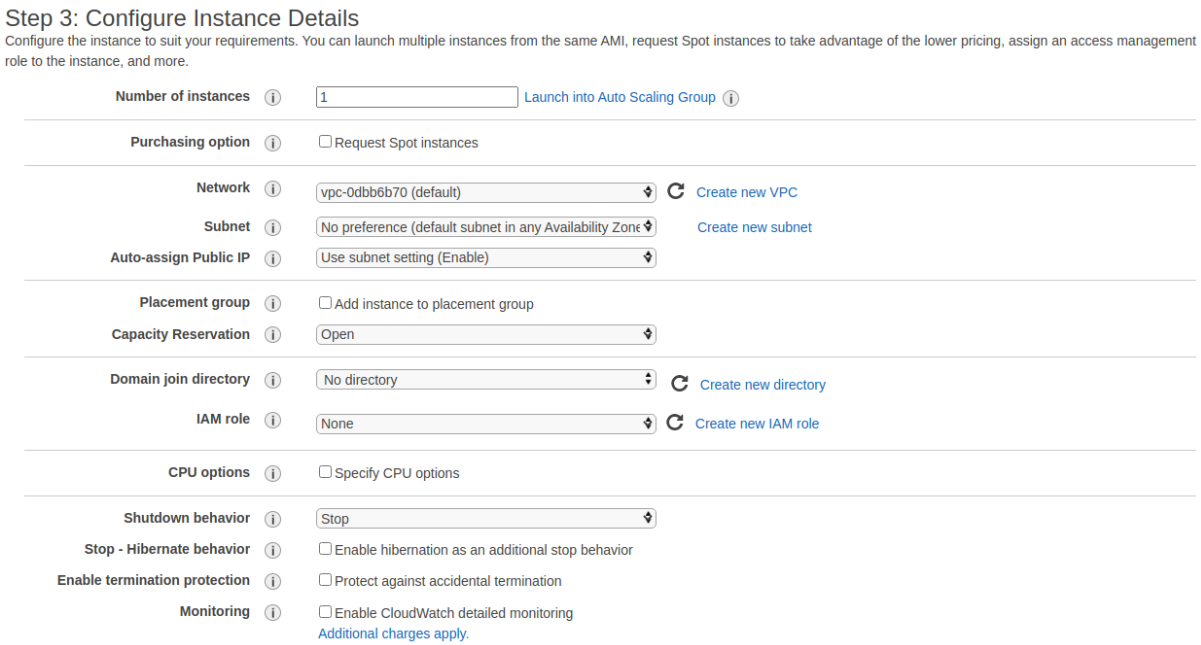


Рисунок 3.3 — Детальні налаштування за замовчуванням

Щодо постійної пам’яті використаємо стандартне запропоноване сервісом налаштування у 8 Гб SSD. Шифрування не будемо використовувати для постійної пам’яті(рис. 3.4).

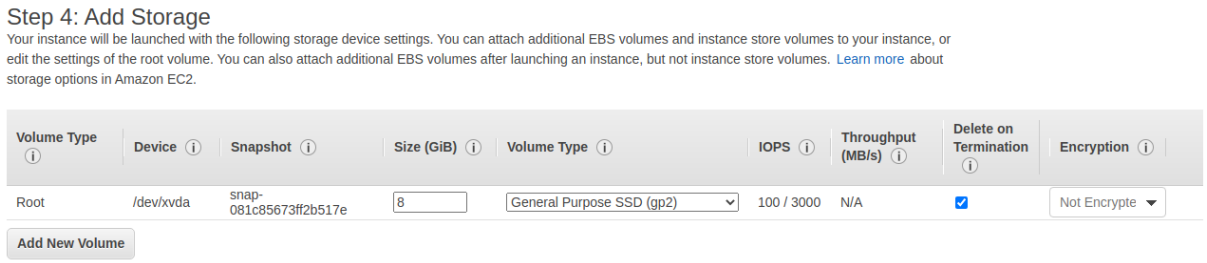


Рисунок 3.4 — Використання пам’яті на машині

Щодо брандмауеру поставимо дозвіл на вхідний трафік для будь якої ip-адреси на порти 22 для під’єднання використовуючи SSh та 8080 HTTP-протоколу(рис 3.5).

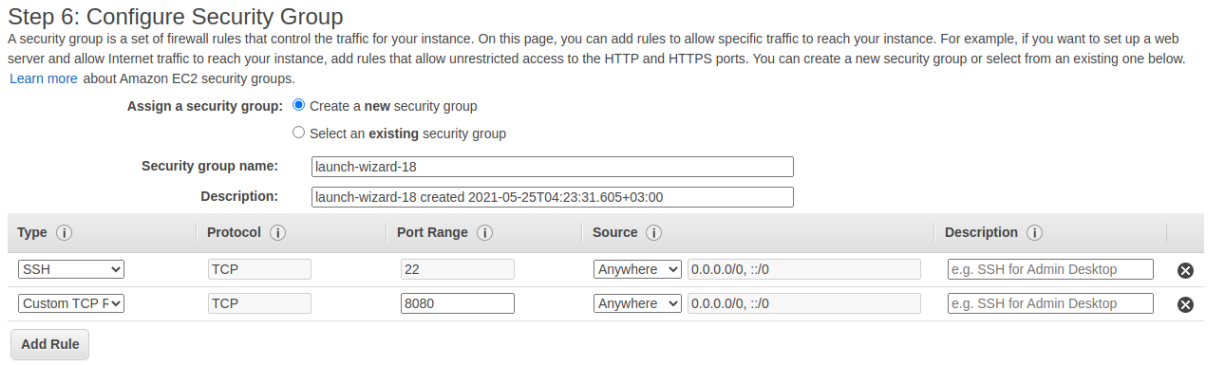


Рисунок 3.5 — Налаштування групи безпеки

Такі характеристики будуть використані для розгортання всіх програмних підчастин проекту(рис. 3.6).

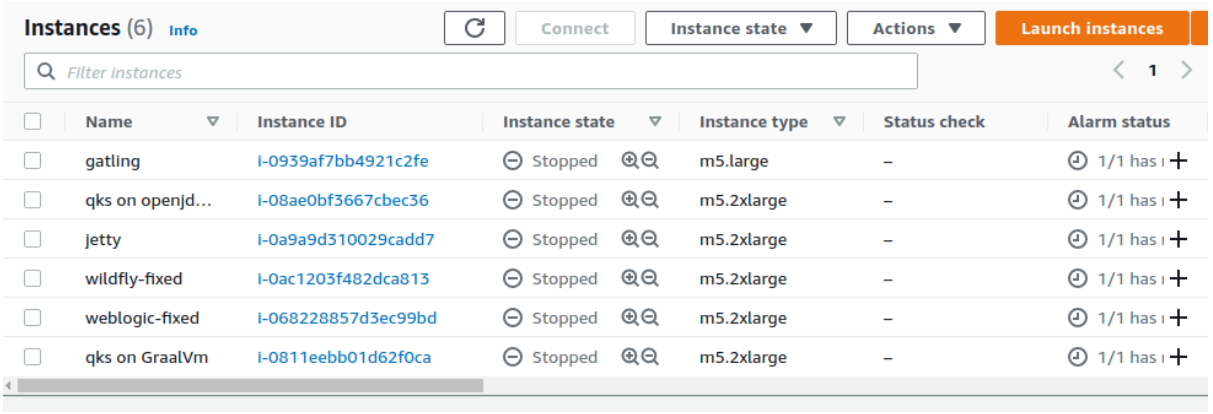


Рисунок 3.6 — Розгорнуті програмні продукти проекту

Як можемо побачити в роботі ведеться чітка специфікація основних характеристик використаних для розгортання програмних продуктів, адже в подальшому їх значення будуть впливати на роботу самого серверного програмного забезпечення і архітектури в цілому.

3.4 Розробка моделі серверного програмного забезпечення

Важливою частиною програмного продукту роботи є створення та налаштування серверного програмного забезпечення. Потрібно не тільки

встановити та розгорнути сервер, а і створити імітацію роботи бізнес моделі, звернення до бази даних, побудувати модель гексагональної архітектури в цілому. При побудові гексагональної архітектури на програмному рівні зробимо специфікацію її частин(рис. 3.7).

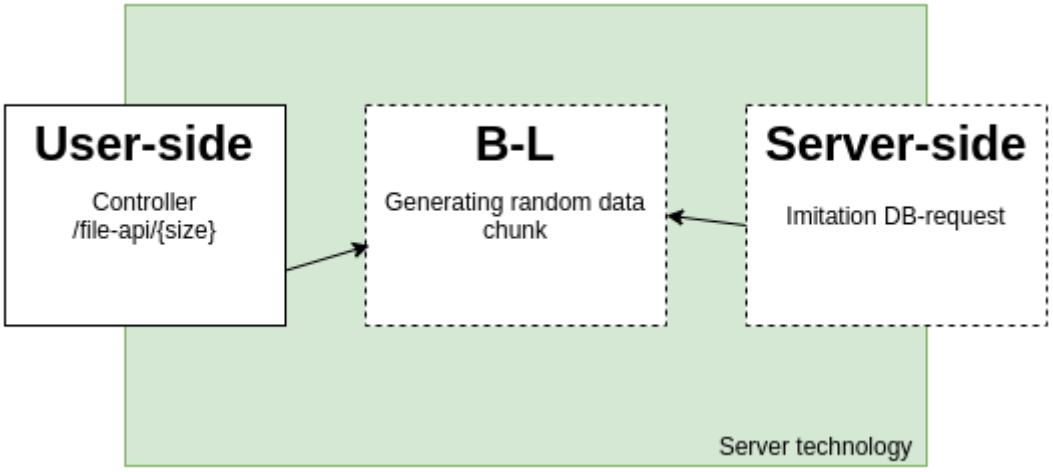


Рисунок 3.7 — Модель побудованої програми в гексагональній архітектурі

Як можемо побачити, побудована модель серверного додатку в гексагональній архітектурі складається з трьох частин:

1. Частина користувача: реалізовано контроллер, що надає дані результат роботи бізнес-логіки. Контролер - реальний механізм, що втілює механізм поширення даних по запиту. В реальних проектах цей механізм не буде змінено.

2. Частина бізнес-логіки: реалізована *імітація* виконання дій бізнес-логіки. З технічної точки зору ця частина робить перетворення, генерування чи інші дії реального життєвого процесу. В даному випадку генерується 1 кілобайт даних. В реальних проектах цей механізм буде змінено на відповідну реалізацію життєвого процесу.

3. Серверна частина: реалізована *імітація* звернення до бази даних. Серверна частина - технічний стек механізмів, що дозволяють відділити відповідальність за дії, що допомагають працювати з

бізнес-моделлю. В даному випадку реалізована затримка на 100 мілісекунд. Ця частина в реальних проектах буде змінена на механізми з'єднання з бд, отримання даних з інших джерел тощо.

Важливо зазначити, що головною ідеєю впровадження імітації було саме спрощення бізнес-логіки та серверної частини обслуговування бізнес-логіки роботи з даними саме для того, щоб навести фокус на серверні технології - технічній частині моделі гексагональної архітектури. Це саме та частина, яка реалізує механізми середи розробки та не залежить від впровадженої бізнес-логіки.

3.5 Алгоритм роботи серверного додатку та механізму тестування

Опишемо алгоритм роботи серверного додатку та програми тестування на продуктивність. Алгоритм роботи серверу починається з отримання користувацького запиту, затримки у 100 мілісекунд, генерації 1Кб даних, та повернення запиту(рис 3.8). Даний алгоритм реалізований вне залежності від того, яку серверну технологію було обрано. Це обумовлено необхідністю створення рівних умов роботи серверів для ефективного їх тестування на продуктивність та стрес.

Рисунок 3.8 — Алгоритм роботи серверного додатку

Алгоритм роботи механізму тестування складається з створення конфігурації протоколу HTTP, створення сценарію тестування - специфікація запиту(GET-запит по протоколу REST API, контекст шляху запиту, перевірка на успішність виконання запиту(статус-код 200)), та впровадження навантаження використовуючи раніше створені конфігурацію запиту та сценарій виконання тесту(рис. 3.9).

Рисунок 3.9 — Алгоритм роботи скрипту для тестування

3.6 Розробка скрипту тестування

Скрипти тестування на продуктивність були написані на мові програмування Scala, що дозволило швидко написати необхідні кейси тестування, використовуючи технологію Gatling. Технологія Gatling дає можливість проводити стрес-тестування, та тестування на продуктивність, генерувати метрики часових рядів роботи серверу після відпрацювання тесту. Розроблені Gatling скрипти специфіковані під створену серверну архітектуру. Частина конфігурування запиту по протоколу HTTP була налаштована на реалізацію дії звернення на IP-адресу серверу, та дозволяла приймати тільки звичайний текст на відповідь. Сценарій тестування складався з виконання GET-запиту по HTTP протоколу REST API на відповідний контекст серверу /1024, та перевірку статусу виконання вищезазначеного запиту. В подальшому відбувався механізм впровадження

навантаження в сценарій тестування. Загалом схема взаємодії елементів системи відбувався так як показано на рисунку 3.10.

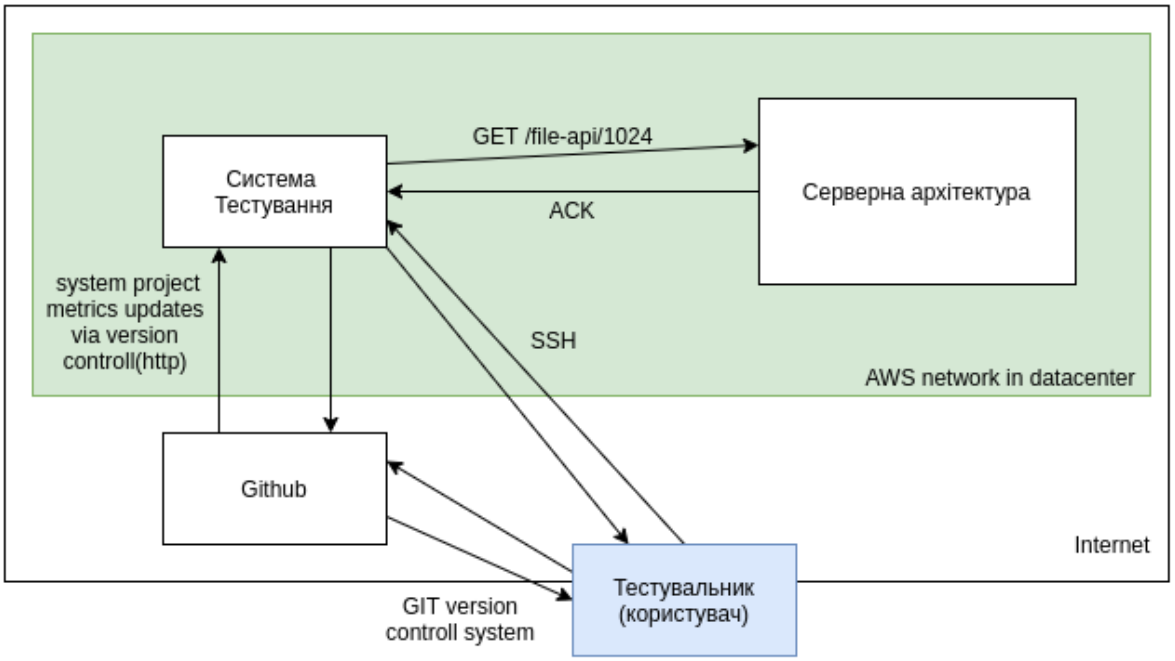


Рисунок 3.10 — Взаємодія елементів системи тестування

По вищезазначеній схемі відбувається тестування серверної архітектури. Тестувальник клонує репозиторій з Github та розгортає на машині апаратної архітектури програмне забезпечення для тестування. В скрипті вказується IP-адреса серверу, що необхідно протестувати на продуктивність та стресостійкість. Після процедури тестування результати завантажуються на Github. Найкраще для цього підійде створити нову гілку з позначенням системи що тестується. Після оновлення на локальному комп’ютері, розробник(користувач) може переглянути результати тестувань, відкривши в браузері згенеровану веб сторінку результатів тестувань та зробити відповідні висновки.

3.7 Результати виконання тестування

В ході роботи було протестовано такі серверні технології: Tomcat, Jetty, Wildfly, Quarkus на віртуальній машині GraalVM. Для проведення тестів на продуктивність оптимальним випадком було налаштувати конфігурацію навантаження реалізувати модель відкритого робочого навантаження, де контроль ведеться над рівнем прибуття користувачів. Механізм вводить користувачів із постійною швидкістю, визначеною в користувачах за секунду протягом тривалого часу. В даному випадку для отримання якісних метрик було визначено стартову швидкість збільшення - 100 користувачів за секунду. Цільова швидкість була визначена у розмірі 1000 користувачів за секунду. Збільшення відбувалося протягом 1 хвилини. Для серверу Jetty було отримано такі результати: всього протягом хвилини було зроблено 33000 запитів, з них успішних запитів складало 28566, неуспішних - 4434(Рис 3.11).

STATISTICS — Expand all groups | Collapse all groups

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Cnt/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	33000	28566	4434	13%	507.692	6	103	120	2172	3083	3828	326	650
Get a ki... of data	33000	28566	4434	13%	507.692	6	103	120	2172	3083	3828	326	650

Рисунок 3.11 — Таблиця результатів тестування Jetty

Бачимо, що 50% вибірки мають час відгуку у 103 мілісекунди, що допустимо, при тому, що імітація звернення до БД у кожного сервера складала 100 мілісекунд. Інші результати тестування позначено у вищезазначеній таблиці. Також варто зазначити графіки результатів активних користувачів та часу відгуку(рис. 3.12),

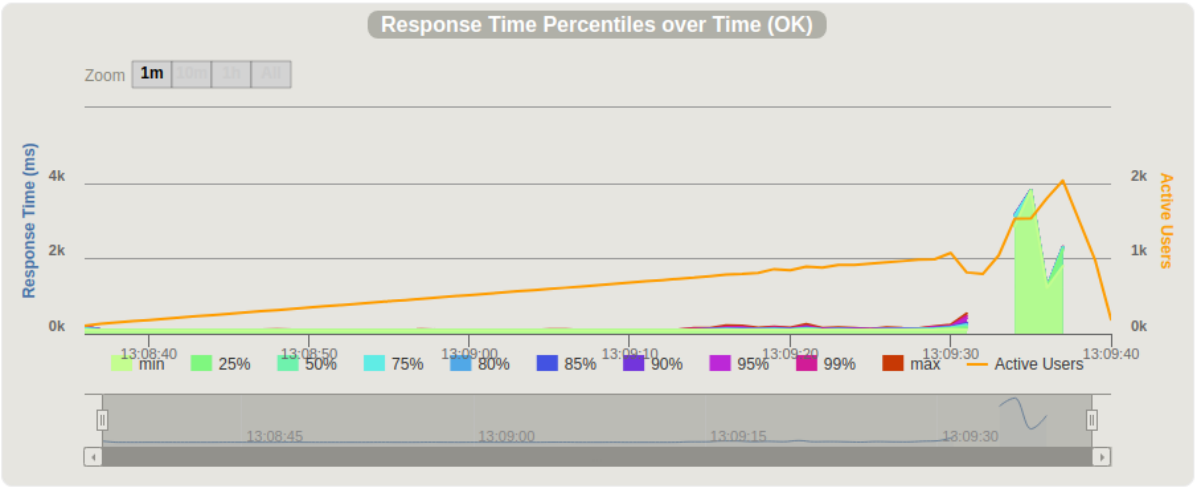


Рисунок 3.12 — Збільшення активних користувачів та часу відгуку

Активними користувачами дамо визначення запитів, які відправив програмний продукт тестування, але відповідь ще не прийшла. Бачимо, що кількість активних користувачів збільшилася до 1073 і далі спостерігаємо стрімке зменшення користувачів(рис 3.13).

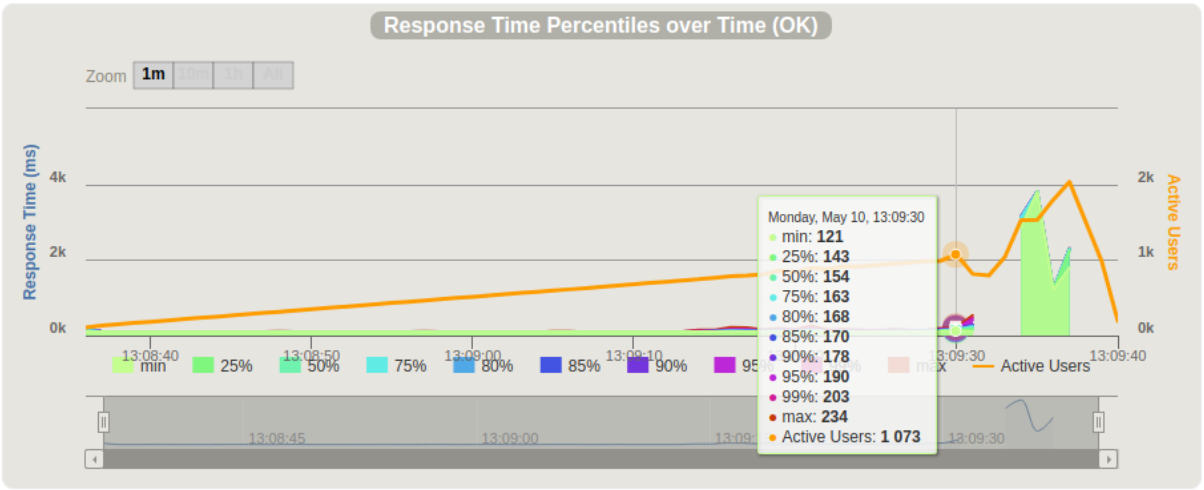


Рисунок 3.13 — Межа активних користувачів

Стрімке зменшення користувачів після описаної межі обумовлюється тим, що запити вважаються неуспішними(рис. 3.14), тобто такими, що тести не отримали відповідь на них.

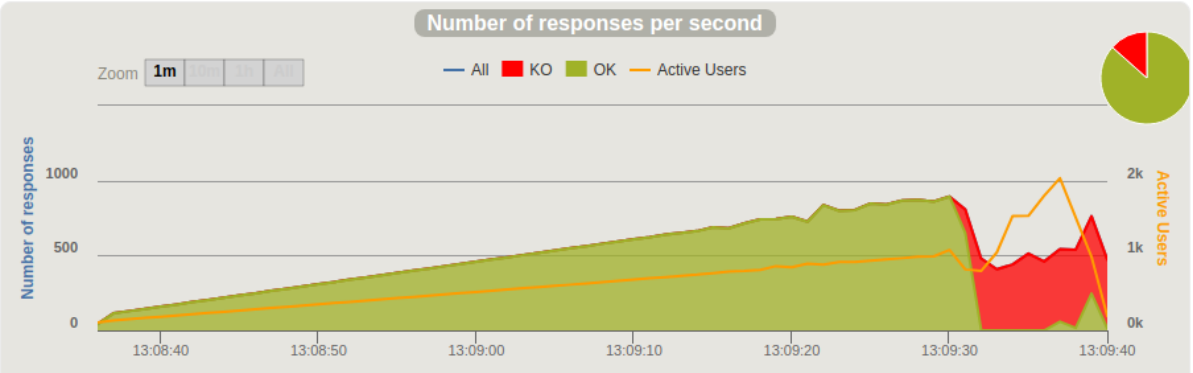


Рисунок 3.14 — Графік успішності запитів Jetty

В загальному випадку гістограма відповіді на користувацькі запити по часу виглядає дуже оптимістично(рис. 3.15). У стабільному режимі обробка запиту сервером буде складати 2 мілісекунди.

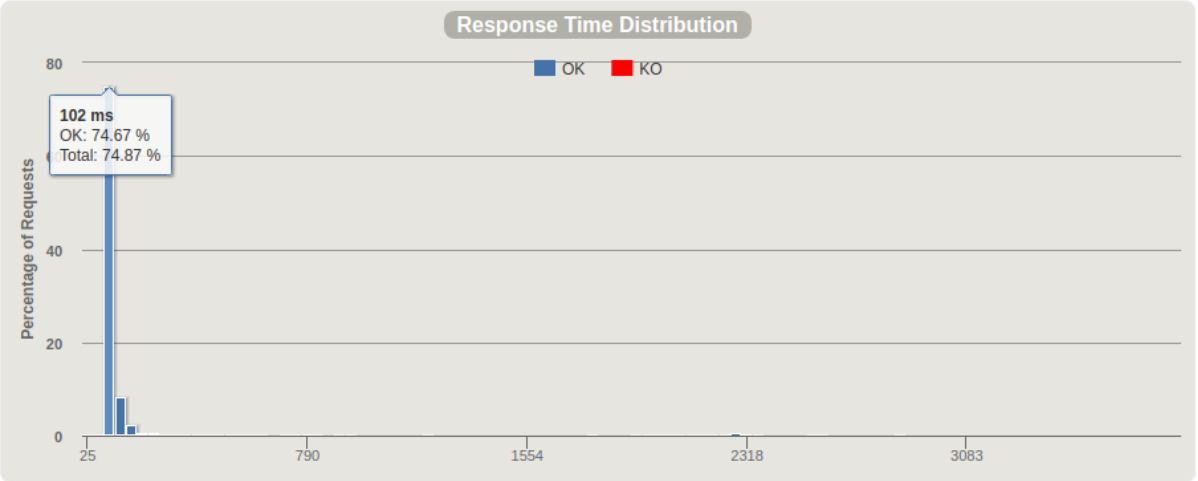


Рисунок 3.15 — Гістограма часу відповіді для Jetty

Розглянемо серверну технологію на технології Tomcat. Загальні рельтати при тому ж збільшенні навантаження отримаємо майже такі ж результати(рис. 3.16).

STATISTICS — Expand all groups | Collapse all groups

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Cnt/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	33000	28585	4415	13%	507.692	6	103	120	1691	3561	4393	306	644
Get a ki... of data	33000	28585	4415	13%	507.692	6	103	121	1691	3561	4393	306	644

Рисунок 3.16 — Результати роботи серверу Tomcat

Можемо побачити, що різниця кількості успішних результатів між двома розглянутими технологіями на рівні похибки. В основній масі кількість успішних запитів складає 87% в обох розглянутих випадках. Досліджуючи верхню межу запитів можемо побачити що вона складає 968 активних запитів(рис. 3.17). Ця межа нижча за попередній результат.

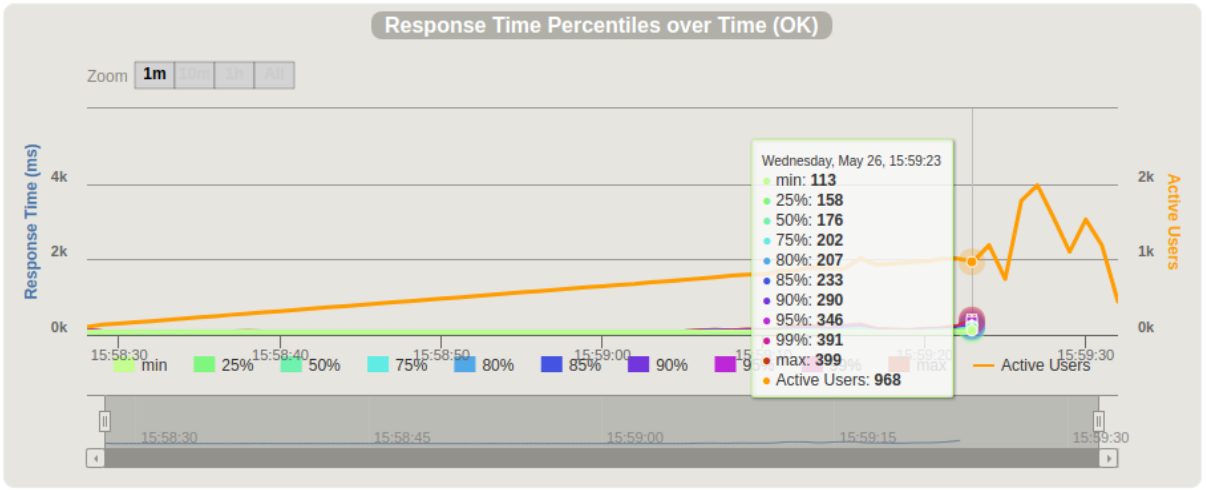


Рисунок 3.17 — Верхня межа запитів Tomcat

Щодо гістограми відповідей(рис. 3.18) можемо побачити, що робота сервера складає 16 мілісекунд. Це значно менше ніж в попередньому випадку. Але доля відповідей з таким результатом складає 79.06%. Що більш стабільніше за попередній результат.

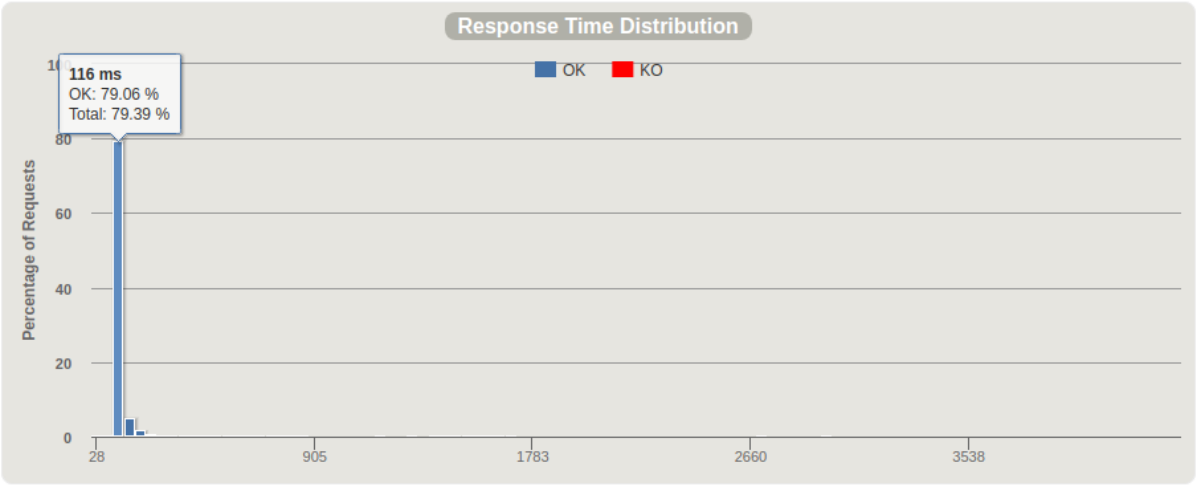


Рисунок 3.18 — Гістограма відповідей Tomcat

В основному випадку статусний графік запитів виглядає так само як і в попередньому випадку(рис. 3.19).

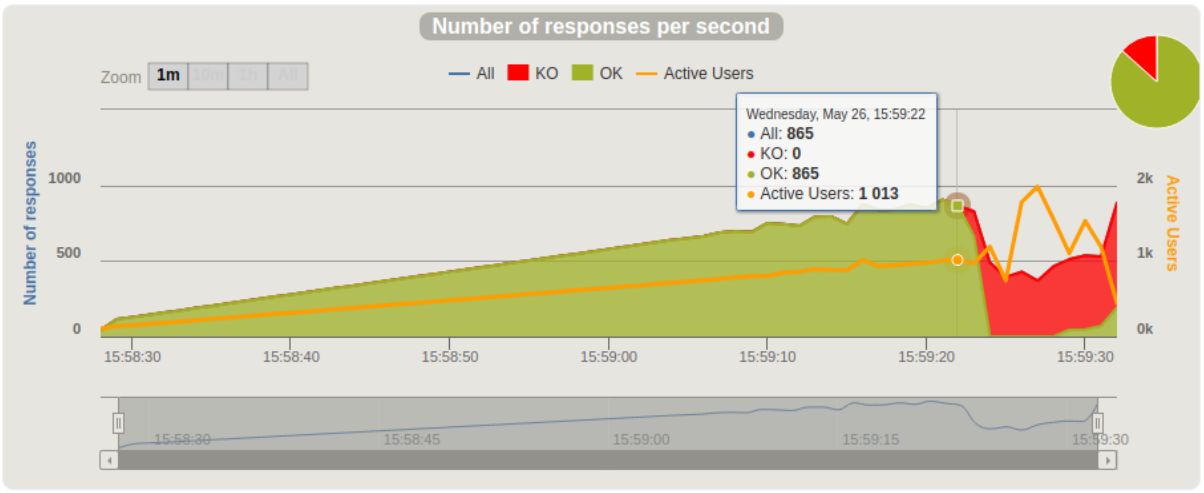


Рисунок 3.19 — Графік успішності запитів Tomcat

Розглядаючи серверну технологію Wildfly можемо побачити схожі результати(рис. 3.20). Кількість успішних запитів складає 87%. Гістограма відповідей на запити показує те, що в основній масі запитів 73.96% відповідь складає 113 мілісекунд(рис. 3.20).

STATISTICS — Expand all groups | Collapse all groups

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Cnt/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	33000	28692	4308	13%	507.692	6	104	137	1686	3124	4288	307	601
Get a ki... of data	33000	28692	4308	13%	507.692	6	104	137	1686	3124	4288	307	601

Рисунок 3.20 — Рузультати тесту для Wildfly

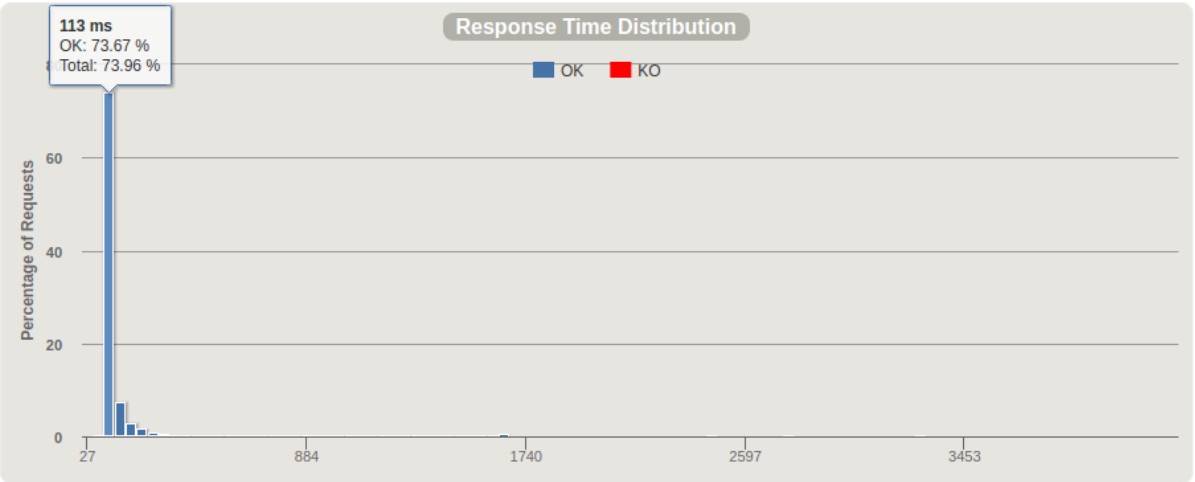


Рисунок 3.20 — Гістограма відповідей на запит Wildfly

Розглянувши графік часу відповіді на запит можемо побачити, що порогова кількість користувачів складає 1050 користувачів(рис. 3.21, рис. 3.22).

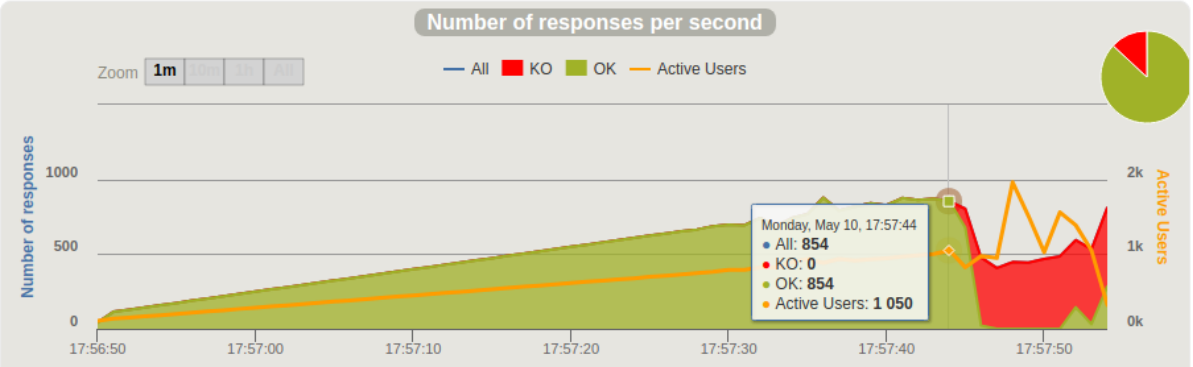


Рисунок 3.21 — Порогова кількість користувачів Wildfly

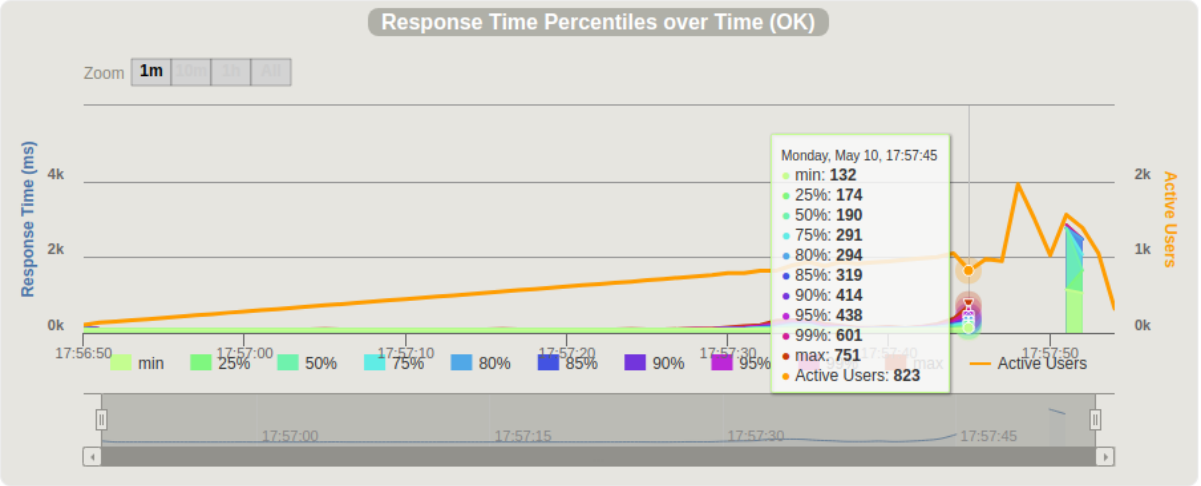


Рис. 3.22 — Графік часу відгуку Wildfly

Розглядаючи нову технологію Quarkus на віртуальній машині GraalVM, результати починають кардинально змінитися в області часу відповіді. На рівні обслуговування запитів результати залишаються такими ж. 13% - відсоток неуспішних запитів(рис. 3.23).

STATISTICS — Expand all groups | Collapse all groups

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Cnt/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	33000	28838	4162	13%	500	14	299	1872	4069	4645	5098	1099	1348
Get a ki... of data	33000	28838	4162	13%	500	14	300	1872	4070	4645	5098	1099	1348

Рисунок 3.23 — Результати роботи Quarkus на GraalVM

Щодо графіку часу відповіді можемо побачити, що ця технологія передує усім вище описаним, але лише до певної межі навантаження(рис. 3.24). Щодо порогу, після якого затримка відповіді починає експоненційно зростати складає приблизно 700 активних користувачів - запитів(рис. 3.25).

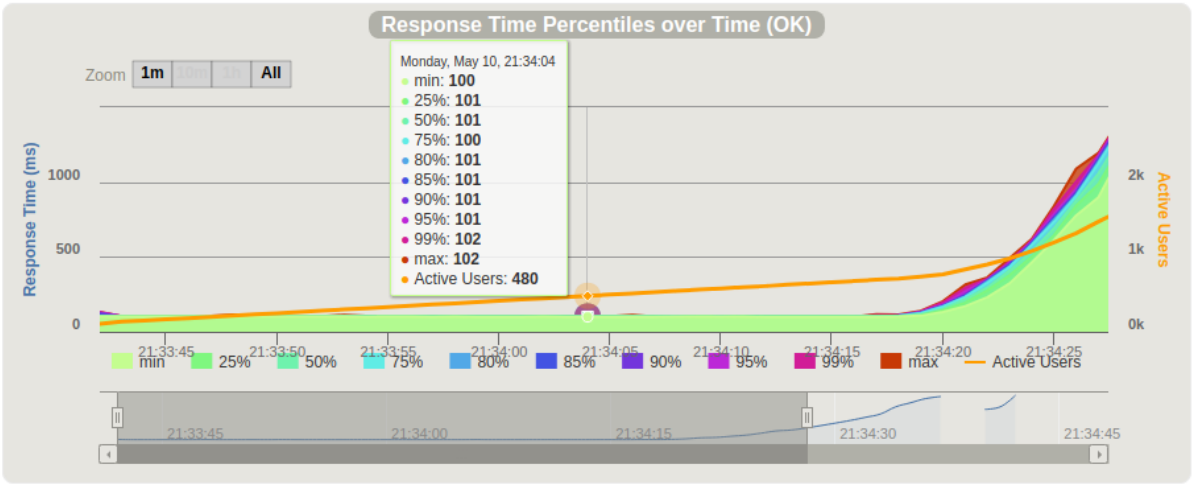


Рисунок 3.24 — Середня швидкість відповіді Quarkus на GraalVM

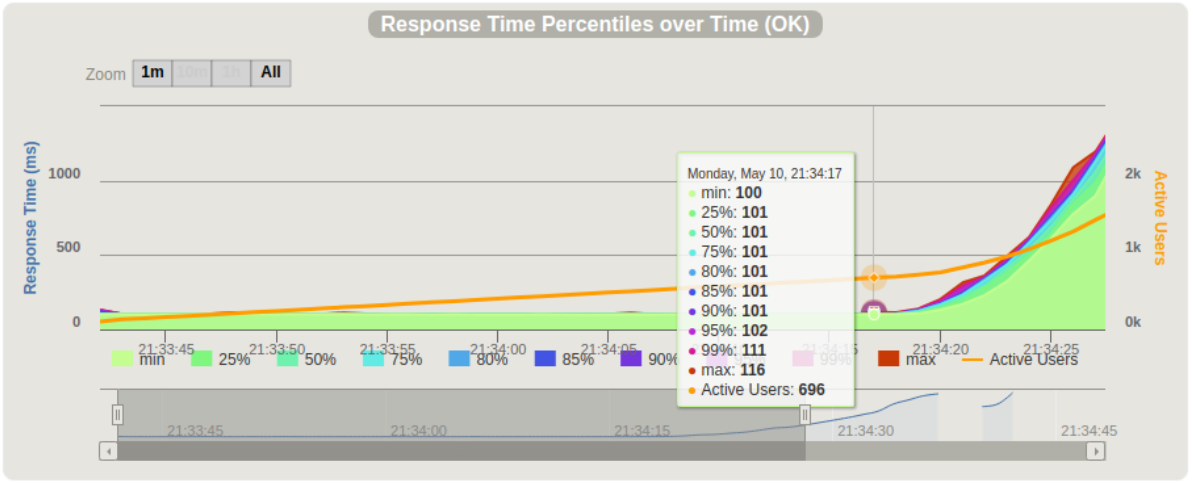


Рисунок 3.25 — Порогова кількість активних користувачів при швидкій роботі Quarkus на GraalVM

Після межі у 700 користувачів час відгуку починає зростати експоненційно до межі відмови у 3400 користувачів(рис. 3.26).

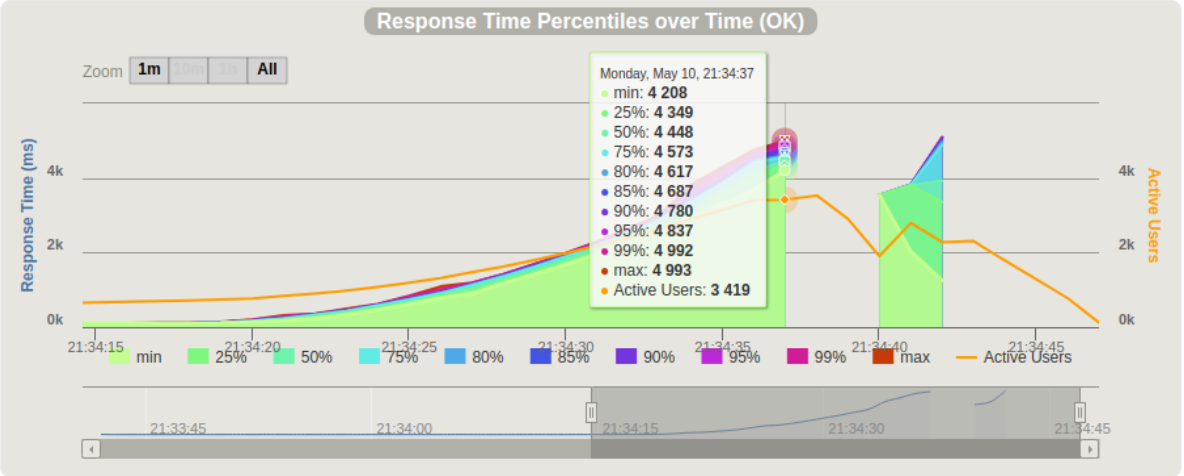


Рисунок 3.26 — Порогова межа роботи серверу Quarkus на GraalVM

Навіть після межі відмови певна кількість запитів продовжується оброблятися сервером(рис. 3.27).

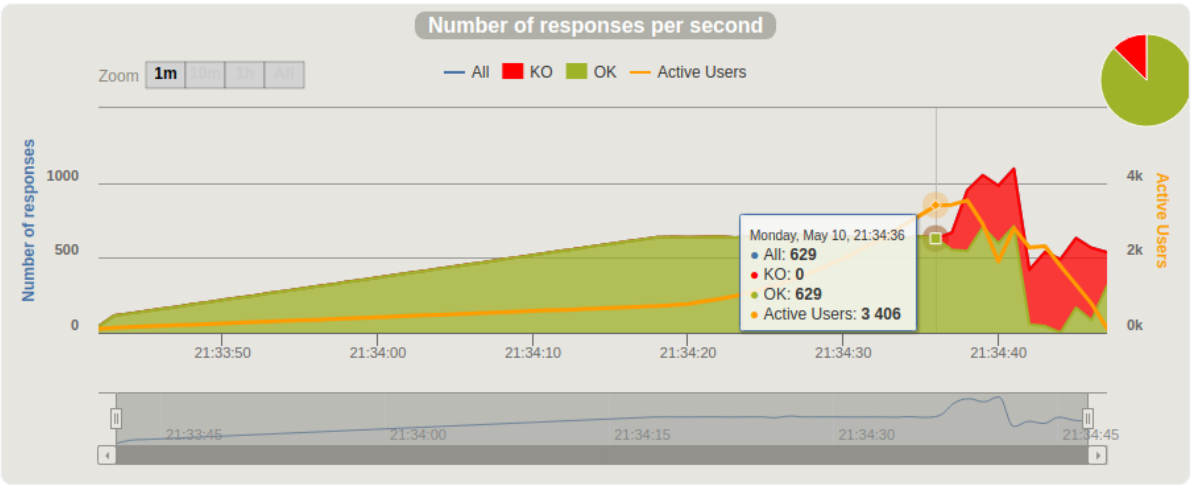


Рисунок 3.27 — Загальна робота серверу Quarkus на GraalVM

Вищезазначені результати можемо узагальнити певною моделлю лінійної залежності, що описує роботу серверних технологій в межах стабільного навантаження. Процес проведення стрес тестування можемо описувати моделлю використовуючи структурний підхід який залежить від двох факторів: N- кількість успішно виконаних запитів, T_{test} - об’єм часу

відгуку на проміжку тестування на продуктивність в режимі стабільної роботи серверу. Маємо формулу 3.1:

$$R = f(N, T_{test}) \tag{3.1}$$

Функцію часу відгуку можемо описати за допомогою лінійної функції к коефіцієнтами t_0- початковий час відгуку в експерименті, a_1- коефіцієнт який будемо визначати в ході проведення експерименту - тестування на продуктивність. В даному випадку спростований варіант обчислення буде виглядати таким чином:

$$y = f(t) = t_0 + a_1 t \tag{3.2}$$

Для оцінки результату проходження тестування на продуктивність можемо порахувати величину результату тестування T_{test}, що описує загальний результат роботи тесту:

$$T_{test} = \int_{t_1}^{t_2} f(t)\, dt = (a_0 t + \frac{1}{2} a_1 t^2)\Big|_{t_1}^{t_2} \tag{3.3}$$

Таким чином ми отримуємо об’єм витраченого часу на запити.

Далі постає необхідність визначити певний коефіцієнт, що дозволяє порівнювати між собою результати тестів. Кількісна величина висновку повинна включати в себе об’єм витраченого часу та кількість успішних запитів. Для об’єктивного порівняння тести проводитимуться на одному проміжку часу $[t_1;\ t_2]$ із однаковим навантаженням N_{total}. В даній роботі ці величини фіксовані: витрачений час тестування складає 60000 мілісекунд

та загальне навантаження у 33000 активних користувачів - запитів. Нехай величина обчислення результату буде мати такий вигляд:

$$R = \frac{1 - \left(\frac{N_{total} - N_{success}}{N_{total}}\right)}{V_{time}} \tag{3.4}$$

Можемо побачити, що шукана величина буде залежати від відносної кількості успішних запитів та об’єму витраченого часу. При збільшенні об’єму часу коефіцієнт буде показувати гірший результат. Так само з відносною кількістю успішних запитів. Чим більше успішних запитів отримуємо, тим краще буде коефіцієнт. Таким чином ми визначили характеристику результатів тестування, що дозволяє порівнювати між собою загальний об’єм пройдених тестів.

Порахувавши кожен коефіцієнт отримали такі результати:

$R_{jetty} = 1.56 \cdot 10^{-7}$, $R_{tomcat} = 1.43 \cdot 10^{-7}$, $R_{widlfly} = 1.28 \cdot 10^{-7}$, $R_{qks} = 1.57 \cdot 10$

Розглядаючи сторону стрес-тестування(підйому навантаження 100 запитів-користувачів за секунду до 5000 протягом хвилини) можемо зробити висновки, що серверне програмне забезпечення на технології Tomcat буде робити “ривки” з успішними відповідями 4 рази(рис. 3.28).

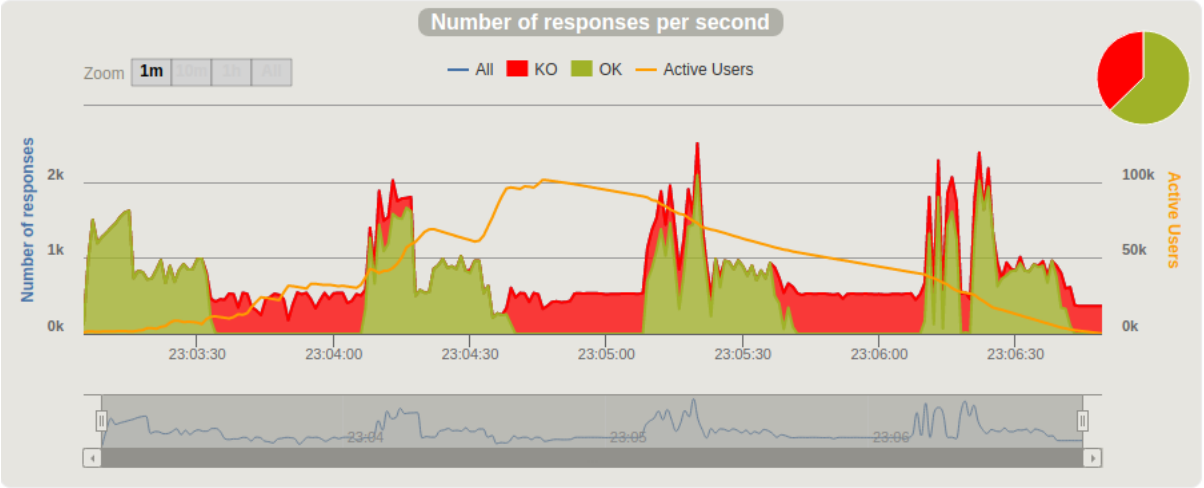


Рисунок 3.28 — Робота серверу в умовах стресу Tomcat

Щодо Jetty та WildFly ми можемо побачити, що буде присутня менша кількість разів, коли сервер обробляє певну стресову кількість запитів за 1 хвилину(рис. 3.29 та рис. 3.30).

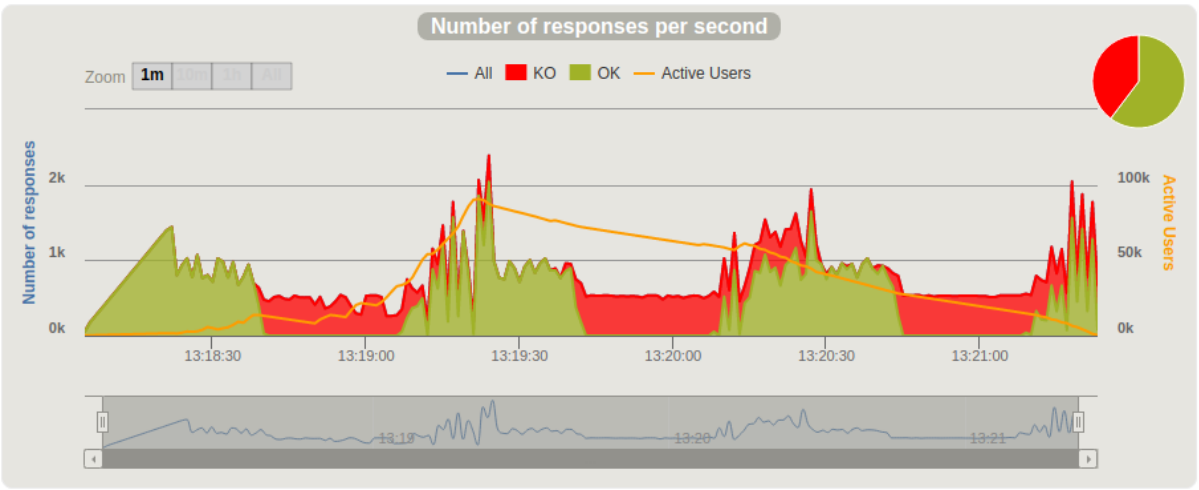


Рисунок 3.29 — Робота серверу Jetty в умовах стресу

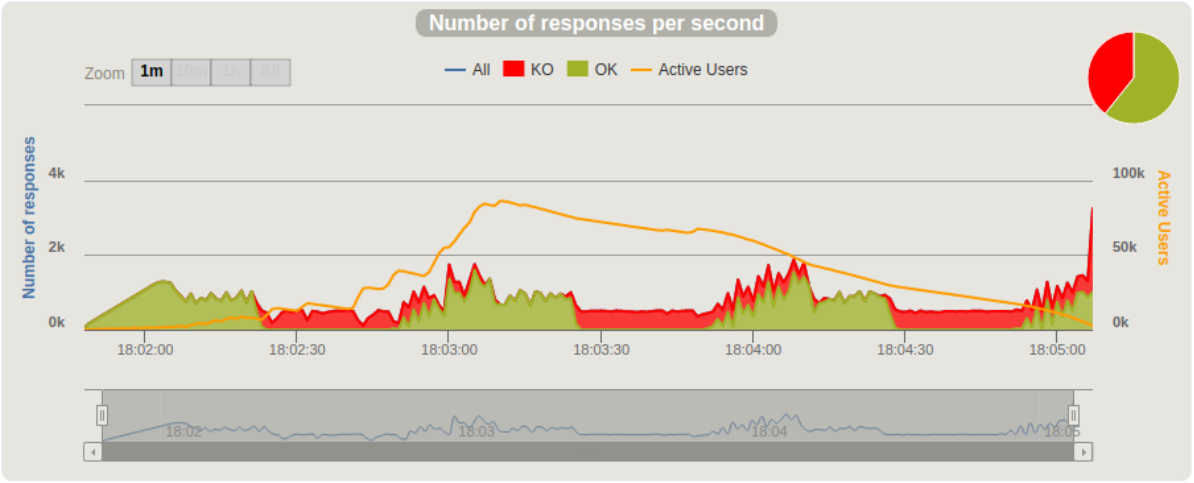


Рисунок 3.30 — Робота Wildfly в умовах стресу

Беручи до уваги Технологія Quarkus на віртуальній машині GraalVM, можемо зробити результати, що циклічні проміжки роботи серверу в умовах стресу довші за попередні а висота “хвилі роботи” більша, що показує, що Quarkus на GraalVM пристосований більше до роботи в умовах стрімкої кількості запитів ніж усі попередні.

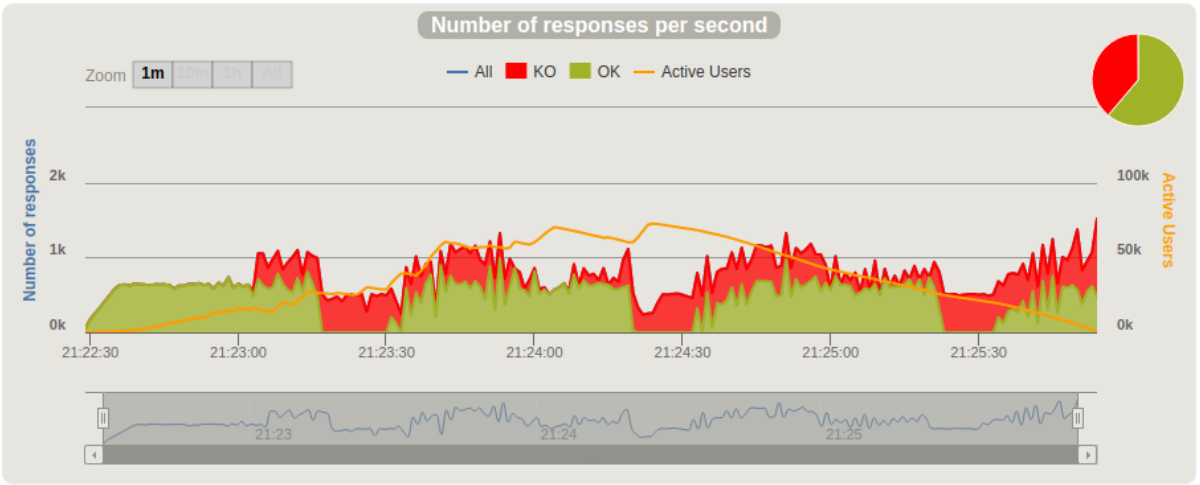


Рисунок 3.31 — Робота Quarkus на GraalVM в умовах стресу

3.8 Висновки

В ході виконання роботи з роботою було продемонстровано на практиці можливість проведення стрес-тестування та тестування на

продуктивність серверних технологій. Було використано різні технології по популярності, даті створення та технології впровадження веб-додатків на сервер. Тестування проводилося із залученням такої технології як Gatling. Серверні технології, над якими проводили тестування: Tomcat, Jetty, WildFly на віртуальній машині OpenJDK, Quarkus був впроваджений на віртуальній машині GraalVm. В ході виконання роботи було отримано навички не тільки створення механізмів тестування а і механізмів розгортання, було досліджено частинну роботу серверних технологій. На прикладі роботи серверного продукту стало можливим проаналізувати роботу серверних технологій. Можемо зробити, що при стабільній роботі та в умовах стресу краще справляється Quarkus на віртуальній машині GraalVm та Jetty на OpenJDK. Висновки зроблено по часу відгуку серверу. Таким чином, в роботі було наведено фокус на серверні технології, технічну частину серверної архітектури, інші частини продукту були нівельовані. Даний програмний продукт можна використовувати при роботі реальних веб-сервісів для проведення аналітики стрес тестування та тестування на продуктивність.

РОЗДІЛ 4 ФУНКЦІОНАЛЬНО-ВАРТІСНИЙ АНАЛІЗ ПРОГРАМНОГО ПРОДУКТУ

4.1 Постановка задачі проектування

Спроектувати програмний продукт для проведення стрес-тестування та тестування на продуктивність серверного програмного забезпечення, для того, щоб тестувальники чи розробники (користувачі) могли оцінювати працездатність серверних веб-додатків. Завдання в першу чергу створити скрипт, що дозволить користувачам просто ввести кількісну характеристику навантаження, запустити за допомогою менеджеру проекту Maven та отримати результати тестування в інтерактивному вигляді. Для тестування було використано технологію Gatling. Для розміщення тестувального програмного забезпечення було використано AWS EC2.

Технічні вимоги до програмного продукту є наступними:

- функціонування на серверній архітектурі із стандартним набором компонент
- швидкість впровадження навантаження
- зручний спосіб отримання результатів
- мінімальні витрати на впровадження програмного продукту

4.2 Обґрунтування функцій програмного продукту

Головна функція F_0 - розробка програмного продукту, який впроваджує тести на продуктивність для серверного ПЗ та зручно відображає результат виконання цих тестів.

F_1- вибір платформи апаратної архітектури розташування

F_2- вибір віртуальної машини

F_3- вибір мови програмування

Кожна з цих функцій має декілька варіантів реалізації:

Функція F_1:

а) AWS

б) Google Cloud

Функція F_2:

а) OpenJDK

б) GraalVM

Функція F_3:

а) Java

б) Scala

Варіанти реалізації основних функцій наведені у морфологічній карті системи (рис. 4.1).

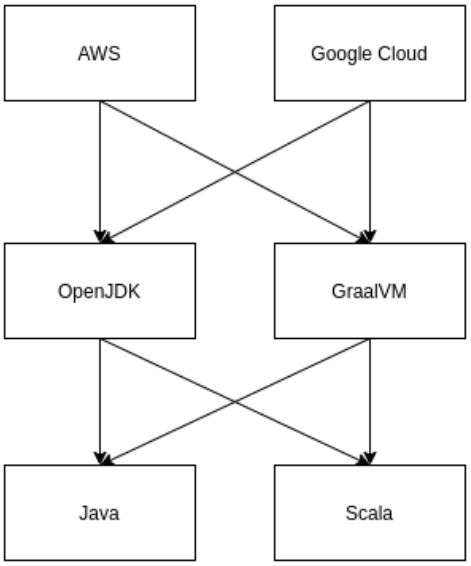


Рисунок 4.1 — Морфологічна картка

Морфологічна карта відображає множину всіх можливих варіанти основних функцій.

Таблиця 4.1 — Позитивно-негативна матриця

Функції	Варіанти реалізації	Переваги	Недоліки
F1	А	Більша кількість можливих варіантів ОС	Висока складність розгортання
	Б	Простота користування	Низька кількість варіантів вибору апаратних архітектур
F2	А	Стабільність у використанні	Споживання великої кількості оперативної пам’яті
	Б	Оптимізована робота з пам’яттю	Можливі присутні помилки у роботі
F3	А	Простота використання, велика кількість вбудованих бібліотек	Низька швидкість розробки
	Б	Висока швидкість розробки	Висока складність структур, що використовуються при розробці

Для характеристики прототипу програмного додатку використовуємо параметри X1 – X5. На основі даних, що представлені у літературі, визначаємо мінімальні, середні отримуванні та максимально допустимі значення(табл. 4.2)

Таблиця 4.2 — Таблиця оцінки економічної середи

Найменування параметру	Позначення параметру	Значення параметру		
		Мінімальне	Середнє	Макимальне
Час розробки, людина*год	X1	285	384	542
Час роботи алгоритму, мс	X2	101	230	3450
Рекомендована частота процесору, ГГц	X3	2.2	2.8	4.2
Час обробки результату, мс	X4	25	113	248
Рекомендована швидкість запису на диск, МБ/с	X5	0	32	64

На рисунках 4.2-4.4 зображено у графічному представлені параметри функцій.

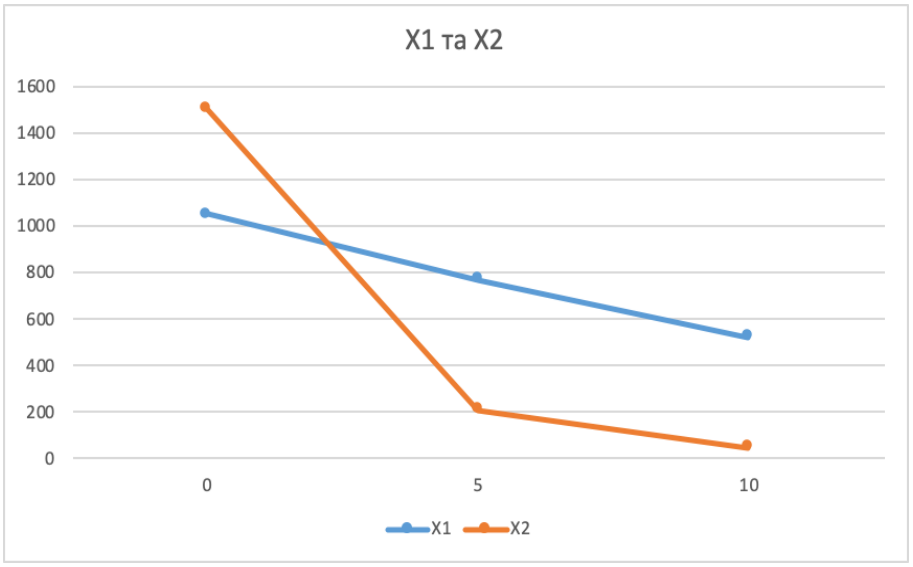


Рисунок 4.2 — Значення параметрів X1, X2

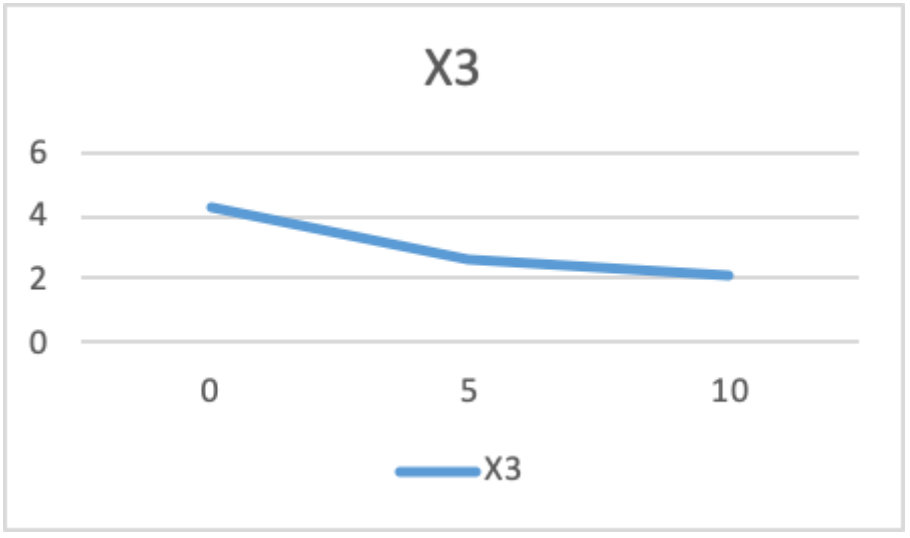


Рисунок 4.3 — Значення параметру X3

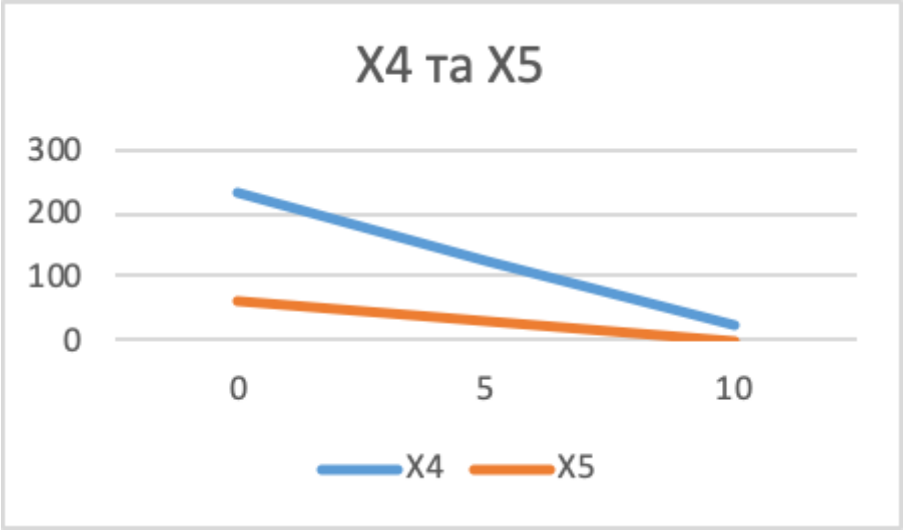


Рисунок 4.4 — Значення параметрів X4, X5

Вагомість параметрів оцінюється за допомогою методів попарного зрівняння. Ранги варіюються від 1 до 5. Результати наведені в табл. 4.3-4.4

Таблиця 4.3 — Результат оцінки параметрів

Параметр	Ранг параметру по оцінці експерта							Сума рангів, R_i	Відхилення Δ_i	Квадрат відхилення, $(\Delta_i)^2$
	1	2	3	4	5	6	7			
X1	2	2	1	2	2	2	2	13	-8	64
X2	1	1	2	1	3	1	1	10	-11	121
X3	4	3	4	4	5	3	3	26	5	25
X4	3	4	3	3	1	4	4	22	1	1
X5	5	5	5	5	4	5	5	34	13	169
Разом	15	15	15	15	15	15	15	105	0	380

Таблиця 4.4 — Попарне зрівняння параметрів

Параметри	Експерти							Кінцева оцінка	Числове значення
	1	**2**	**3**	**4**	**5**	**6**	**7**		
X1 та X2	<	<	>	<	>	>	<	<	0.5
X1 та X3	<	>	<	<	<	<	>	<	0.5
X1 та X4	>	<	>	<	<	>	>	>	1.5
X1 та X5	>	>	>	<	>	<	>	>	1.5
X2 та X3	<	<	>	<	<	<	>	<	0.5
X2 та X4	>	>	>	>	<	>	>	>	1.5
X2 та X5	>	<	<	>	<	<	<	<	0.5
X3 та X4	<	>	>	>	<	>	>	>	1.5
X3 та X5	>	>	>	>	<	>	>	>	1.5
X4 та X5	>	>	>	>	>	>	>	>	1.5

Визначемо коефіцієнт конкординації:

Так як коефіцієнт конкординації більше нормативного, результати вважають достовірними.

Розрахунок вагомості параметрів наведено в табл. 4.5

Таблиця 4.5 — Розрахунок вагомості параметрів

Параметри	Параметри xj					Перший крок		Другий крок	
	X1	X2	X3	X4	X5	b_i	$К_{ві}$	b_i	$К_{ві}$
X1	1	0,5	0,5	1,5	1,5	5	0,21	22.5	0,19
X2	1,5	1	0,5	1,5	0,5	5	0,21	23.5	0,2
X3	1,5	1,5	1	1,5	1,5	6	0,25	33	0,28
X4	0,5	0,5	0,5	1	1,5	4	0,165	18	0,15
X5	0,5	1,5	0,5	0,5	1	4	0,165	19	0,18
Загалом:						24	1	116	1

Враховуючи дані з порівнянь варіантів реалізацій функцій можна виключити з реалізацій функцій наступні варіанти: F1(б), F3(б)

Залишаються наступні варіанти:

1. F1(а)=>F2(а)=>F3(а)
2. F1(а)=>F2(б)=>F3(а)
3. F1(а)=>F2(а)=>F3(а)
4. F1(а)=>F2(б)=>F3(а)

Таблиця 4.6 — Розрахунок коефіцієнтів якості

Основна функція	Варіант реалізації	Абсолютне значення параметру	Бальна оцінка параметру	Коефіцієнт вагомості параметру	Коефіцієнт якості
F1(x1)	а)	768	5	0,19	0,95
F2(x2)	а)	1500	0	0,2	0
	б)	210	5	0.2	1
F3(x3)	а)	2.6	5	0,28	1,4

Обрахуємо коефіцієнти якості кожного з варіантів розробки: -

$$K_{я1} = 0,95 + 0 + 1,4 + 1,5 + 0,9 = 4,75$$

$$K_{я2} = 0,95 + 1 + 1,4 + 0,75 + 0,9 = 5$$

Оскільки варіант 2 має найбільший коефіцієнт якості, він є найкращим.

4.3 Економічний аналіз варіантів розробки

Для оцінки трудомісткості розробки спочатку проведемо розрахунок трудомісткості. Усі варіанти мають наступні основні завдання:

1) Завантаження даних з фізичного носія
2) Використання дерев прийняття рішень
3) Запис даних у БД та вивід користувачу

Також кожний з варіантів має два додаткових завдання, які є реалізаціями розгалужених варіантів розробки незалежного модуля. Далі наведено варіанти додаткових завдань(два завдання, які мають номери 4 в реалізаціях та два завдання, які мають номери 5 в реалізаціях)

2.1) Використання акумулюючих ознак

2.2) Використання окремих ознак

3.1) Використання основної ознаки - зарахування

3.2) Використання основної ознаки - незарахування

В варіанті 1 присутні наступні додаткові завдання під номерами 2.1 та 3.1

В варіанті 2 присутні наступні додаткові завдання під номерами 2.2 та 3.2

В варіанті 3 присутні наступні додаткові завдання під номерами 2.1 та 3.2

В варіанті 4 присутні наступні додаткові завдання під номерами 2.2 та 3.1

За ступенем новизни до групи Б відноситься задання 5.1, до групи В відносяться завдання 1, 3, до групи Г завдання 2.1, 2.2, 4, 5.2.

За складністю алгоритмів до групи 1 відносяться завдання 1, 4, 2.1 до групи 2 відноситься завдання 2.2, 5.1, до групи 3 відноситься завдання 3, 4.2

Спираючись на норми розрахункового часу визначимо трудомісткість. Вона складає для першого завдання $T_р$=43 людино-днів. Поправочний коефіцієнт складає $К_п$=1,35 (нормативно-довідкова інформація). Оскільки під час виконання даного завдання використовуються новостворенні модулі, врахуємо це за допомогою коефіцієнта $К_{ст}$ = 0,6. Коефіцієнти $К_м$ і $К_{ст.п}$, які враховують відповідно програмування на мові низького рівня та розробку стандартного програмного забезпечення, для всіх семи завдань дорівнюють 1.

Повна трудомісткість першого завдання (складність – 1, новизна – В):

$$T_1 = 43 * 1,35 * 0,6 = 34,83$$

Аналогічно для завдання 4, 2.1 (складність – 1, новизна – Г): –

$T_р$= 64; $К_п$ = 1,08; $К_{ст}$ = 0,8; $T_2 = 64 * 1,08 * 0,8 = 55,3$

Аналогічно для завдання 3 (складність – 3, новизна – В):

$T_р$= 12; $К_п$ = 0,6; $К_{ст}$ = 0,6; $T_3 = 12 * 0,6 * 0,6 = 4,32$

Аналогічно для завдання 2.2 (складність – 3, новизна – Г)

$T_р$= 8; $К_п$ = 0,36; $К_{ст}$ = 0,6; $T_4 = 8 * 0,36 * 0,6 = 1,73$

Аналогічно для завдання 5.1 (складність – 2, новизна – Б):

$T_р$= 27; $К_п$ = 1,08; $К_{ст}$ = 0,8; $T_5 = 27 * 1,08 * 0,8 = 23,33$

Визначимо повну трудомісткість варіантів(людино-днів):

$$T_1 = T_3 = 34,83 + 55,3 + 4,32 + 55,3 + 23,33 = 173,08$$

$$T_2 = T_4 = 34,83 + 1,73 + 4,32 + 55,3 + 23,33 = 119,51$$

Найбільш трудомісткими завданнями є 2.1 та 4, найбільш трудомісткий варіант - 1

Далі вважається, що робочий день складає 8 годин, в тижні п’ять робочих днів. В розробці бере участь один програміст з окладом 13500 грн та тестувальник з окладом 10000 грн. Визначимо середню заробітну плату за годину:

$$С_ч = \frac{13500+10000}{2*22*8} = 66,76$$

Тоді заробітна плата для кожного з варіантів реалізації(грн):

1) $С_{ЗП} = 66,76 * 8 * 173,08 = 92438,57$
2) $С_{ЗП} = 66,76 * 8 * 119,51 = 63827,90$

Відрахування на соціальне страхування(22%)(грн):

1) $С_{ВІД} = 92438,57 * 0,22 = 20336,49$
2) $С_{ВІД} = 63827,90 * 0,22 = 14042,14$

Далі розрахуємо витрати на оплату однієї машино-години. Враховуючи, що вона обслуговує одного спеціаліста з окладом 13500 грн

та одного з окладом 10000 грн з коефіцієнтом зайнятості 0,6, то для двох машин отримаємо

$$С_г = 12 * 13500 * 0,6 + 12 * 10000 * 0,6 = 169200 \text{ грн}$$

Враховуючи додаткову заробітну плату (40%)

$$С_{ЗП} = 169200 * (1 + 0,4) = 270720$$

Відрахування на соціальне страхування 22%

$$С_{ВІД} = 270720 * 0,22 = 59558,4$$

Розрахуємо амортизаційні підрахунки (амортизація 25%, вартість ЕОМ 30000 грн)

$$С_А = К_{ТМ} * К_А * Ц_{ПР} = 1,15 * 0,25 * 30000 = 8625 \text{ грн}$$

Розрахуємо витрати на ремонт та профілактику:

$$С_Р = К_{ТМ} * Ц_{ПР} * К_Р = 1,15 * 30000 * 0,05 = 1725 \text{грн}$$

Розрахуємо ефективний годинний фонд часу ПК за рік

$$Т_{ЕФ} = (365 - 142 - 16) * 8 * 0,8 = 1324,8 \text{ год}$$

Розрахуємо витрати на електроенергію

$$С_{ЕЛ} = 1324,8 * 0,6 * 0,6 * 3.52 = 1678,78 \text{ грн}$$

Накладні витрати рівні:

$$С_Н = 30000 * 0,67 = 20100 \text{ грн.}$$

Отже, експлуатаційні витрати(грн):

$$С_{ЕКС} = 270720 + 59558,4 + 8625 + 1725 + 834,62 + 20100 = 361563,02$$

Тоді собівартість однієї машино-години ЕОМ дорівнюватиме:

$$С_{М-Г} = \frac{361563,02}{1324,8} = 272,92 \text{ грн/год}$$

Враховуючи, що всі роботи ведуться на ЕОМ, витрати на оплату машинного часу:

1) $С_м = 272,92 * 8 * 173,08 = 377895,08$
2) $С_м = 272,92 * 8 * 119,51 = 260933,35$

Накладні витрати відповідно

1) $С_{Н} = 377895,08 * 0,67 = 253189,70$
2) $С_{Н} = 260933,35 * 0,67 = 174825,34$

Розрахуємо повну вартість розробки за варіантами:

1) $С_{ПП} = 92438,57 + 20336,49 + 377895,08 + 253189,70 = 743859,84$
2) $С_{ПП} = 63827,90 + 14042,14 + 260933,35 + 174825,34 = 513628,73$

4.4 Вибір кращого варіанта ПП техніко-економічного рівня

Розрахуємо коефіцієнт техніко-економічного рівня

$$К_{я1} = 0,95 + 0 + 1,4 + 1,5 + 0,9 = 4,75$$

$$К_{я2} = 0,95 + 1 + 1,4 + 0,75 + 0,9 = 5$$

$$К_{я3} = 0,95 + 1,4 + 0 + 0,75 + 0,9 = 4$$

$$К_{я4} = 0,95 + 1 + 1,4 + 1,5 + 0,9 = 5,75$$

$$К_{ТЕР1} = \frac{4,75}{743859,84} = 6,38 * 10^{-6}$$

$$К_{ТЕР2} = \frac{5}{513628,73} = 9,73 * 10^{-6}$$

$$К_{ТЕР3} = \frac{4}{743859,84} = 5,38 * 10^{-6}$$

$$К_{ТЕР4} = \frac{5,75}{513628,73} = 1,12 * 10^{-5}$$

4.5 Висновки

Отже, враховуючи всі дослідження, що описані вище, можна сказати, що 4 варіант реалізації є найбільш оптимальним зі сторони якісно-економічної оцінки. Його коефіцієнт техніко-економічного рівня складає $1,12*10^{-5}$.

Розробка цього варіанту передбачає такі обов’язкові завдання як:

1) Завантаження даних з фізичного носія

2) Проведення тестування на продуктивність

3) Запис даних у БД та вивід користувачу

Серед завдань між якими ставився вибір в даному варіанті реалізовані такі завдання:

2.2) Використання окремих ознак

3.1) Використання основної ознаки - зарахування

ВИСНОВКИ

Для написання дипломної роботи, були досліджені види тестувань програмного забезпечення, їх класифікація в залежності від функціональності та процесу, які ці методи реалізують. Додатковою необхідністю стало дослідження серверних технологій, що використовуються для налаштування та впровадження веб-додатків бізнес-логіки. В ході роботи було проаналізовано серверні технології з використанням середи тестування Gatling.

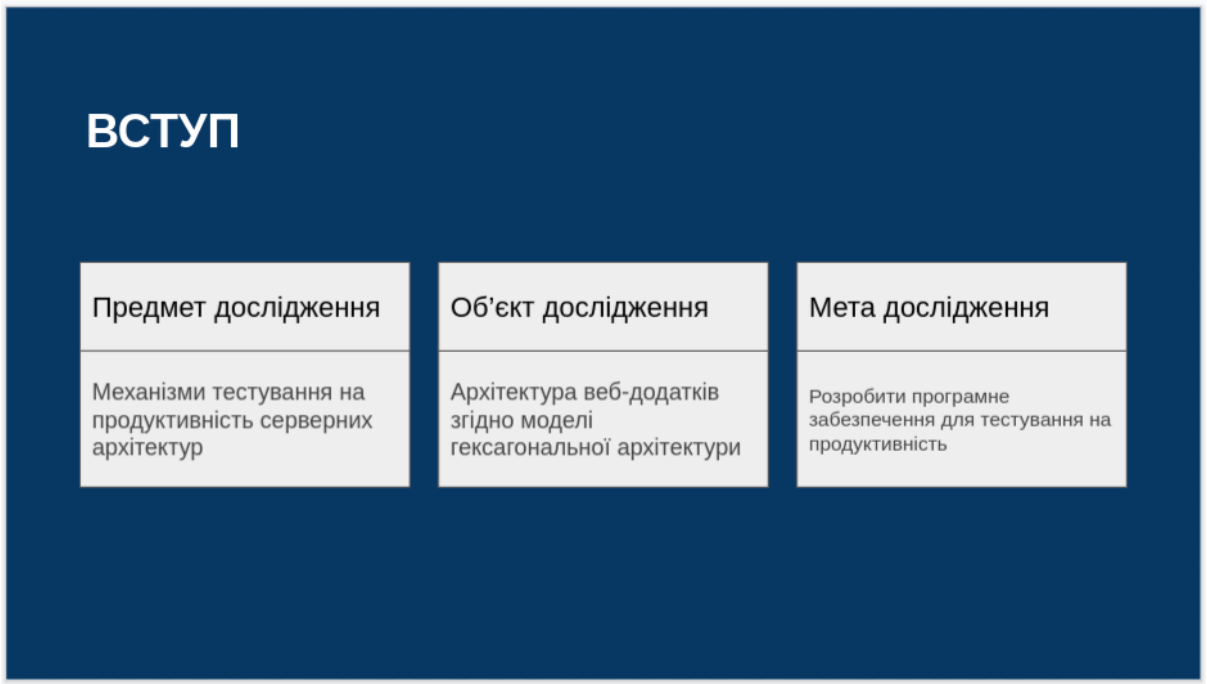
За допомогою цієї роботи було опрацьовано матеріал з принципів тестування серверного програмного забезпечення, її структура. Також виконуючи цю роботу, були придбанні знання з побудування внутрішньої структури серверних технологій, особливості розгортання бізнес-логіки веб-додатків на цих серверних технологіях використовуючи модель гексагональної архітектури.

Дослідивши отримані результати можемо зробити висновки про те, що найкраща серверна технологія є Jetty на віртуальній машині OpenJDK та Quarkus на віртуальній машині GraalVM. У цих технологіях час відгуку є найменшим. Quarkus технологія показала можливість обслуговування великої кількості користувацьких запитів у порівнянні з іншими технологіями при такому ж навантаженні.

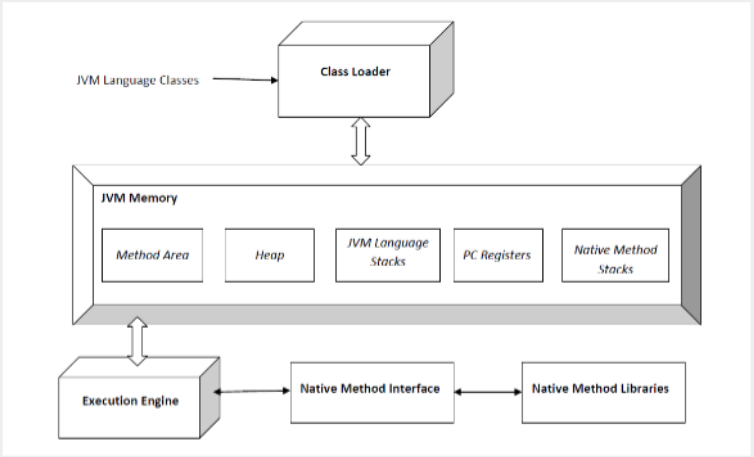
ПЕРЕЛІК ПОСИЛАНЬ

1. Open Versus Closed: A Cautionary Tale
   (URL:
   https://www.usenix.org/legacy/event/nsdi06/tech/full_papers/schroeder/schroeder.pdf)
2. Hexagonal Architecture: three principles and an implementation example (URL:https://blog.octo.com/en/hexagonal-architecture-three-principles-and-an-implementation-example/)
3. Different Types of Testing in Software
   (URL: https://www.perfecto.io/resources/types-of-testing)
4. Darwin I. F Java cookbook. 2nd ed. Sebastopol: книга. USA:O'Reilly, 2004. 829p.
5. Goodrich M. T. Data structures and algorithms in Java. 5th ed: книга. USA: J. Wiley & Sons, 2010. 358p.
6. Laaksonen E., Simons T., Vartola A. Research and practice in architecture: книга. USA:Rakennustieto Publishing, 2001. 124 p.
7. Liang Y. D., Liang Y. Introduction to java programming and data structures, comprehensive version: книга. USA:Pearson, 2019. 1240 p.
8. Manelli L., Zambon G. Beginning jakarta EE web development: книга USA:Apress, 2020. 407p.
9. Porter J., Bueno A. S., Gumbrecht A. Testing java microservices: using arquillian, hoverfly, assertj, junit, selenium, and mockito: книга. USA: Manning Publications, 2018. 296 p.
10. Porter J., Soto A. Quarkus cookbook: kubernetes-optimized java solutions: книга. USA:O'Reilly Media, Incorporated, 2020. 382 p.
11. Skiena S. S. The algorithm design manual: книга. USA:Springer, 2020. 810 p.

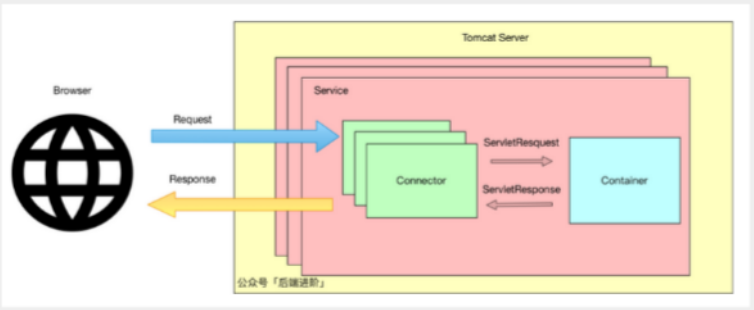
ДОДАТОК А ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

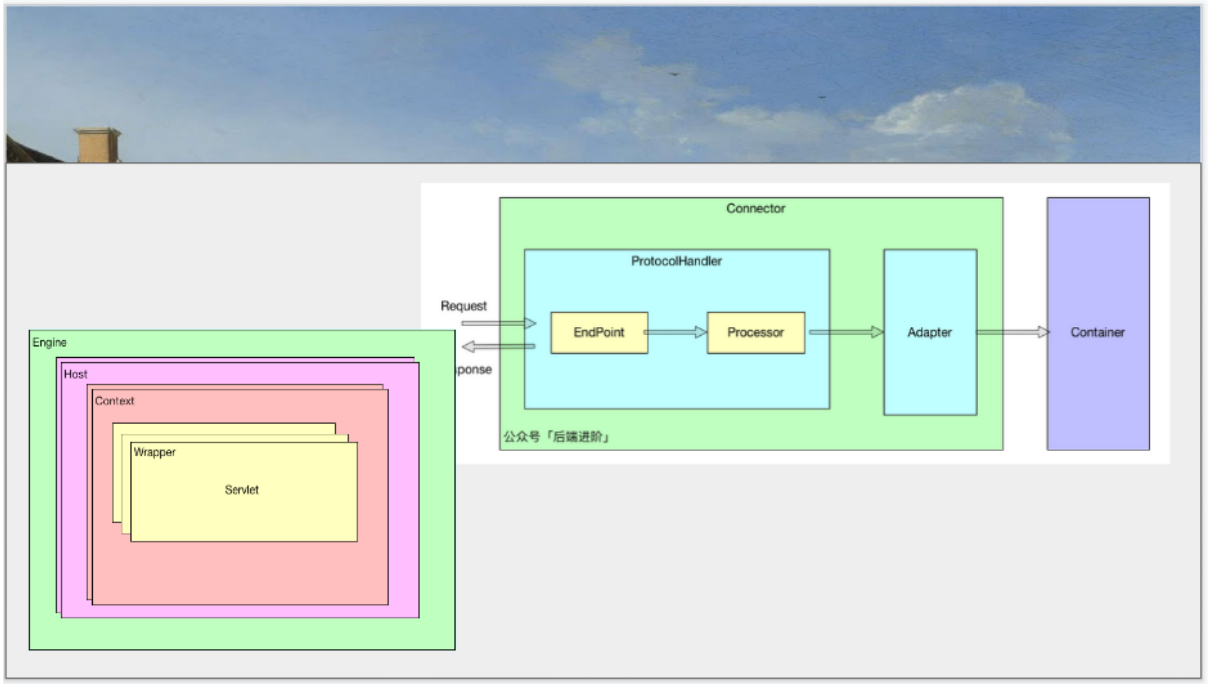


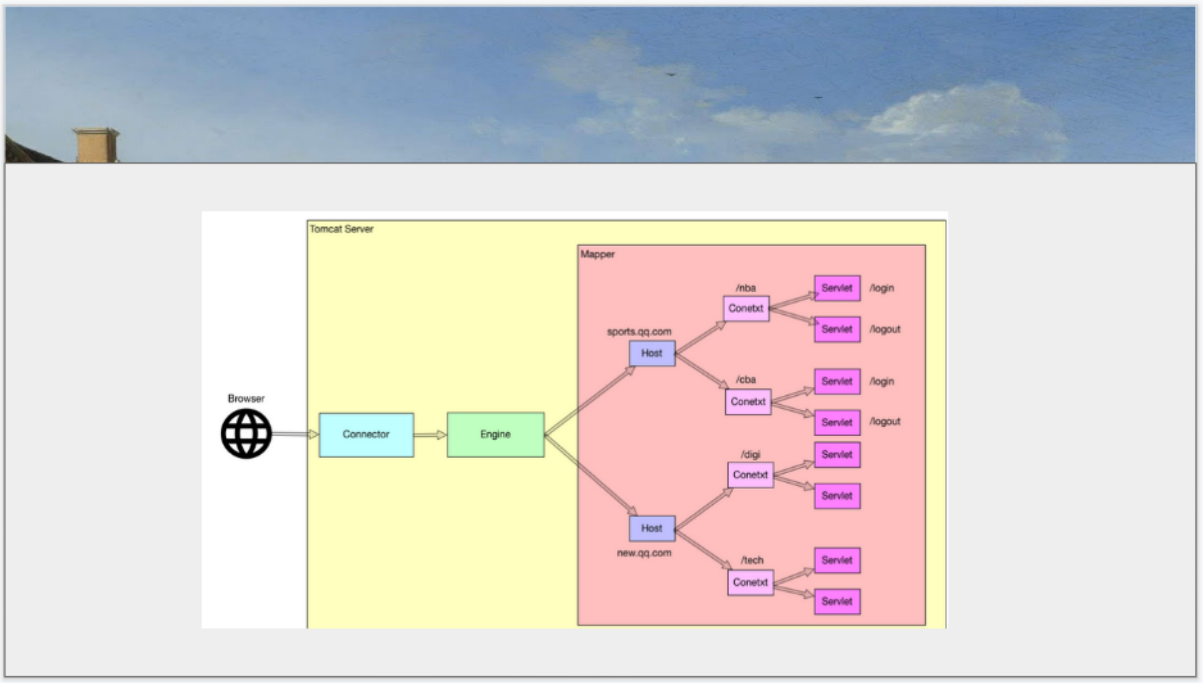
JVM


JVM Language Classes
Class Loader
JVM Memory
Method Area
Heap
JVM Language Stacks
PC Registers
Native Method Stacks
Execution Engine
Native Method Interface
Native Method Libraries

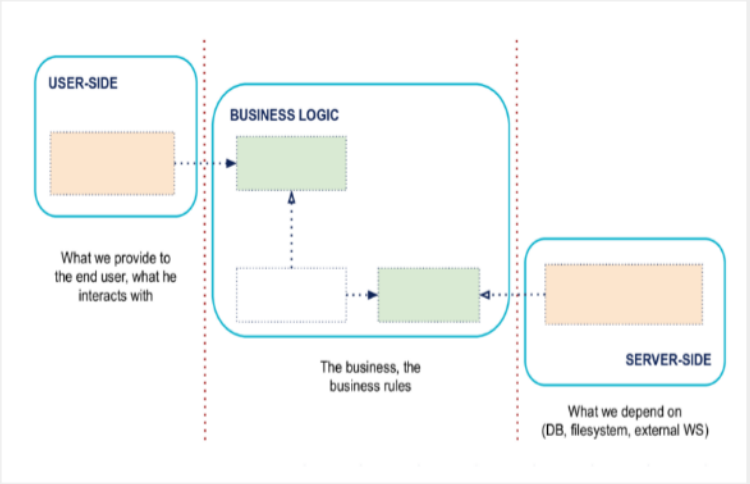
Tomcat


Tomcat Server
Service
Browser
Request
Response
Connector
ServletResquest
ServletResponse
Container
公众号「后端进阶」

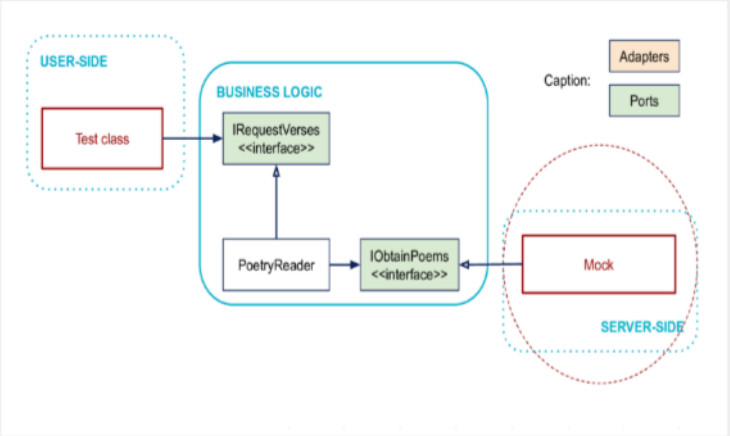

Connector
ProtocolHandler
Request
EndPoint
Processor
Adapter
Container
ponse
Engine
Host
Context
Wrapper
Servlet
公众号「后端进阶」


Tomcat Server
Mapper
Browser
Connector
Engine
sports.qq.com
Host
/nba
Conetxt
Servlet
/login
Servlet
/logout
/cba
Conetxt
Servlet
/login
Servlet
/logout
/digi
Conetxt
Servlet
Servlet
Host
new.qq.com
/tech
Conetxt
Servlet
Servlet

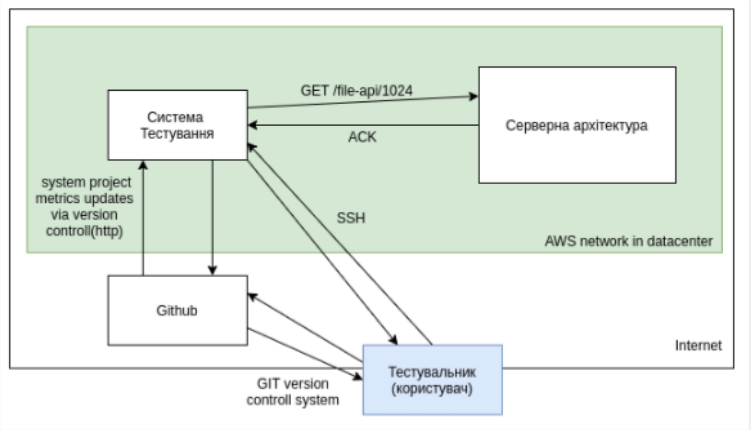
Гексагональна архітектура



Тестування у гексагональній архітектурі



Загальний вигляд середи тестування



Апаратна архітектура

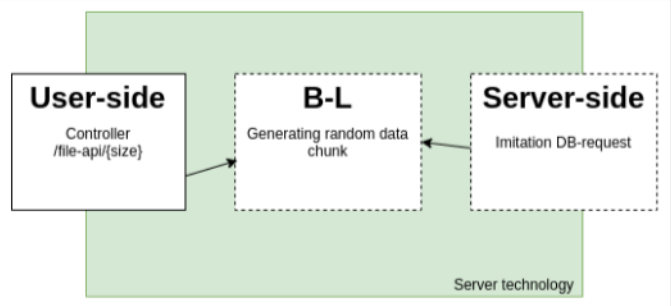
AWS m5.2xlarge EC2

8 ядер 32гб оперативної пам’яті

Тестування на продуктивність
Збільшення 100 користувачів в секунду з межою до 1000
протягом 1 хвилини

Стрес тестування
Збільшення 100 користувачів в секнуду з межою до 5000
протягом 1 хвилини

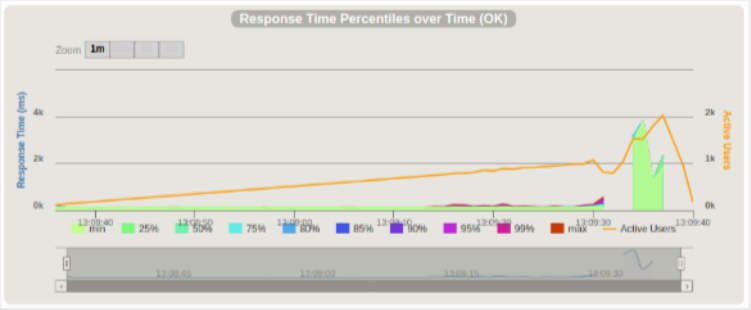
Реалізація середи тестування серверної частини у гексагональній архітектурі



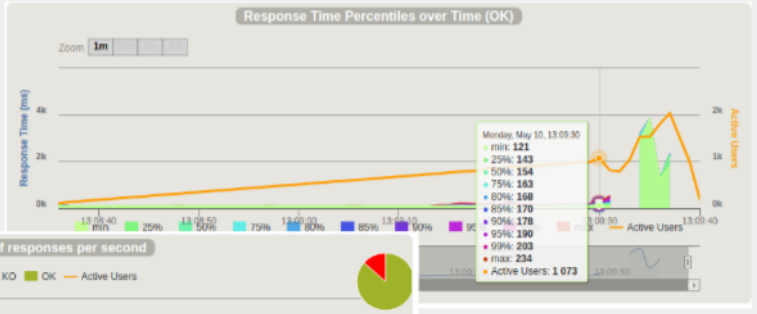
Результати роботи тестування Jetty

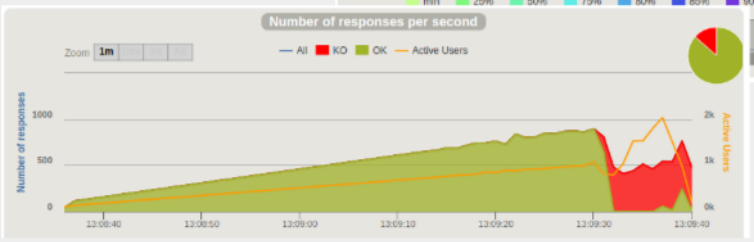
STATISTICS — Expand all groups | Collapse all groups

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Cnt/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	33000	28566	4434	13%	507.692	6	103	120	2172	3083	3828	326	650
Get a ki... of data	33000	28566	4434	13%	507.692	6	103	120	2172	3083	3828	326	650

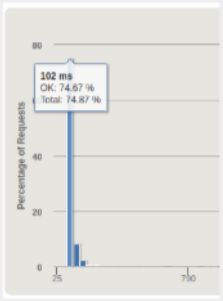


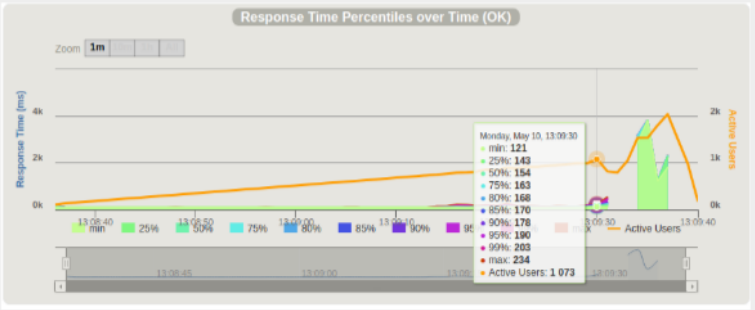
Результати роботи тестування Jetty



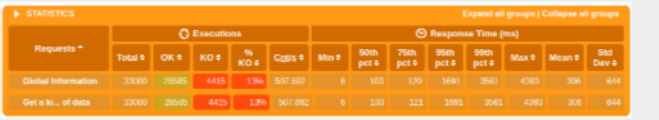


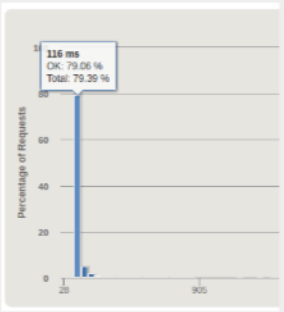
Результати роботи тестування Jetty

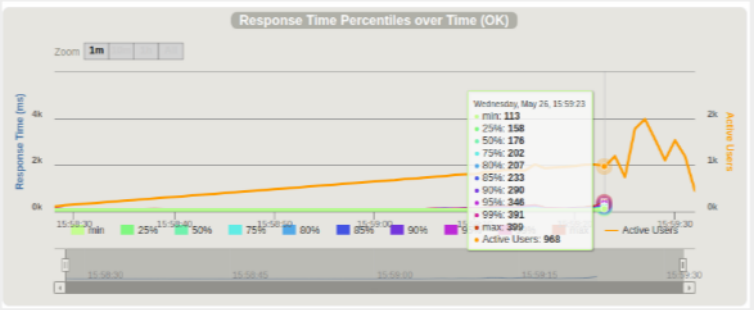




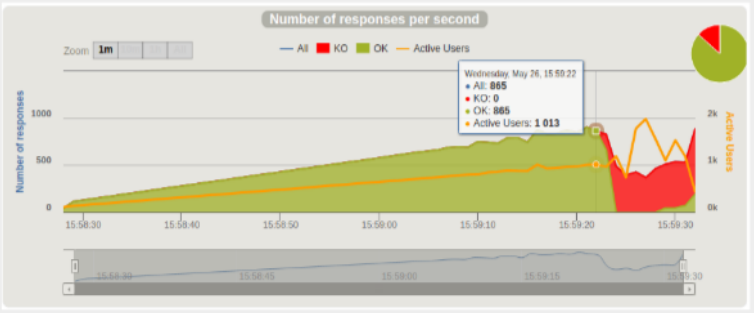
Результати роботи тестування Tomcat



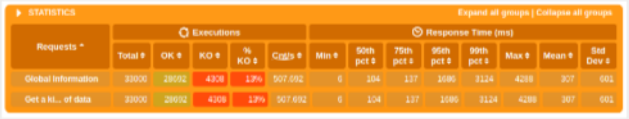


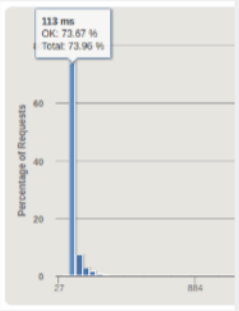


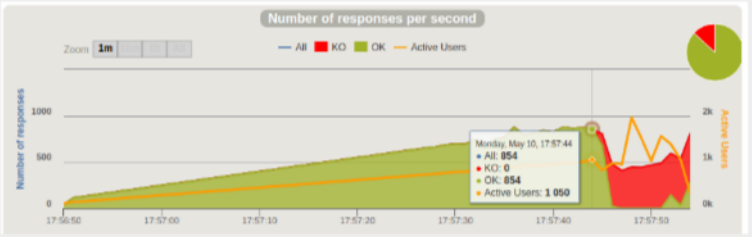
Результати роботи тестування Tomcat

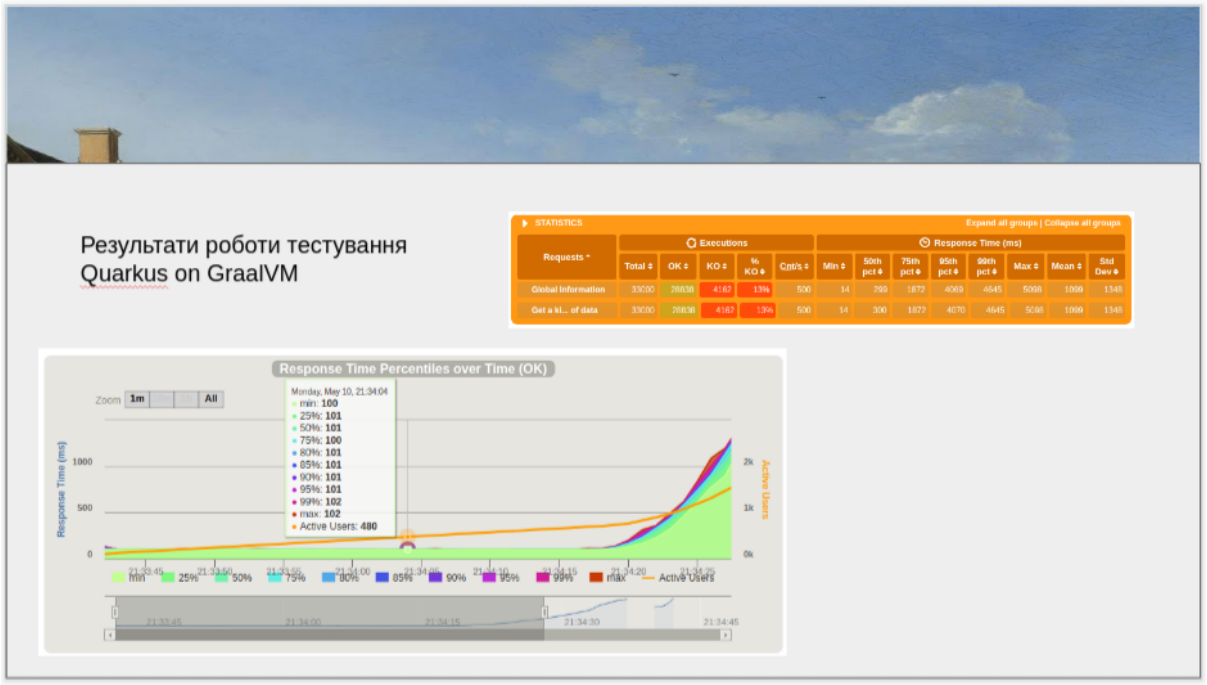


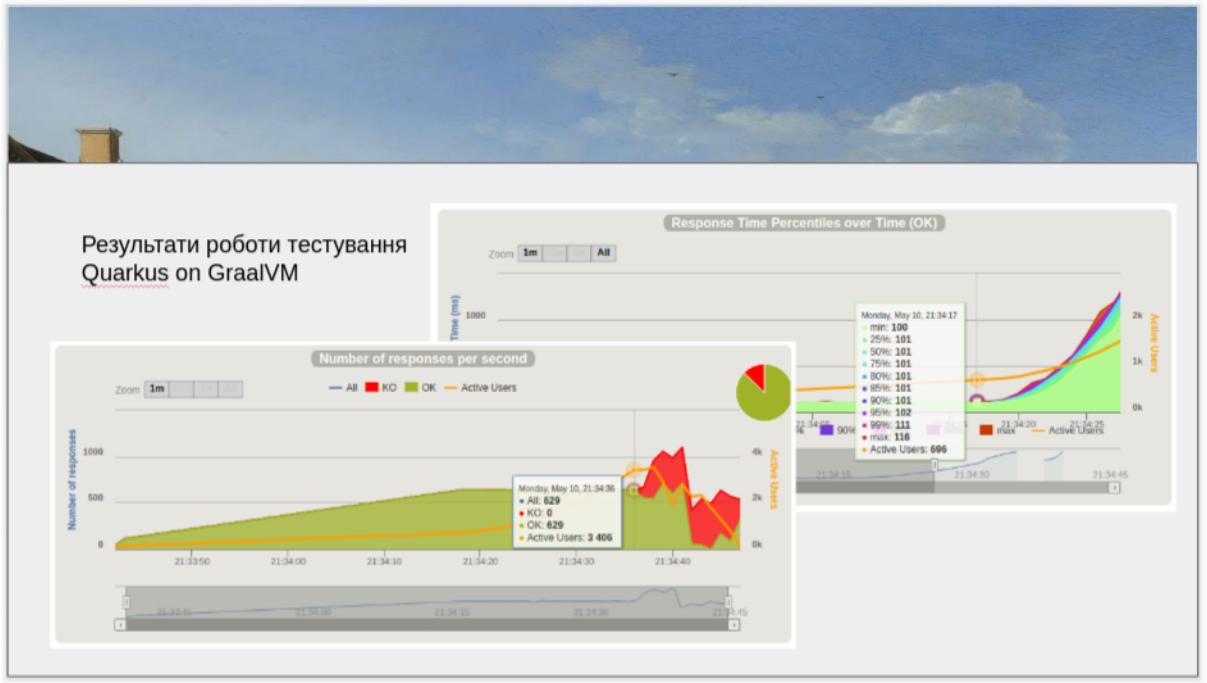
Результати роботи тестування Wildfly

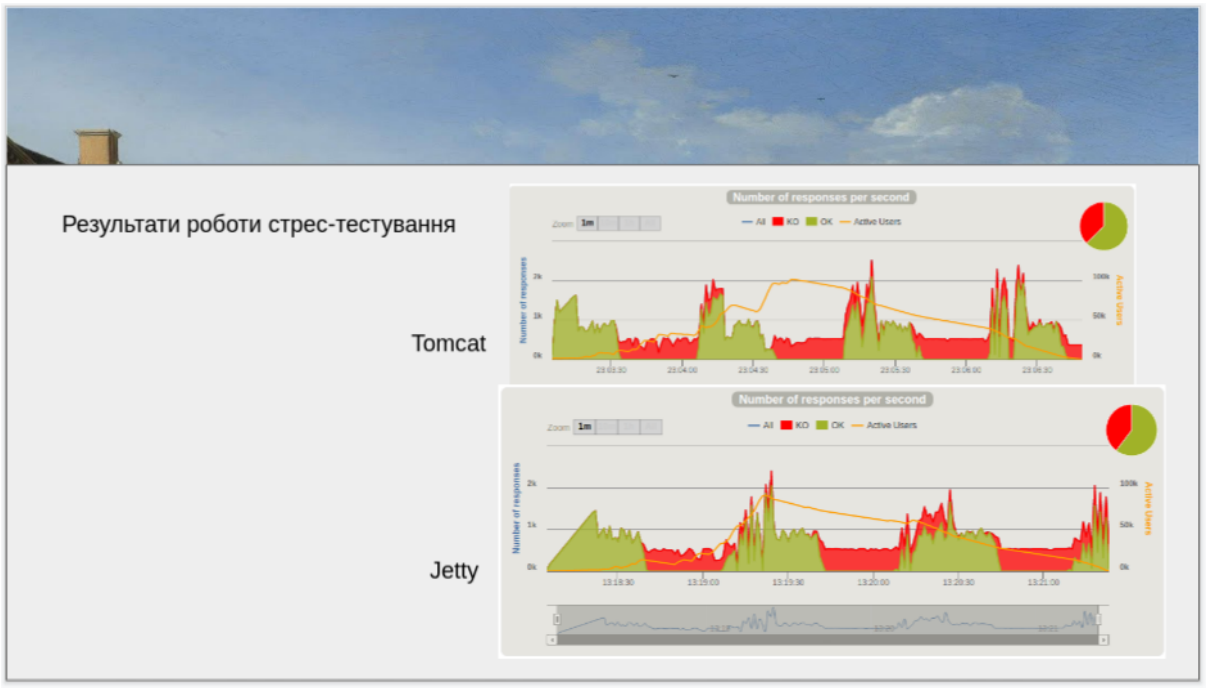


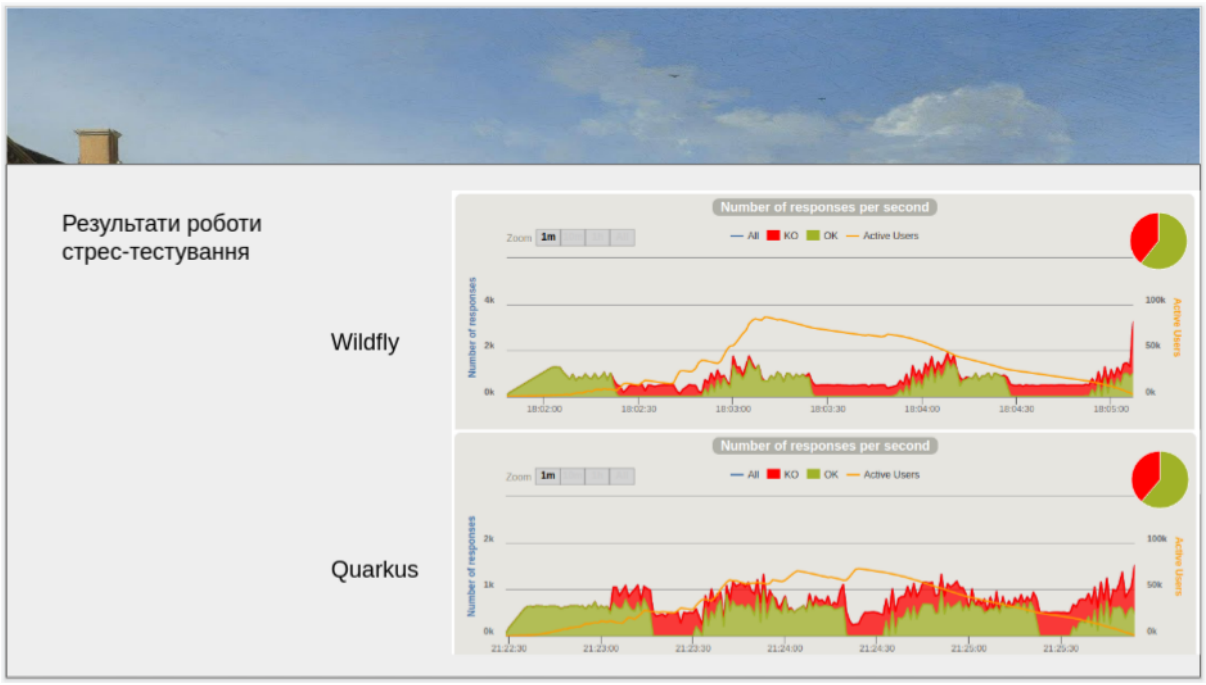





Результати роботи тестування
Quarkus on GraalVM
STATISTICS
Expand all groups | Collapse all groups
Requests
Executions
Response Time (ms)
Total
OK
KO
% KO
Cnt/s
Min
50th pct
75th pct
95th pct
99th pct
Max
Mean
Std Dev
Global Information
33000
28838
4162
13%
500
14
299
1872
4069
4645
5098
1099
1348
Get a kl... of data
33000
28838
4162
13%
500
14
300
1872
4070
4645
5098
1099
1348
Response Time Percentiles over Time (OK)
Zoom
1m
All
Monday, May 10, 21:34:04
min: 100
25%: 101
50%: 101
75%: 100
80%: 101
85%: 101
90%: 101
95%: 101
99%: 102
max: 102
Active Users: 480
Response Time (ms)
Active Users
min
25%
50%
75%
80%
85%
90%
95%
99%
max
Active Users


Результати роботи тестування
Quarkus on GraalVM
Response Time Percentiles over Time (OK)
Zoom
1m
All
Monday, May 10, 21:34:17
min: 100
25%: 101
50%: 101
75%: 101
80%: 101
85%: 101
90%: 101
95%: 102
99%: 111
max: 116
Active Users: 696
Number of responses per second
Zoom
1m
All
KO
OK
Active Users
Number of responses
Active Users
Monday, May 10, 21:34:36
All: 629
KO: 0
OK: 629
Active Users: 3 406


Результати роботи стрес-тестування
Tomcat
Number of responses per second
Zoom 1m
All KO OK Active Users
Number of responses
Active Users
Jetty
Number of responses per second
Zoom 1m
All KO OK Active Users
Number of responses
Active Users


Результати роботи стрес-тестування
Wildfly
Number of responses per second
Zoom 1m
All KO OK Active Users
Number of responses
Active Users
Quarkus
Number of responses per second
Zoom 1m
All KO OK Active Users
Number of responses
Active Users

ДОДАТОК Б ЛІСТИНГ ПРОГРАМИ

ServerPerformanceAnalytics.scala - скрипт тестування

```
package simulations

import io.gatling.core.Predef._
import io.gatling.http.Predef._

import scala.concurrent.duration._
import scala.language.postfixOps

class ServerPerformanceAnalytics extends Simulation {

 var httpConfig = http.baseUrl("http://34.236.33.146:8080/file-api/")
   .header("Accept", "text/plain")

 def getKilobyteOfData() = {
   exec(
     http("Get a kilobyte of data")
       .get("1024")
       .check(status.is(200))
   )
 }

 val scn = scenario("Test of performance qks on graalvm")
   .exec(getKilobyteOfData())

 setUp(
   scn.inject(
     nothingFor(1 seconds),
```

```
    rampUsersPerSec(100) to (1000) during(1 minutes)
  ).protocols(httpConfig.inferHtmlResources())
 )
}
```

Engine.scala - налаштування двигуну тестування

```
import io.gatling.app.Gatling
import io.gatling.core.config.GatlingPropertiesBuilder

object Engine extends App {

val props = new GatlingPropertiesBuilder()
 .resourcesDirectory(IDEPathHelper.mavenResourcesDirectory.toString)
 .resultsDirectory(IDEPathHelper.resultsDirectory.toString)
 .binariesDirectory(IDEPathHelper.mavenBinariesDirectory.toString)

Gatling.fromMap(props.build)
}
```

IDEPathHelper.scala - налаштування середи тестування

```
import java.nio.file.Paths
object IDEPathHelper {

private val projectRootDir =
Paths.get(getClass.getClassLoader.getResource("gatling.conf").toURI).getParen
t.getParent.getParent
private val mavenTargetDirectory = projectRootDir.resolve("target")
private val mavenSrcTestDirectory =
projectRootDir.resolve("src").resolve("test")
```

```
val mavenSourcesDirectory = mavenSrcTestDirectory.resolve("scala")
val mavenResourcesDirectory = mavenSrcTestDirectory.resolve("resources")
val mavenBinariesDirectory = mavenTargetDirectory.resolve("test-classes")
val resultsDirectory = mavenTargetDirectory.resolve("gatling")
val recorderConfigFile = mavenResourcesDirectory.resolve("recorder.conf")
}
```

Recorder.scala - налаштування генерації результату системою тестування.

```
import io.gatling.recorder.GatlingRecorder
import io.gatling.recorder.config.RecorderPropertiesBuilder

object Recorder extends App {

val props = new RecorderPropertiesBuilder()
 .simulationsFolder(IDEPathHelper.mavenSourcesDirectory.toString)
 .resourcesFolder(IDEPathHelper.mavenResourcesDirectory.toString)
 .simulationPackage("computerdatabase")

GatlingRecorder.fromMap(props.build,
Some(IDEPathHelper.recorderConfigFile))
}
```

pom.xml - файл проектної структури системи тестування

```
<?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
http://maven.apache.org/xsd/maven-4.0.0.xsd">
```

```
<modelVersion>4.0.0</modelVersion>

<groupId>com.james-willett</groupId>
<artifactId>gatling3-fundamentals</artifactId>
<version>3.5.0</version>

<properties>
  <maven.compiler.source>1.8</maven.compiler.source>
  <maven.compiler.target>1.8</maven.compiler.target>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <gatling.version>${project.version}</gatling.version>
  <gatling-maven-plugin.version>3.1.1</gatling-maven-plugin.version>
  <maven-jar-plugin.version>3.2.0</maven-jar-plugin.version>
  <scala-maven-plugin.version>4.4.0</scala-maven-plugin.version>
</properties>

<dependencies>
  <dependency>
    <groupId>io.gatling.highcharts</groupId>
    <artifactId>gatling-charts-highcharts</artifactId>
    <version>${gatling.version}</version>
  </dependency>
  <dependency>
    <groupId>io.gatling</groupId>
    <artifactId>gatling-app</artifactId>
    <version>${gatling.version}</version>
  </dependency>
  <dependency>
    <groupId>io.gatling</groupId>
    <artifactId>gatling-recorder</artifactId>
```

```
    <version>${gatling.version}</version>
  </dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <artifactId>maven-jar-plugin</artifactId>
      <version>${maven-jar-plugin.version}</version>
    </plugin>
    <plugin>
      <groupId>net.alchim31.maven</groupId>
      <artifactId>scala-maven-plugin</artifactId>
      <version>${scala-maven-plugin.version}</version>
      <executions>
        <execution>
          <goals>
            <goal>testCompile</goal>
          </goals>
          <configuration>
            <jvmArgs>
              <jvmArg>-Xss100M</jvmArg>
            </jvmArgs>
            <args>
              <arg>-target:jvm-1.8</arg>
              <arg>-deprecation</arg>
              <arg>-feature</arg>
              <arg>-unchecked</arg>
              <arg>-language:implicitConversions</arg>
              <arg>-language:postfixOps</arg>
```

```
            </args>
          </configuration>
        </execution>
      </executions>
    </plugin>
    <plugin>
      <groupId>io.gatling</groupId>
      <artifactId>gatling-maven-plugin</artifactId>
      <version>${gatling-maven-plugin.version}</version>
      <configuration>

<simulationClass>simulations.ServerPerformanceAnalytics</simulationClass>
      </configuration>
    </plugin>
  </plugins>
 </build>
</project>
```

SampleController.java - контролер серверу

```
package com.onetwostory.simpletomcatanalyticssample.controller;

import
com.onetwostory.simpletomcatanalyticssample.manager.DataChunkGeneration
Manager;
import java.io.IOException;
import lombok.RequiredArgsConstructor;
import org.springframework.metrics.annotation.Timed;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
```

```
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequiredArgsConstructor
@RequestMapping("/file-api")
@Timed
public class SampleController {

  private final DataChunkGenerationManager dataChunkGenerationManager;

  @GetMapping(value = "/{returnFileSize}",
      produces = "text/plain")
  public @ResponseBody byte[] processRequestWithFixedFileSizeAndLatency(
      @PathVariable("returnFileSize") Integer dataSize) {

    try { Thread.sleep(100); }
    catch (InterruptedException ex) { ex.printStackTrace(); }

    return
dataChunkGenerationManager.generateFixedSizeDataChunk(dataSize);

  }

}
```

Data Chunk File Manager - Генератор випадкового кілобайту даних.

```
import java.nio.file.Path;
import lombok.extern.log4j.Log4j2;
```

```
import org.springframework.stereotype.Component;

@Log4j2
@Component
public class DataChunkFileManager {

  private static volatile byte[] dataChunk;
  private static final String chunkFileName = "1024 bytes";
  private static final int size = 1024;

  private DataChunkFileManager() {}

  public static byte[] getDataChunk() {
    if (dataChunk == null) {
      synchronized (DataChunkFileManager.class) {
        if (dataChunk == null) {

          dataChunk = new byte[size];

          try {
            dataChunk = Files.readAllBytes(Path.of(chunkFileName));
          } catch (IOException e) {
            e.printStackTrace();
          }

          log.info(String.format("Fetched %s bytes", size));

        }
      }
    }
```

```
    return dataChunk;
  }

}
```

Simple Analytics Sample Application - точка входу серверної частини. Даний код однаковий для всіх тестувальних серверів.

```
package com.onetwostory.simpletomcatanalyticssample;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import
org.springframework.metrics.export.prometheus.EnablePrometheusMetrics;

@SpringBootApplication
// @EnablePrometheusMetrics
public class SimpleTomcatAnalyticsSampleApplication {

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

}
```

pom.xml - файл проектної структури серверу.

```
<?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>2.4.4</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>
  <groupId>com.onetwostory</groupId>
  <artifactId>simple-tomcat-analytics-sample</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>simple-tomcat-analytics-sample</name>
  <description>Demo project for Spring Boot</description>
  <properties>
    <java.version>11</java.version>
  </properties>
  <dependencies>
```

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

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-devtools</artifactId>
  <scope>runtime</scope>
  <optional>true</optional>
</dependency>
<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
  <optional>true</optional>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
```

```
    </dependency>

    <!--
https://mvnrepository.com/artifact/org.springframework.metrics/spring-metrics
-->

    <dependency>

      <groupId>org.springframework.metrics</groupId>

      <artifactId>spring-metrics</artifactId>

      <version>0.5.1.RELEASE</version>

    </dependency>


    <!-- https://mvnrepository.com/artifact/io.prometheus/simpleclient_common
-->

    <dependency>

      <groupId>io.prometheus</groupId>

      <artifactId>simpleclient_common</artifactId>

      <version>0.10.0</version>

    </dependency>


  </dependencies>


  <build>
```

```
<!--    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
        <configuration>
          <excludes>
            <exclude>
              <groupId>org.projectlombok</groupId>
              <artifactId>lombok</artifactId>
            </exclude>
          </excludes>
        </configuration>
      </plugin>
    </plugins>-->
  </build>


</project>
```