НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

«На правах рукопису»
УДК 004.42

«До захисту допущено»
Завідувач кафедри
_____ Едуард ЖАРІКОВ
«___»_____2024 р.

Магістерська дисертація

на здобуття ступеня магістра

за освітньо-професійною програмою «Інженерія програмного забезпечення інформаційних систем»

зі спеціальності 121 «Інженерія програмного забезпечення»

на тему: «Фреймворк для зниження витрат на утримання Java WEB серверів»

Виконав:
студент ІІ курсу, групи ІП-22мп
Венделовський Іван Сергійович _____

Керівник:
ст.в. кафедри ІПІ
Халус Олена Андріївна _____

Рецензент:
д.т.н. проф каф ІСТ
Корнага Ярослав Ігорович _____

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент_____

Київ – 2024 року

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

Рівень вищої освіти – другий (магістерський)

Спеціальність – 121 «Інженерія програмного забезпечення»

Освітньо-професійна програма «Інженерія програмного забезпечення інформаційних систем»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Едуард ЖАРІКОВ

«___»_____2023р.

ЗАВДАННЯ

на магістерську дисертацію студенту

Венделовському Івану Сергійовичу

1. Тема дисертації «Фреймворк для зниження витрат на утримання Java WEB серверів», науковий керівник дисертації Халус Олена Андріївна, старший викладач кафедри ІПІ, затверджені наказом по університету від «11» листопада 2023 р. № 5168-с

2. Термін подання студентом дисертації «16» січня 2024 р.

3. Об’єкт дослідження – Java Web фреймворки

4. Предмет дослідження – Методи зниження часу запуску веб фреймворку та покращення архітектури для зниження витрат на підтримку.

5. Перелік завдань, які потрібно розробити – аналіз напрямків зниження витрат на утримання Java Web серверів; розробка методу створення фреймворку для мінімізації вартості утримання; реалізація фреймворку за запропонованим методом, дослідження ефективності розробленого фреймворку в порівнянні з лідерами ринку.

6. Орієнтовний перелік графічного (ілюстративного) матеріалу – 28 плакати

7. Орієнтовний перелік публікацій – Венделовський І.С., Халус О.А. Фреймворк для зниження витрат на утримання java WEB серверів // V Міжнародна науково-практична конференція молодих вчених та студентів «ІНЖЕНЕРІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ І ПЕРЕДОВІ ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ» (SoftTech-2023), Матеріали конференції.

8. Консультанти розділів дисертації

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

9.Дата видачі завдання «1» листопада 2022 р.

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання	Примітка
1	Огляд літератури	1.11.2022	
2	Аналіз існуючих методів зниження витрат на утримання java WEB серверів	01.01.2023	
3	Розробка методу створення фреймворку для мінімізації витрат на утримання java WEB серверів	01.04.2023	
4	Розробка фреймворку за розробленим методом	01.08.2023	
5	Виконання експериментальних досліджень	01.09.2023	
6	Оформлення пояснювальної записки	01.10.2023	
7	Подання дисертації на попередній захист	19.12.2023	
8	Подання дисертації на захист	19.01.2024	

Студент Іван ВЕНДЕЛОВСЬКИЙ

Науковий керівник Олена ХАЛУС

РЕФЕРАТ

Розмір пояснювальної записки – 131 аркушів, містить 28 ілюстрацій, 22 таблиць, 2 додатків, 34 посилань на джерела.

Актуальність теми.

Зниження вартості утримання WEB серверів досі актуальне. Для досягнення цього наразі відбувається активний перехід до хмарних сервісів. Розроблений фреймворк пропонує зниження витрат на утримання за рахунок розширення кола можливостей застосування дешевих спотових віртуальних машин.

Попутно у фреймворку впроваджено деякі додаткові архітектурні обмеження та доповнення. Вони покликані додатково знизити час запуску за рахунок спрощення інфраструктури та одночасно знизити витрати на підтримку за рахунок змушення розробника до слідування кращим практикам.

Мета дослідження. Знизити витрати на утримання Java WEB серверів.

Об’єкт дослідження: Java WEB фреймворки.

Предмет дослідження: Методи зниження часу запуску веб фреймворку та покращення архітектури для зниження витрат на підтримку.

Для реалізації поставленої мети **сформульовані наступні завдання**:

- аналіз напрямків зниження витрат на утримання.
- розробити метод побудови WEB фреймворку з мінімальним часом запуску.
- розробити фреймворк за методом.
- знизити наслідки компромісних рішень.
- порівняти з лідерами ринку ступінь зниження часу запуску.

Наукова новизна результатів магістерської дисертації полягає в тому, що запропоновано метод розробки WEB фреймворків для зниження витрат на утримання шляхом зниження часу запуску та мінімізації помилок у користувацькому коді. Запропоновано архітектурне рішення для унеможливлення допущення користувачами помилок при розробці WEB додатків за архітектурою REST. Запропоновано архітектурне рішення для підтримки робти з WEB формами у архітектурному стилі REST.

Практичне значення отриманих результатів полягає в тому, що реалізований фреймворк дозволяє розширити коло застосування спотових віртуальних машин. Що є значно дешевшими за віртуальні машини що пропонуються в рамках інших тарифів з тою самою конфігурацією.

Ліквідація з допомогою розробленого рішення можливості допущення деяких типових помилок дозволить ефективніше використовувати низько кваліфікованих розробників.

Зв'язок з науковими програмами, планами, темами. Робота виконувалась на кафедрі інформатики та програмної інженерії Національного технічного університету України "Київський політехнічний інститут імені Ігоря Сікорського".

Апробація. Наукові положення дисертації пройшли апробацію на V міжнародній науково-практичній конференції молодих вчених та студентів «ІНЖЕНЕРІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ І ПЕРЕДОВІ ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ» (SoftTech-2023).

Публікації. Наукові положення дисертації опубліковані в:

- Венделовський І.С., Халус О.А. Фреймворк для зниження витрат на утримання java WEB серверів // Матеріали V міжнародної науково-практичної конференції молодих вчених та студентів «ІНЖЕНЕРІЯ

ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ І ПЕРЕДОВІ ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ» (SoftTech-2023).

Ключові слова: JAVA WEB СЕРВЕР, КОНТЕЙНЕР ІНВЕРСІЇ ЗАЛЕЖНОСТЕЙ, СПОТОВІ ВІРТУАЛЬНІ МАШИНИ, ЗНИЖЕННЯ ВАРТОСТІ ПІДТРИМКИ.

ABSTRACT

Explanatory note size – 131 pages, contains 28 illustrations, 22 tables, 2 applications, 34 references.

Topicality. Minimizing the costs of maintaining WEB servers is still relevant. To achieve this, there is currently an active transition to cloud services. The developed framework offers a reduction in maintenance costs due to expanding the range of possibilities for using cheap spot virtual machines.

Along the way, additional architectural constraints and improvements have been introduced into the framework. They are designed to further reduce startup time by simplifying the infrastructure and reducing support costs by forcing the developer to follow best practices.

The aim of the study. Reduce the cost of maintaining Java WEB servers.

The object of research: Java WEB frameworks

The subject of research: Methods to reduce web framework startup time and improve architecture to reduce support costs.

To achieve this goal, the **following tasks** were formulated:

- analysis of ways to reduce maintenance costs.
- to develop a method of building a WEB framework with a minimum start-up time.
- develop a framework according to the method.
- reduce the consequences of compromise decisions.
- compare the degree of decrease in start-up time with the market leaders.

The scientific novelty of the results of the master's dissertation is that a method of developing WEB frameworks is proposed to reduce maintenance costs by reducing startup time and minimizing errors in user code. An architectural solution is proposed to prevent users from making mistakes when developing WEB applications based on the REST architecture. An architectural solution is proposed to support working with WEB forms in the REST architectural style.

The practical value of the obtained results is that the implemented framework allows expanding the scope of application of spot virtual machines. Which is much cheaper than virtual machines offered under other tariffs with the same configuration.

Eliminating with the help of the developed solution the possibility of making some typical mistakes will allow more efficient use of low-skilled developers.

Relationship with working with scientific programs, plans, topics. Work was performed at the Department of Informatics and Software Engineering of the National Technical University of Ukraine «Kyiv Polytechnic Institute. Igor Sikorsky».

Approbation. The scientific provisions of the dissertation were approved at the 5th international scientific and practical conference of young scientists and students "SOFTWARE ENGINEERING AND ADVANCED INFORMATION TECHNOLOGIES" (SoftTech-2023).

Publications. The scientific provisions of the dissertation were published in:

- Vendelovskyi I.S., Khalus O.A. Framework for reducing maintenance costs of java WEB servers // Materials of the 5th international scientific and practical conference of young scientists and students "SOFTWARE ENGINEERING AND ADVANCED INFORMATION TECHNOLOGIES" (SoftTech-2023).

Keywords: JAVA WEB SERVER, INVERSION OF CONTROL CONTAINER, SPOT VIRTUAL MACHINES, REDUCING THE COST OF SUPPORT.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

REST (Representational State Transfer) – це архітектурний стиль для розробки веб-сервісів. Він використовує HTTP протокол для передачі даних між клієнтом і сервером і базується на концепції ресурсів, які ідентифікуються унікальними URI (Uniform Resource Identifiers).

WEB (World Wide Web) - це система взаємопов'язаних гіпертекстових документів та ресурсів, доступних через Інтернет.

CRUD - чотири основні операції в управлінні даними: Create (створення), Read (читання), Update (оновлення) і Delete (видалення).

Фреймворк - це структурована або стандартизована платформа для розробки програмного забезпечення. Він надає базові структури, компоненти та інструменти для полегшення роботи програмістів при створенні програм або веб-додатків. Фреймворки допомагають вирішувати типові завдання та забезпечують концепції архітектури для ефективного розроблення програмного забезпечення.

БД – база даних;

ВСТУП

У сучасному світі комп'ютерні ресурси стають все доступнішими. Утримання серверів значно здешевлено за рахунок хмарних сервісів. Проте проблема мінімізації витрат на утримання Java WEB серверів залишається актуальною. В наслідок кризових процесів викликаних ковідом та війною багато компаній шукають можливості для зниження витрат.

Метою дослідження є зниження витрат на утримання Java WEB серверів. **Об’єктом дослідження** є Java WEB фреймворки і **предметом дослідження** є методи зниження часу запуску WEB фреймворку та покращення архітектури для зниження витрат на підтримку.

В розділі аналіз предметної області прведено розгляд напрямків що дозволяють здешевити утримання веб серверів. В ньому розглянуті такі можливі зони для покращення як веб фреймворки, інструменти тестування, хмарні сервіси, DevOps підхід та застосування штучного інтелекту.

В результаті аналізу проведеного в першому розділі сформовано два напрямки для досягнення мети - зниження часу запуску та кількості помилок в коді.

В другому розділі описано метод що дозволяє знизити витрати на утримання WEB серверів шляхом застосування спотових віртульних машин. Метод пропонує застосування фреймворку заточеного на мінімальний час запуску що дозволяє застосовувати більш дешеві спотові віртуальні машини.

Далі в третьому розділі описано фреймворк що було реалізовано відповідно до методу. Після цього в четвертому розділі було проведено перевірку його працездатності. З допомогою експерименту в контрольованому середовищі було протестовано час запуску WEB серверу та підтверджено його ефективність.

Після чого було продемонстровано який виграш у зниженню вартості утримання хмарних серверів може бути отримано завдяки застосуванню цього методу.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Аналіз поля для пошуку проблем для розгляду.

Фундаментально здешевлення утримання будь-якого програмного сервісу може бути досягнуто з допомогою оптимізації цього сервісу для більш ефективного використання комп’ютерних ресурсів, застосування інструментів і сервісів з більш дешевими ліцензіями та зниження витрат на підтримку продукту шляхом полегшення його підтримки. Під полегшенням підтримки мається на увазі зниження людино годин на внесення змін та виправлень у сервіс.[1]

В конкретних інструментах зниження витрат на розробку та на утримання відрізняються, проте концептуально можуть досягатись за рахунок тих самих трьох напрямків.

Описані вище 3 глобальні напрямки зниження витрат практично завжди конфліктують один з одним. Наприклад знижуючи витрати за рахунок застосування програмного забезпечення з дешевшими або безкоштовними ліцензіями часто ростуть витрати на підтримку через нижчу якість та нижчі можливості більш дешевого інструменту.

Таким чином при рішенні питання мінімізації цих витрат потрібно у кожному конкретному випадку обирати правильне співвідношення між цими напрямками ураховуючи специфіку конкретної ситуації. Наприклад у випадку розробки програмного забезпечення що в подальшому не буде модернізуватись можна повністю пожертвувати його придатністю до внесення змін та таким чином знизити витрати. Головне правильно ідентифікувати специфіку конкретного випадку, або знайти спосіб мінімізувати наслідки компромісів.

1.1.1 Аналіз можливостей здешевлення підтримки WEB серверів.

Переносячи наш аналіз поля для пошуку проблем на розробку WEB серверів найрозповсюдженішими методами досягнення зниження витрат на підтримку є різноманітні WEB фреймфорки, такі як Spring, JavaServer Faces, Apache Wicket тощо. Ці фреймфорки забезпечують широкі можливості для розробки веб серверів, включаючи управління залежностями, обробку Http запитів, роботу з базами даних, та багато інших можливостей.

Використання фреймворків дозволяє зменшити витрати на підтримку, оскільки значна частина бойлер коду знаходиться під капотом фреймворку та не потребує підтримки з боку розробників. Крім цього саме застосування фреймворків веде до підвищення якості бізнес коду і як наслідок до полегшення його підтримки.[2]

Наступним інструментом зниження витрат на підтримку і також для прискорення розробки є автоматизація тестування. Наприклад, використання інструментів автоматичного тестування, таких як Selenium, Junit тощо. При їх ефективному застосуванні внесення змін в існуючий код не потребує перевірки чи не порушили щось ці зміни. Ці перевірки будуть виконані автоматичними інструментами тестування. У випадку виникнення помилок розробник побачить де з’явились проблеми та не витрачаючи час на їх локалізацію зможе їх швидко виправити.[3]

Ще одним ефективним інструментом для зниження витрат на утримання WEB серверів є хмарні сервіси. Наприклад, Amazon Web Services (AWS) та Microsoft Azure надають можливості для зберігання даних, обчислень та інфраструктури в хмарі. Це зменшує витрати на утримання власної інфраструктури та забезпечує гнучкість управління витратами та потужностями.[4]

Використання DevOps підходу може допомогти забезпечити ефективне управління життєвим циклом продукту, від розробки до розгортання та утримання. DevOps - це підхід до розробки програмного забезпечення, що поєднує в собі розробку (Development) та експлуатацію (Operations) продукту. Використання DevOps дозволяє автоматизувати процеси розгортання, тестування та моніторингу продукту, що зменшує час та витрати на утримання.[5]

Нарешті, використання машинного навчання та штучного інтелекту може бути застосовано щоб покращити якість та ефективність моніторингу роботи WEB серверу. Гарним прикладом є моніторингова система Dinatrace. Вона використовує штучний інтелект для аналізу даних про продуктивність додатків та інфраструктури.

Шляхом аналізу великого обсягу моніторингових даних в реальному часі за допомогою алгоритмів штучного інтелекту, Dinatrace виявляє закономірності та надає практичні висновки, які допомагають оптимізувати продуктивність додатків, вирішувати проблеми та забезпечувати ефективне використання ресурсів. [6]

1.2 Аналіз здобутків в напрямку у Java WEB фреймворків

WEB фреймворк - це програмне забезпечення, яке надає розробникам інструменти та структуру для створення додатків. Вони допомагають забезпечити базовий функціонал, такий як обробка запитів, маршрутизація, взаємодія з базою даних, безпека та інші важливі аспекти розробки WEB додатків.

Їх використання дозволяє розробникам зосередитися на розробці унікального функціоналу своїх додатків, не витрачаючи час і зусилля на реалізацію базових компонентів. WEB фреймворки надають готові інструменти,

бібліотеки та шаблони, що спрощують процес розробки та допомагають підтримувати добру організацію коду.

1.2.1 Аналіз існуючих Java WEB фреймворків

Для розгляду можливостей та особливостей Java WEB фреймворків в роботі було розглянуто 4 най розповсюдженіші фреймворки що часом довели свою ефективність. Далі в розділі на їх прикладі виділено ідеї на яких ґрунтується розробка фреймворків у галузі.

Першим є Spring Framework це один з найпопулярніших та найбільш використовуваних Java WEB фреймворків. Він пропонує широкий набір функціональних модулів, які допомагають в розробці додатків, включаючи модулі для керування залежностями, аспектно-орієнтованого програмування, безпеки, обробки запитів та багато іншого.

Java EE це по суті не конкретний фреймворк а специфікація для серверів-фреймворків.[7] Ця специфікація є стандартом платформи для розробки промислових додатків на Java. Він надає набір специфікацій та програмних інтерфейсів для розробки додатків, включаючи роботу з сервлетами, JSP, EJB, JPA та іншими технологіями. Java EE забезпечує високу масштабованість та надійність додатків, а також інтеграцію з різними технологіями та інфраструктурою.

Сервери-фреймфорки що реалізують Java EE специфікацію по суті конкурують зі Spring Framework представляючи, аналогічний набір можливостей з різницею у філософії підходу. У Java EE прикладний код додатку вставляється в велику працючу машину що поставляється як є, Spring навпаки дає враження, що сам фреймворк це частина додатку.[8] За можливість комфортно та ефективно

його налаштовувати, а також більшої інтуїтивності Spring захопив лідуючу позицію на ринку.

Наступним розповсюдженим фреймворком є JavaServer Faces або JSF. Цей фреймворк для розробки WEB інтерфейсів надає набір компонентів та інструментів для побудови користувацького інтерфейсу, управління станом та взаємодії з сервером. JSF пропонує гнучкість та простоту в розробці багатосторінкових додатків, а також підтримку шаблонів та розширення функціональності.[9]

Для додатків у реактивному стилі на моделі акторів є Play Framework.[10] Він дозволяє створювати швидкі та масштабовані додатки. Play має простий та експресивний синтаксис, підтримку асинхронної обробки запитів, вбудовану систему маршрутизації.

Окрім вищезгаданих для виявлення зон що не були розвинуті було розглянуто можливості фреймфорків Apache Struts, Vaadin, Grails, Wicket. Кожен з них має свої особливості та специфіку, що дозволяє розробникам вибрати найбільш підходящий фреймворк для своїх потреб проте концептуально не пропонують нічого нового в порівнянні з описаними вище фреймворками.

1.2.2 Спільні ідеї на яких ґрунтуються Java WEB фреймворки

Описані вище фреймворки мають дуже широкий набір інструментів та бібліотек для створення різноманітних програмних рішень у різних напрямках таких як повноцінні WEB додатки, RESTful API, платформи для обробки даних та інші типи програмних систем.

В рамках даної роботи нас цікавлять мінімально необхідні можливості для забезпечення роботи фреймворку саме у сфері WEB. Саме принцип роботи цих

модулів та можливості їх покращення для зниження витрат на утримання нас цікавлять. В цьому розділі розглянуто спільні риси згаданих вище фреймворків. Їх перевагу та недоліки що можуть стати місцем для розробки в рамках дисертації.

Перша загальна особливість підходу до їх створення є ідея готової інфраструктури та готових рішень для типових задач з метою зниження часу необхідного на розробку та її вартості.

Другим принципом, що доповнює перший є філософія уніфікації типових рішень та формалізоване застосування найкращих архітектурних практик. Саме в рамках екосистеми Java створються та постійно оновлюються специфікації для типових рішень. Наприклад специфікація JPA, що регламентує програмний інтерфейс роботи з базою даних. Окрім цього прикладу існують специфікації що регламентують буквально все – роботу в мережі за різними протоколами, обробку виключень, роботу з під'єднанням до бази даних, контролю інверсії залежностей і ще безмежну кількість питань.

Третім принципом є відносно низька ціна обчислювальних потужностей у порівнянні з вартістю людських ресурсів необхідних для розробки прикладних програм. Виходячи з нього де це можливо ефективність з точки зору використання обчислювальних потужностей жертвується на користь швидкості і простоті розробки.

Останній принцип що можна виділити є орієнтованість в першу чергу на швидкість роботи та обробки запитів у порівнянні з часом старту серверу. А також жертва за для цього обчислювальними ресурсами. Наприклад застосування великих кешів для запитів до сторонніх систем та баз даних з метою мінімізації часу відповіді на запит користувача.

1.2.3 Структура Java WEB фреймворків

З точки зору структури всі вони ґрунтуються на принципах модульності та інкапсуляції. Модульність визначається як здатність розділити складний системний компонент на окремі незалежні модулі, які можна розробляти, тестувати та підтримувати окремо.

Один із ключових аспектів модульності - це здатність контролювати залежності між модулями. За допомогою інкапсуляції та використання інтерфейсів, модулі можуть взаємодіяти між собою, не розкриваючи внутрішніх деталей своєї реалізації. Це дозволяє забезпечити високий рівень абстракції та знизити залежність між модулями, що сприяє гнучкості та стабільності системи.

Концепція інкапсуляції полягає у приховуванні внутрішньої реалізації модулів та наданні зовнішньому світу лише необхідного інтерфейсу для взаємодії.

Інкапсуляція дозволяє захистити внутрішні деталі реалізації модуля від несанкціонованого доступу або неконтрольованої зміни.

За допомогою інкапсуляції, модулі можуть взаємодіяти між собою через публічні інтерфейси, які визначаються для зовнішнього використання. Інкапсуляція також сприяє підтримці інтерфейсної спадковості, що дозволяє замінювати одну реалізацію модуля на іншу без впливу на код, який використовує цей модуль.

Ґрунтування на принципах інкапсуляції та модульності в купі дає можливості відносно ефективної мінімізації жертв у вигляді жадібності до ресурсів шляхом можливості застосування лише тих модулів що є необхідними для конкретного проекту. Також можливість легко додавати та міняти ці модулі у випадку потреби.

Проте сама пособі розробка за таким підходом веде до додаткових втрат обчислювальних потужностей через довші ланцюги викликів. А також через не ідеальне розташування фрагментів коду для забезпечення модульності.

По при розділення фреймворків на модулі кожен модуль все одно є величезним та включає дуже багато функціоналу що не завжди потрібен. Це вимагає зайвих системних ресурсів. Вирішення цієї проблеми може дати можливість знизити втирати на утримання WEB серверів.

Другим недоліком що підвищує витрати на підтримку є дуже велика гнучкість у способах написання коду. Розробники фреймворків намагаються дозволити використовувати функції що вони надають в максимально можливій кількості сценаріїв.

Це дає можливість більш швидко створювати код і одночасно допускати більше архітектурних помилок. Оскільки нас цікавить зниження вартості саме підтримки, а не розробки цей напрям також є перспективним для подальшого дослідження.

1.2.4 Мінімально необхідні можливості WEB фреймфорку

Маючи на меті знайти спосіб знизити вартість утримання серверів логічним є проаналізувати які модулі WEB фреймворків є мінімально необхідними саме для забезпечення WEB серверів. В подальшому орієнтуючись на ці можливості можна за рахунок їх прибирання чи обмеження досягти зниження витрат на обчислювальні потужності.

Безперечно коректна архітектура є необхідною для ефективної підтримки код додатку і відповідно інструменти що її забезпечують є необхідними. Для

забезпечення гнучкої архітектури необхідно дотримуватися кращих принципів проектування, в першу чергу принципів SOLID.[11]

Для забезпечення принципу інверсії залежностей фреймворк повинен мати контейнер інверсії залежностей. Він присутній в усіх розглянутих вище фреймворках і дозволяє розробникам не задумуватись про механізм визначення залежностей між компонентами.

Він керує життєвим циклом об'єктів і дозволяє зосередитись на бізнес логіці додатку, а не на управлінні ресурсами та залежностями.

Стандартною для сучасних java WEB серверів є архітектура MVC і відповідно модулі що її забезпечують є також обов’язковими. Інструменти цих модулів дають розробникам можливість ефективно та гнучко керувати кінцевими точками, їх обробниками, моделями та представленнями.[12]

Наступним необхідним модулем що підтримують всі фреймворки є модуль роботи з базою даних. Його призначення полягає у наданні можливості ефективно взаємодіяти з базами даних. Ключовою функцією його є контроль за транзакціями на рівні бізнес логіки. Без цього організувати гнучку роботу з транзакціями просто не можливо.

Підтримка ОРМ дозволяє ліквідуючи необхідність писати бойлер код для перетворення запитів в сутності і навпаки але не є не обхідним функціоналом модулю роботи з базою даних. Ці функції можуть бути реалізовані з допомогою технології JDBC.[13]

Останнім необхідним для функціонування є модуль безпеки додатків. До функцій якого власне відноситься забезпечення безпеки серверу, включаючи аутентифікацію, авторизацію, керування сеансами та захист від атак.

1.3 Інструменти автоматичного тестування

Інструменти автоматичного тестування - це програмні засоби або фреймворки, що використовуються для автоматизації процесу тестування програмного забезпечення. Вони дозволяють знизити втирати на підтримку даючи можливість тестувальникам та розробникам швидко та ефективно виконувати автоматичні тести, забезпечуючи повторюваність, та точність результатів.

Ці інструменти значно прискорюють процес тестування, зменшують зусилля, пов'язані з ручним тестуванням, та забезпечують високу якість продукту шляхом раннього виявлення помилок та проблем. Завдяки цьому знижуючи вартість підтримки.[14]

У таблиці 1.1 представлено види тестування та найпопулярніші інструменти для цього типу автоматичних тестів. З кількості напрямків тестування та кількості інструментів для цих напрямків видно що ця тема розвинена досить глибоко.

Усі розповсюджені види тестування покриті інструментами. Розвиток цього напрямку немає перспектив для зниження витрат на утримання WEB серверів. Можливо існуючі інструменти можна було б покращити щоб знизити витрати на створення тестів, але саме на вартість підтримки це не впливає.

Таблиця 1.1 – Інструменти автоматичного тестування за його видами

Вид автоматичного тестування	Інструменти що його забезпечують
Функціональне тестування	Selenium WebDriver, Appium, TestComplete, Cucumber
Регресійне тестування	Selenium WebDriver, Apache JMeter, TestComplete, Cucumber, Jenkins
Юніт-тестування	JUnit, NUnit, PyTest, PHPUnit, TestNG
Тестування інтерфейсу користувача (UI)	Selenium WebDriver, Cypress, TestComplete, Appium

Продовження таблиці 1.1

Вид автоматичного тестування	Інструменти що його забезпечують
Тестування продуктивності	Apache JMeter, Gatling, LoadRunner, Locust, BlazeMeter
Тестування безпеки	OWASP ZAP, Burp, Suite, Nessus, Acunetix, Wireshark
Тестування сумісності	BrowserStack, Sauce Labs, CrossBrowserTesting, LambdaTest
інтеграційного тестування	Apache JMeter, SoapUI, RestAssured, Karate, Citrus Framework
API-тестування	Postman, Insomnia, Newman, REST-assured, Guzzle
Pact тестування	Pact (Pact-JS, Pact-JVM, Pact-Ruby) Postman + Newman з використанням Pact Collection Runner, Spring Cloud Contract, Pactflow

1.4 Хмарні сервіси

Використання хмарних сервісів дозволяє суттєво знизити витрати на утримання серверів. По перше інфраструктура в хмарі дозволяє позбутись витрат та ризиків пов’язаних з адмініструванням цих серверів. Стає не потрібним власний IT-персонал для утримання сервера, його підтримки, забезпечення безпеки та

резервного копіювання. По друге зникає необхідність в одномоментних стартових вкладенях в приміщення та серверне обладнання.[15]

Най більшими провайдерами хмарних послуг на сучасному ринку є Amazon Web Services (AWS), Microsoft Azure або Google Cloud Platform (GCP). Вони надають доступ до готової інфраструктури і безлічі різноманітних PaaS та SaaS рішень, які вони утримують і оновлюють.[16]

За допомогою хмарних сервісів також знижується вартість масштабування. Бо по-перше в кілька кліків можна збільшити потужності северу або налаштувати це збільшення автоматично. По-друге зараз хмарні сервіси надають можливості динамічного масштабування що дозволяє знизити потужності тоді коли в них потреби немає і таким чином знизити витрати на утримання.

У ситуаціях коли потрібна географічна розподіленість серверів витрати на утримання власних серерів різко зростають. Для забезпечення цього росту потрібні нові приміщення, новий персонал для підтримки та юридичний персонал оскільки у різних юрисдикціях будуть різні вимоги до, наприклад, протипожежної безпеки.

Хмарні сервіси мають центри обробки даних у різних регіонах світу. Це дає можливість розмістити сервер одночасно в різних куточках світу практично за ціною просто додання нових комп’ютерних потужностей. [17]

1.4.1 Тарифікація віртуальних серверів у хмарі

В умовах стрімкого розвитку хмарних технологій провайдерами, була розроблена справедлива та ефективна політика тарифікації. Особливості тарифікації можуть дати поле для здешевлення утримання шляхом оптимального застосування тарифів.

На вартість хмарних послуг впливають різні фактори. По перше це види ресурсів. Різні типи ресурсів, такі як обчислювальні потужності, обсяги оперативної пам'яті, мережеві ресурси, зберігання даних, мають різну вартість та для різних задач можуть бути потрібні у різних пропорціях.[18]

Другим фактором є рівень якості сервісу SLA (Service Level Agreement). Що в даному контексті включає в себе такі фактори:

- надійність – Визначає мінімальний час доступності сервісу (наприклад, 99,9% доступності), який вказує, наскільки стабільно працює хмарне середовище.
- продуктивність – Задає мінімальні вимоги до часу обробки запитів, завантаження інформації тощо.
- безпека – Описує заходи, прийняті хмарним провайдером для захисту даних і запобігання злому або несанкціонованому доступу.
- підтримка – Визначає рівень підтримки, який надає хмарний провайдер, і час відгуку на звернення або вирішення проблем.

Також на ціни тарифів впливає географічне розташування. Його вплив на ціну йде через різні правові та економічні обмеження в різних країнах або регіонах, а також різну вартість енергоресурсів та людських ресурсів.

Для забезпечення різних ситуацій хмарні провайдери пропонують три типи тарифікації, це планована тарифікація, оплата за замовленням та комбінована.[19,20,21]

Планована тарифікація базується на фіксованому плані, де користувачі хмарних послуг обирають певний обсяг ресурсів та період їх використання. Цей підхід надає простоту та прогнозованість для користувачів, але може бути не ефективним, оскільки не дозволяє гнучко реагувати на потреби користувачів що змінюються.

Тарифікація за споживання полягає у вимірюванні ресурсів, які реально використовуються користувачами, і нараховує вартість відповідно до цих показників. Цей підхід забезпечує більшу гнучкість і справедливість, але вимагає більш складних механізмів моніторингу та обліку.

На одиницю ресурсу цей підхід є дорожчим за планові тарифи тому його застосування дає вигоду лише у випадку змін у потребі ресурсів в рамках часового періоду. Застосування планової тарифікації вимагає обрання плану з запасом що зазвичай буде дорожче ніж тарифікація за споживання.[19,20,21]

Комбінований підхід поєднує переваги планованої тарифікації та тарифікації за споживання. Користувачі можуть обирати базовий план з фіксованими обсягами ресурсів, але також матимуть можливість доплачувати за додатково використані ресурси вже по факту споживання.[19,20,21]

Існує ще однин особливий вид тарифікації – спотовий. Спотові віртуальні машини є особливим типом обчислювальних ресурсів у хмарному середовищі. Замість фіксованої ціни або пакетів ресурсів, спотові віртуальні машини пропонуються на аукціонах. Користувачі можуть ставити свої пропозиції щодо ціни, яку вони готові заплатити за обчислювальні ресурси. Це робить спотові віртуальні машини більш доступними з економічної точки зору, але також створює невизначеність щодо доступності та вартості ресурсів.[19,20,21]

Тарифікація спотових віртуальних машин відбувається за допомогою аукціонних механізмів. Ці механізми визначають вартість ресурсів на основі заявлених пропозицій користувачів. Аукціон може мати різні типи, такі як аукціон з підвищенням ціни (англ. English auction), де обчислювальні потужності надаються користувачу що запропонує більшу ціну, або аукціон зі зниженням ціни (англ. Dutch auction), де ціна спочатку встановлюється високою і поступово знижується до того моменту, коли користувач прийме умови.

Однією з основних особливостей тарифікації спотових віртуальних машин є невизначеність доступності ресурсів. Оскільки вартість ресурсів залежить від аукціонних пропозицій і попиту, користувачам можуть бути недоступні віртуальні машини у певний час або їх вартість може змінюватися в залежності від потреб інших користувачів.[19,20,21]

Для ефективного використання спотових віртуальних машин, користувачам необхідно розробляти стратегії планування та оптимізації вартості. Це може включати моніторинг ринку аукціонів, прогнозування вартості ресурсів та розробку алгоритмів автоматичного запуску та зупинки спотових машин залежно від потреб.

Важливо зазначити, що використання спотових машин пов'язане з ризиком, оскільки хмарні провайдери мають право зупиняти спотові машини у випадку, якщо їх потрібно зарезервувати для користувачів, які платять повну ціну. Тому користувачі повинні бути готові до можливості тимчасового втрати доступу до спотових машин.[19,20,21]

Таким чином тарифікація спотових віртуальних машин має свої унікальні виклики. Проте одночасно ці машини можуть бути дешевшими на десятки відсотків у порівнянні з плановими і ефективне їх застосування може дуже суттєво знизити вартість утримання серверу.

1.5 Інструменти DevOps підходу

DevOps є підхідом до розробки програмного забезпечення, який поєднує розробку (Development) і експлуатацію (Operations). Він дає покращення ефективності та швидкості процесу розробки коду, його впровадження і підтримки. Основна мета DevOps полягає у автоматизації процесів та

використанні інструментів, що дозволяють швидко та надійно впроваджувати зміни в програмному забезпеченні. Також у забезпеченні спільної відповідальності команд розробки та експлуатації[22]

В контексті зниження витрат на підтримку серверів, DevOps підхід є дає виграш за рахунок автоматизації процесів. Таких як розгортання серверів, налаштування середовищ, моніторинг та документування роботи. Таким чином знижуючи витрати через ручне втручання та зменшення кількікості помилок. Крім цього завдяки постійній інтеграції та постійному впровадженню (CI/CD) знижується вартість впровадження нових функцій і прискорюється потрапляння змін на користувацькі сервери.

До інструментів що це забезпечують відноситься:

- контроль версій (Version Control System, VCS): Найпопулярнішими VCS є Git та SVN.
- системи автоматичної збирання (Continuous Integration, CI): Популярні CI-системи включають Jenkins, Travis CI, CircleCI, GitLab CI/CD та інші.
- інструменти конфігурації (Configuration Management): Puppet, Chef, Ansible та SaltStack є прикладам інструментів для автоматизації налаштування та управління інфраструктурою.
- контейнеризація: Docker забезпечує стандартизацію та уніфікацію навколо впровадження додатків та спрощує розгортання та масштабування середовищ.
- моніторинг та документування: Інструменти моніторингу, такі як Nagios, Zabbix, Prometheus, допомагають відстежувати стан і продуктивність серверів, додатків та інфраструктури. Інструменти документування, такі як ELK Stack (Elasticsearch, Logstash, Kibana), Splunk та Graylog.

- інструменти управління: Kubernetes є одним з провідних інструментів для управління контейнерами.

З переліку інструментів та документації що описує їх можливості видно що описаний у першому абзаці напрямок є широко вивченим, інструментально підтриманим та ефективним, і як наслідок низько перспективним для розробки та пошуку можливостей для зниження витрат в рамках теми даної дипломної роботи.

1.6 Використання штучного інтелекту

Впровадження штучного інтелекту для зниження витрат на утримання може виявитись корисним. Нижче наведено способи застосування штучного інтелекту що можуть знизити витрати на утримання WEB серверів.

По перше це автоматизація процесів. Деякі рутинні завдання, такі як моніторинг, підтримка безпеки та аналіз журналів можуть бути автоматизовані з допомогою штучного інтелекту.

По друге це прогнозування з його допомогою попиту на серверні ресурси. Це дозволяє ефективно розміщувати ресурси та збільшувати їх кількість лише в тих випадках, коли це дійсно необхідно.

Окрім зазначеного вище застосування штучного інтелекту дозволяє аналізувати великі обсяги даних про використання ресурсів та створювати звіти з цих даних. Це допомагає ідентифікувати витрати та точки оптимізації.

Незважаючи на потенційні переваги, впровадження штучного інтелекту також має обмеження. Побудова систем штучного інтелекту може бути складним і витратним процесом. Штучний інтелект потребує навчання та постійного вдосконалення. Це вимагає додаткових витрат на навчання персоналу та

підтримку системи. Крім цього інтеграція з існуючих системам з штучним інтелектом може бути складною та значних ресурсів.

1.7 Висновки розділу

У сфері тестування ситуація є не перспективною для розробки. Бо, по перше це не впливає на пряму на вартість у тримання, а по друге, бо тема є дуже глибоко вивченою. В першу чергу це пов’язано з тим що тестування присутнє при розробці будь-якого ПЗ на всіх етапах розробки. Тому розробка цього напрямку ведеться усією сферою розробки ПЗ, а не лише WEB серверів.

Запровадження DevOps підходу для зниження витрат на підтримку є високо ефективним. Автоматизація процесів розгортання, налаштування, моніторингу та логування дозволяє зменшити час і ресурси, витрачені на рутинні операції, і тим самим знизити витрати за рахунок зменшення ручного втручання і кількості помилок в коді що потрапляє користувачу. Крім того, постійна інтеграція та постійне впровадження (CI/CD) сприяють швидкому впровадженню змін і нових функцій, що також знижує вартість утримання серверів.

З аналізу видно що вже створено широкий спектр інструментів для DevOps. Вони включають системи контролю версій, системи автоматичної збирання, інструменти конфігурації, контейнеризацію, моніторинг та логування, а також інструменти оркестрування та оркестратори контейнерів. Ці інструменти є добре дослідженими, широко підтримуваними і довели ефективність у практичному застосуванні.

У зв'язку з цим, можна зробити висновок, що напрямок застосування DevOps підходу у розробці програмного забезпечення не є новим і має досить високий рівень зрілості та ефективності. Тому для даної дипломної роботи нема

перспектив для додаткового дослідження та пошуку способів зниження витрат за допомогою DevOps підходу.

Використання штучного інтелекту також не є перспективним для розвитку в рамках даної дисертації. Його застосування дає можливість знизити витрати за рахунок автоматизації рутинних завдань моніторингу, підтримці безпеки та аналізі журналів. Але одночасно таке застосування призводить до додаткових витрат на обчислювальні потужності та посуті може дати ефективний результат тільки в дуже великих системах, а дана робота орієнтована на низько бюджетні не великі сервери.

Аналогічно, штучний інтелект може допомогти в прогнозуванні попиту на серверні ресурси та проводити аналіз великих обсягів даних про використання ресурсів дозволяє ідентифікувати витрати та точки оптимізації. Це може призвести до зниження витрат проте саме в рамках типових задач цільової аудеторії цієї розробки не дасть результату.

Таким чином, хоча використання штучного інтелекту має потенціал для зниження витрат на утримання серверів, ретельно розглянувши специфіку нашої задачі поглиблення цієї сфери не є ефективним та перспективним. Особливо у порівнянні з потенціалом комбінації нової філософії вреймворку та спотових віртуальних машин розглянутий раніше у цьому розділі.

З розглянутих можливостей WEB фреймворків зрозуміло що ця сфера є глибоко дослідженою, має широкі можливості, але повністю орієнтована на мінімізацію витрат на створення нового коду часто жертвуючи легкістю його підтримки та завжди жертвуючи комп’ютерними ресурсами в силу того що в сучасному світі обчислювальні потужності вважаються дешевими.

Також в абсолютний пріоритет ставиться швидкість роботи запущеного додатку по відношенню до швидкості запуску. Це пов’язано з тим що

користувацький досвід вимагає погоні за максимально швидкою роботою додатку і одночасно сучасні сервери та провайдери хмарних серверів пропонують досить високу надійність обладнання що робить вимушені перезапуски додатку дуже рідкісним явищем.

Проте в рамках теми даної роботи ми розглядаємо роботу WEB фрейморків виключно з точки зору мінімізації витрат на утримання. І в цьому питанні концепція існуючих фреймворків або протирічить нашій меті або не допомагає їй. Адже максимальна швидкість роботи означає що фреймворк робить всі можливі розрахунки заздалегідь, під час запуску задовго до того моменту коли вони знадобляться, якщо взагалі знадобляться. На це відповідно витрачаються додаткові обчислювальні ресурси.

З розгляду можливостей хмарних сервісів на перший погляд здається що сфера дуже розвинута в напрямку зниження витрат на підтримку та не є перспективною в рамках даної роботи. Проте звернувши увагу на різні цінові політики, а саме на дуже значні поступки в ціні для спотових серверів. Можна побачити не очевидну але ефективну можливість для зниження витрат у комбінації з застосування фреймворку орієнтованого на мінімальний час запуску.

Висока швидкість старту дозволила б мінімізувати негативний вплив від того що віртуальний сервер може бути забрано провайдером. А дуже низька ціна застосування спотових тарифів дозволила б ефективно знизити вартість утримання.

Це додає додаткових перспектив напрямку розробки фреймфорку з ідеєю протилежною прийнятій в даний час. Фреймворк який буде робити мінімум обчислень під час запуску не тільки знизить кількість необхідних обчислювальних ресурсів, а й дасть можливість застосувати спотові тарифи для тих задач де раніше

це було неможливо. Ця архітектурна комбінація може дозволити значно знизити витрати на утримання серверу.

В рамках аналізу фреймворків було зазначено занадто високу гнучкість в деяких питаннях що веде до того що недосвідчені розробники можуть з її допомогою робити зайві архітектурні помилки. Результат даної дисертації Результут даної дисертації буде в першу чергу цікавим проектам з обмеженими фінансовими можливостями. Це означатиме що на цих проектах вірогідно буде більша кількість низько досвідчених розробників.

Тому третім ефективним аспектом в рамках змін підходу до фреймворку є зниження свободи у порівняння з аналогами щоб змусити розробників писати код що відповідає правильній архітектурі. Це дозволить знизити витрати на підтримку кодової бази що є частиною витрат на утримання серверів.

2 МЕТОД ПОБУДОВИ ФРЕЙМВОРКА ДЛЯ ЗНИЖЕННЯ ВАРТОСТІ УТРИМАННЯ JAVA WEB СЕРВЕРІВ

2.1 Ідеї на яких ґрунтується розробка та розвиток існуючих Java WEB фреймворків

Почнемо аналіз методу прискорення запуску фреймворку з розгляду причин чому та як фреймворки жертвують швидкістю запуску, і як це може бути змінено в розробці та чим доведеться пожертвувати за для досягнення більш швидкого запуску.

Найрозповсюдженішими фреймворками для є Spring, Django, JavaServer Faces, Spark Java та сервери що реалізують специфікацію Jakarta EE наприклад glassfish. Кожен з цих фреймворків має свою архітектуру та набір компонентів, які впливають на швидкість запуску серверу. Ця модульність пов’язана з бажанням розробників по можливості полегшити громіздкі фреймворки та знизити ресурси що він вимагає і час необхідний для запуску.

Проте, як вже було зазначено в аналізі предметної області, про швидкість запуску розробники турбуються лише коли її підвищення не веде до компромісів по іншим параметрам. Тому не дивлячись на модульність завантаження та ініціалізація цих компонентів займають значний час, що призводить до повільного запуску сервера.

В широкому сенсі у фреймворках швидкістю запуску жертвують для обробки усього що можна обробити перед запуском:

- формування запитів в базу даних.
- аналіз та будування дерева залежностей.
- жадібне завантаження при старті усіх компонентів фреймворку.

- загалом завантаження та виконання всього що можна до запуску щоб потім додаток працював швидше.

2.2 Процес запуску серверів з існуючими фреймворками

На рисунку 2.1 продемонстровано узагальнену діаграму запуску серверу з застосуванням фреймворку Spring. Спочатку Spring Framework зчитує конфігураційні файли додатку. Це може бути, наприклад, файл application.properties або application.yml, де зберігаються налаштування додатку, такі як параметри підключення до бази даних, порти, логування тощо.

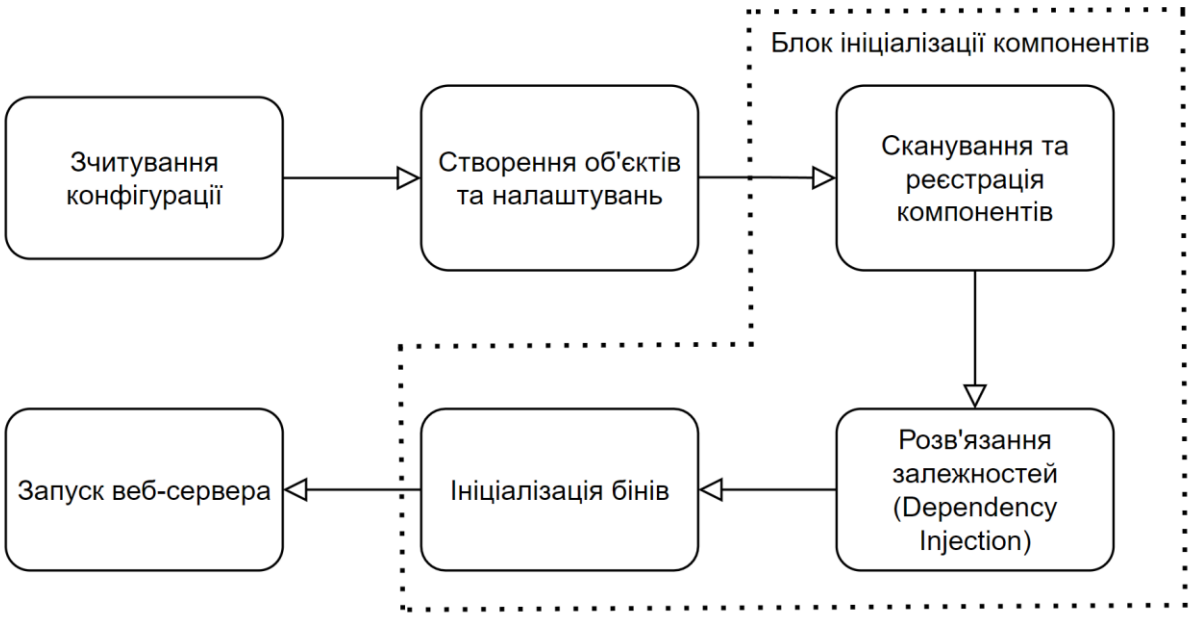


Рисунок 2.1 - Процес запуску програми Spring Boot

Після цього створюється спеціальний контейнер інверсії залежностей у випадку Spring відомий як інтерфейс ApplicationContext. Цей контейнер використовується для управління компонентами додатку, що включають контролери, сервіси та інші компоненти. Контекст додатку зберігає та керує всіма компонентами, які використовуються в додатку.

Далі відбувається сканування всіх класів у проекті та його залежностях для знаходження тих класів, що відповідають критеріям завантаження та обробки до старту. Це власне компоненти та різні модифікації над ними по типу накладання транзакцій, обгорток та підключення аспектного коду. У випадку Spring це класи помічені анотаціями @Component, @Service, @Controller тощо. Коли він знаходить такі класи, вони ініціалізуються як компоненти та додаються в контекст додатку.[23]

Наступним кроком з допомогою механізму інверсії керування (IoC) відбувається впровадження залежностей між компонентами. Фреймворк самостійно управляє створенням та ін'єкцією залежностей між класами, що дозволяє легко взаємодіяти з різними компонентами.

Після цього відбувається ініціалізація компонентів. Якщо компонент має методи ініціалізації, вони виконуються автоматично після того, як він створений і всі його залежності встановлені.

Нарешті останнім кроком йде запуск серверу коли усі компоненти та їх залежності ініціалізовані, і контекст додатку готовий. Spring застосовує наприклад Tomcat або Jetty. Після чого починають прийматись і оброблятись WEB-запити.

2.3 Час та необхідність кроків процесу запуску серверу

Час, необхідний для старту сервера, може суттєво відрізнятися в залежності від різних факторів, таких як розмір та складність додатку, використовувані бібліотеки, залежності, а також характеристики обчислювального обладнання.

На початку, сервер має завантажити конфігурацію, яка може зайняти від декількох мілісекунд до декількох секунд. Далі, необхідно створити контекст

додатку порядок часу що йде на цей крок також кілька мілісекунд. Обидва кроки практично не залежать від розміру додатку.

Далі йдуть кроки пов’язані з управлінням компонентами та їх модифікацією. Сканування та реєстрація компонентів може зайняти від декількох мілісекунд до декількох секунд. Розв'язання залежностей зазвичай займає дуже мало часу, але для великих додатків зі значною кількістю компонентів та залежностей між ними, час може збільшитися від декількох мілісекунд до декількох секунд. Чим більше компонентів і складніша їх конфігурація тим більше час.

Нарешті, запуск сервера залежить від використовуваного сервера та обладнання. Зазвичай цей крок займає кілька секунд, але для дуже великих додатків зі складними серверами та великою кількістю ресурсів, час старту може збільшитися.

З усіх кроків найбільший час займає блок кроків по ініціалізації компонентів та крок запуску серверу. Кроки зчитування конфігурації та створення самого контексту по є обов’язковими. Оскільки не зчитавши конфігурацію не можливо знати наприклад на якому порту запустити додадотк, а без конктексту по суті фреймворк не почав працювати та відповідно сервер ще не готовий до роботи. Крок запуску серверу також є необхідним і його час не залежить від фреймворку. А ось кроки пов’язані з обробкою компонентів можна взагалі винести з запуску прискоривши старт.

2.4 Зміни у процесі запуску додатку та компроміси до яких вони ведуть

В аналізі предметної області як одну із ідей мінімізації часу запуску було сформовано відмову від надлишкового функціоналу та відмову від виконання при старті тих дій котрі можна зробити пізніше. Далі ми детально розглянемо від чого

було вирішено відмовимось при старті та на які компроміси для цього довелось піти.

2.4.1 Контейнер інверсії залежностей

Найбільші втрати часу під час запуску пов’язані з IOC контейнером і відбуваються під час створення дерева компонентів з конфігурації та наступного їх завантаження і обробки.

Принцип інверсії залежностей (Dependency Inversion Principle) - є ключовим в області розробки програмного забезпечення. Виконання цього принципу дозволяє досягти модульності, розширюваності та можливості пере використання коду. Це також дозволяє зменшити кількість прямих посилань між класами та полегшити тестування і підтримку. [24]

IOC контейнер - це спосіб реалізації принципу інверсії залежностей з допомогою підходу впровадження залежностей або «Dependecy Injection». Він має забезпечувати управління залежностями між компонентами, створення та впровадження об'єктів, вирішення залежностей та надання компонентам необхідних ресурсів.[24] За допомогою IOC контейнера здійснюється автоматичне, з точки зору користувача фреймворку впровадження залежностей.

Таким чином ми не можемо відмовитись від такого базового функціоналу і мусимо обмежитись мінімізацією його впливу на запуск.

В архітектурі нашого контейнеру для мінімізації часу було принципово відкинуто створення дерева залежностей. Завдяки цьому зникає проходження по всіх компонентах та побудова графу їх залежностей. Натомість залежності вирішуються за жадібним принципом по ходу створення об’єкту.

За виграш у швидкості цей підхід несе ряд недоліків. Підвищується складність контролю за залежностями та виправлення помилок що можуть виникати в наслідок конфліктів. Стає не можливою переверірка коректності конфігурації залежностей на старті, у випадку наявності помилок у конфігурації про них стане відомо лише у момент коли конкретний компонент знадобиться. Також це веде до неможливості існування кругових залежностей, оскільки створення вкладених залежностей відбувається жадібно.

Жертва циклічними залежностями є корисною оскільки це одна із тих свобод що дозволяються швидше написати код але потім призводять до складностей у його підтримці. І віт таких рішень було вирішено відмовитись. Відсутність такої можливості веде до недопущення частини архітектурних помилок етапі розробки що знижує вартість вартість утримання.

Відсутність дерева дозволяє нам повністю виключити всі кроки пов’язані з обробкою компонентів зі старту додатку. Проте в залежності від ситуації може бути потрібно виконати деяку обробку на старті. Наприклад завантажити компоненти патерну Singletone. Це питання можна вирішити без компромісів створивши додаткові конфігураційні властивості. Ґрунтуючись на яких контейнер або не буде виконувати зовсім ніякої обробки при старті або виконувати ті обробки що були ввімкнуті користувачем і нічого зайвого.

Таким чином користувач буде вільний обирати у кожній конкретній ситуації, чим він готовий пожертвувати для прискорення старту, а що бажає залишити для прискорення роботи вже запущеного додатку.

2.4.2 Робота з базою даних

Стек сучасних фреймворків завжди включає підтримку функцій для роботи з базами даних. До основних функцій у цьому напрямку відносяться:

- абстракція бази даних – високорівневий API для взаємодії з базою даних, що дозволяє розробникам працювати з даними на рівні об'єктів або моделей замість безпосередньої маніпуляції SQL запитами. Це спрощує розробку та підтримку додатків.
- ORM (Object-Relational Mapping) – дозволяє взаємодіяти з базою даних через об'єктно-орієнтований підхід. ORM забезпечує перетворення між об'єктами додатку та таблицями бази даних, дозволяючи розробникам працювати з об'єктами даних замість написання складних SQL запитів.
- міграції баз даних – інструменти для управління версіями схеми бази даних. Це дозволяє легко змінювати структуру бази даних та забезпечує міграцію даних при оновленні версії додатку.
- підтримка транзакцій – механізми для управління транзакціями бази даних. Це дозволяє забезпечити цілісність даних та гарантує, що операції здійснюються атомарно і консистентно.
- кешування даних – підтримка кешування даних, що дозволяє зберігати результати запитів до бази даних в оперативній пам'яті. Це покращує продуктивність додатків шляхом зменшення навантаження на базу даних та зменшення часу відповіді.
- підтримка різних баз даних – можливість використовувати бази даних без необхідності писати специфічний код для взаємодії з кожним типом бази даних.

Перечисленні вище можливості є дуже зручними наприклад на рисунку 2.2 зображено весь необхідний код для отримання користувача за ім`ям або

прізвищем з застосуванням фреймворку Spring. На рисунку 2.3 представлено код що потрібно написати для тої самої операції отримання користувача за іменем без застосування фреймфорку.

```java
@Repository
public interface UserRepository extends JpaRepository<User, Long> {
    List<User> findByFirstName(String firstName);
    List<User> findByLastName(String lastName);
}
```

Рисунок 2.2 – Код репозиторію для сутності користувача

```java
public class UserDao {
    private final DataSource dataSource;
    public UserDao(DataSource dataSource) {
        this.dataSource = dataSource;
    }
    public List<User> getUsersByFirstName(String firstName) {
        try (Connection connection = dataSource.getConnection();
             PreparedStatement statement = connection.prepareStatement("SELECT * FROM users WHERE first_name = ?")) {
            statement.setString(1, firstName);
            try (ResultSet resultSet = statement.executeQuery()) {
                List<User> users = new ArrayList<>();
                while (resultSet.next()) {
                    User user = new User();
                    user.setId(resultSet.getLong("id"));
                    user.setFirstName(resultSet.getString("first_name"));
                    user.setLastName(resultSet.getString("last_name"));
                    users.add(user);
                }
                return users;
            }
        } catch (SQLException e) {
            // Обробка винятків
        }
        return Collections.emptyList();
    }
}
```

Рисунок 2.3 – Код DAO для сутності користувача без застосування ORM

Різниця очевидна, фреймворки роблять роботу з базами даних більш простою з точки зору програміста, надаючи розробникам потужні інструменти та абстракції, що спрощують доступ до даних, забезпечують гнучкість та полегшують розширення додатків.

Проте підтримка усіх цих можливостей вимагає значних витрат ресурсів при запуску програми. Стосовно прикладу наведеного вище, фреймворк при запуску

має прочитати назви методів інтерфейсів помічених анотацією @Repository, провалідувати їх та згенерувати код для роботи з базою даних.

Це протирічить максимально швидкому старту і тому від більшості цих можливостей нам доведеться відмовитись. З усіх перерахованих можливостей наша розробка буде підтримувати лише автоматизацію транзакцій. Підтримка транзакцій при лінивій ініціалізації вимагає мінімальної кількості ресурсів при запуску серверу та є необхідною для створення зрозумілої та гнучкої бізнеслогіки.

Зниження часу запуску буде досягнуте за рахунок винесення ініціалізації всього коду та обгорток для підтримки транзакцій із етапу запуску додатку. Та відсутності можливостей автоматичного створення запитів та перетворення відповідей бази даних у об’єктно орієнтований вид. Це приведе до зниження швидкості розробки проте на утримання не вплине.

Підтримка роботи з різними базами даних буде підтримуватись за рахунок того що буде застосовуватись технологія JDBC, що сама надає абстракцію для розділення коду специфічного для різних реляційних баз даних та бізнеслогіки. І оскільки це досягається зміною в конфігурації конкретної реалізації дравйверу ця гнучкість жодним чином не випливатиме на швидкість запуску серверу.[13]

2.4.3 WEB модуль

Загально прийнятою практикою для організації обробки різних мережевих запитів є застосування патерну команда часто в комбінації з архітектурним шаблоном MVC.[25] Цей патерн є одним із шаблонів проектування, який використовується для впорядкування обробників за ключами. Він дозволяє ізолювати виконання запитів від їх ініціаторів, що робить структуру програми більш гнучкою та сприяє зменшенню залежностей між класами.

Основна ідея патерну "Команда" полягає в тому, що запити передаються об'єктам команд, які самостійно вирішують, як і коли обробити запит. Це дозволяє ізолювати клієнта від виконавця. Така ізоляція дозволяє легко змінювати та додавати нові команди без впливу на викликача.[26]

В контексті обробки запитів, патерн "Команда" є особливо корисним, оскільки викликач може бути здатен взаємодіяти з різними командами, не знаючи про їх деталі виконання. З цим підходом, логіка обробки запитів стає гнучкою і розширюваною, а код стає більш читабельним і легше підтримуваним.

На рисунку 2.4 зображено діаграму класів класичної реалізації партерну команда.[26] У випадку реалізації у фреймворку для обробки веб запитів патерн було модифіковано додатковими можливостями технології Java Reflection API.[27] Invoker реалізується в фреймворку він використовуючи в якості ключа стрічку адреси запиту визначає яка команда має його обробляти.

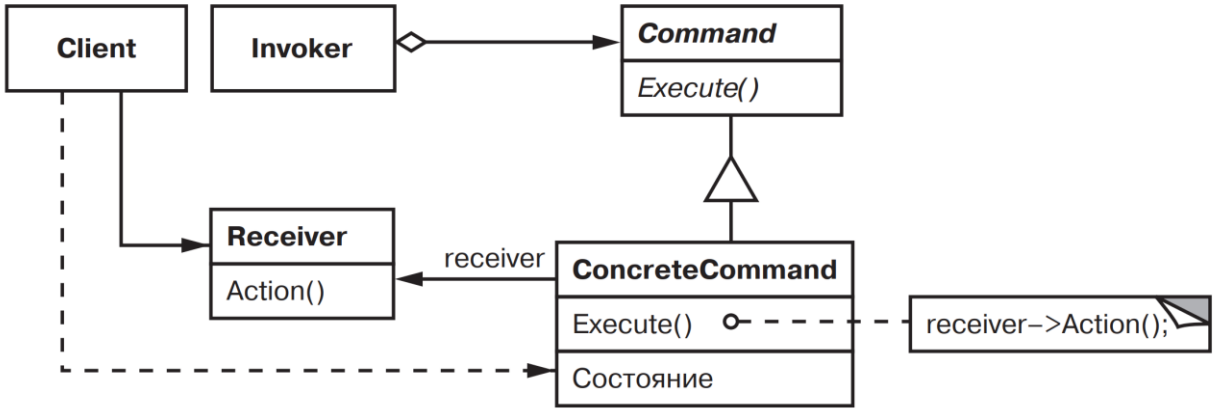


Рисунок 2.4 – Діаграма патерну команда

При побудові відношень ключі та команди зазвичай визначаються не за принципом наслідування, а з допомогою анотування. У Spring це анотація @Contoller. Проте може застосовуватись і пошук за наслідуванням від інтерфейсу чи абстрактного контролеру.

Конфігурування того які ключі буде обробляти команда виконується з допомогою параметрів анотації. Пари ключів запиту та обробників зберігаються у контексті додатку.

Ці команди прийнято називати як контролери тому, що для обробки запитів цей патерн застосовується у комбінації з архітектурним патерном MVC і ці класи виконують роль контролерів у ньому.[12]

У контексті обробки мережевих запитів архітектура "MVC" використовується щоб не було прямого зв'язку між контролером та об'єктами, що виконують запити, контролер може використовувати об'єкти команд для представлення та інкапсуляції запитів.

Таким чином, застосування патерну "Команда" у взаємодії з патерном "MVC" дозволяє добре розділити логіку обробки запитів від управління представленням та моделлю. Це поліпшує модульність, легкість читання та підтримки коду, оскільки логіка команд може бути легко масштабована та змінювана без впливу на контролер.

Крім того, використання патерну "Команда" допомагає зменшити залежність між контролером та об'єктами, що виконують запити, що робить програму більш гнучкою та підходящою для розширення функціональності. Взаємодія за допомогою об'єктів команд також сприяє створенню багатофункціональних та повторно використовуваних модулів.

Для забезпечення цього функціоналу на старті додатку проводяться підготовчі дії по пошуку контролерів в коді та побудові зв’язків ключів запитів і контролерів обробників. Для зниження часу запуску ці операції буде зроблено ліниво аналогічно до IOC контейнеру.

За виграш у швидкості старту нам доведеться заплатити більшим часом обробки першого запиту за кожним ключем. Як і в питанні вирішення залежностей IOC можуть бути ситуації коли цей компроміс буде не прийнятним, для компенсації цього також можна ввести властивість включення цієї обробки у процес запуску серверу за бажанням користувача.

Робота з мережевими запитами може мати різні підходи. Найрозповсюдженішими є архітектурний стиль REST, методо орієнтована обробка запитів в купі із технологією JSP сторінок. Для взаємодії у реальному часі по TCP протоколу застосовується технологія WEB сокетів. Для кожного з цих підходів можна створити окрему властивість конфігурації забезпечивши таким чином більше зручне та ефективне рішення питання компромісу швидкості обробки першого запиту та швидкості запуску серверу.

2.5 Зміни у підході до API фреймворку для зниження витрат на підтримку

В аналізі предметної області було звернуто увагу що максимальна свобода хоч і прискорює швидкість розробки проте вона може негативно впливати на підтримку оскільки дає можливість застосовувати не коректні архітектурні рішення. Для уникнення цього інтерфейси та анотації WEB частини фреймворку спроектовано згідно ідеї звуження свободи дій розробників для підвищення якості коду.

2.5.1 Зміни у API організації REST контролерів

REST – це архітектурний підхід до організації API. Його суть полягає у застосуванні ключових методів HTTP протоколу у якості ключів для CRUD операцій. Та побудови адрес запитів за концепцією ресурсів.[28]

У класичній REST архітектурі посилання будуються таким чином – спочатку іде протокол запиту потім адреса самого WEB сервісу, потім шлях до ресурсу і потім його ідентифікатор. На рисунку 2.5 у першій стрічці представлено цей шаблон. Протокол запиту завжди HTTP або HTTPS.

https://айпі.адреса.або.доменне.імя/ресурс/{опціонально код ресурсу}

1) GET https://example.com/users

2) GET https://example.com/users/12

3) POST https://example.com/users/12

4) PUT https://example.com/users

5) DELETE https://example.com/users/12

Рисунок 2.5 – Формула побудови URL запиту до REST ресурсу

Також на рисунку 2.5 зображено приклади всіх можливих запитів для ресурсу користувач. Операції читання відповідають запити з методом GET, оновлення POST, створення PUT і видалення DELET. Число 12 на кінці є ідентифікатором сутності ресурсу.

Згідно протоколу REST кожен ресурс може мати такі 5 операцій. Перший запит повертає набір всіх сутностей, і зазвичай застосовується з параметрами для пейджинації. Другий слугує для отримання конкретної сутності за ідентифікатором. Третій, четвертий і п’ятий для зміни, створення і видалення ресурсу відповідно.

Для вкладених ресурсів у формулу додається шлях в форматі назва ресурсу потім його код. На рисунку 2.6 зображено приклад запитів до ресурсу адреси користувача.

1) GET https://example.com/users/12/address

2) GET https://example.com/users/12/ address/5

3) POST https://example.com/users/12/ address/5

4) PUT https://example.com/users/12/ address

5) DELETE https://example.com/users/12/ address/5

Рисунок 2.6 – Приклади URL для взаємодії з вкладеним REST ресурсом адреси

Тепер розглянемо як робота з цими кінцевими точками організована в найпоширенішому фреймворку для цієї задачі «Spring WEB». Вона реалізується з допомогою анотації для класу @RestController та анотацій для методів @GetMapping, @PostMapping, @PutMapping, @DeleteMapping та @RequestMapping.

Анотація @RestController дозволяє задати загальний URL шлях для контролеру. Після чого анотаціями @GetMapping, @PostMapping, @PutMapping та @DeleteMapping можна його доповнювати. Власне чи має метод класу обробляти запит вирішується по відповідності посилання на ресурс в запиту та конкатенованому патерну адреса з анотації на рівні класу і з анотації на рівні методу».

Причому анотація на рівні класу не є обов’язковою. Це повністю розв’язує руки розробнику. Іноді він може собі дозволити ефективно об’єднати обробку

вкладених ресурсів в одному класі. Проте також це призводить до виникнення спокуси покласти все в один клас. Або взагалі відійти від конвенції іменування шляхів до ресурсів.

На рисунку 2.7 можна побачити приклад як могла б бути організована робота з допомогою фреймворку Spring, в одному контролері. Це є архітектурною помилкою що в майбутньому приведе до проблем, пов'язаних з розростанням класу та його залежностей.

```
@RestController
@RequestMapping(value = "/users/")
public class UserContoller{

    @GetMapping()
    public List<User> getAllUsers() {...}
    @GetMapping(value = "{userId}")
    public User getUser(Integer userId) {...}
    @PostMapping(value = "{userId}")
    public User getEditUser() {...}
    @PutMapping()
    public User getAddUser(Integer userId) {...}
    @DeleteMapping(value = "{userId}")
    public User getRemoveUser(Integer userId) {...}

    @GetMapping(value = "{userId}/address")
    public List<AddressData> getAllAddresses(Integer userId) {...}
    @GetMapping(value = "{userId}/address/{addressId}")
    public AddressData getAddress(Integer userId, Integer addressId) {...}
    @PostMapping(value = "{userId}/address/{addressId}")
    public AddressData getEditAddress(Integer userId, Integer addressId) {...}
    @PutMapping(value = "{userId}/address")
    public AddressData getAddAddress(Integer userId, Integer addressId) {...}
    @DeleteMapping(value = "{userId}/address/{addressId}")
    public AddressData getRemoveAddress(Integer userId, Integer addressId) {...}
```

Рисунок 2.7 – Приклад помилкового з архітектурної точки зору застосування анотації REST контролеру

Це можливо тому що методові анотації універсальні в цьому фреймворку. Вони також можуть бути застосовані наприклад для написання контролеру для роботи за протоколом SOAP. І там це буде доречним і коректним.

Для унеможливлення подібної ситуації в нашій реалізації було реалізовано окремо анотації для ідентифікації методів при роботі за архітектурою REST.

Шлях ресурсу котрий обробляється задається виключно у анотації @RestEndpoint на рівні класу. Після чого на методах застосовуються анотації @RestGetAll @RestGetById @RestUpdate @RestPut та @RestDelete. Завдяки цьому користувач фреймворку фізично не зможе зробити архітектурну помилку обробки кількох ресурсів в одному контролері.

На рисунку 2.8 зображено реалізацію прикладу з допомогою фреймворку що розробляється в рамках дисертації. Ми можемо бачити що відійшовши від погоні за універсальністю ми досягли структурного змушення розробника писати код відповідно до архітектурного стилю котрий він обрав.

Відхід від універсальності також дав можливість назвати анотації не за методами HTTP протоколу, а за їх реальною семантикою у архітектурі REST.

2.5.2 Можливість застосовувати REST подібний підхід в WEB формах

Змістовна робота з WEB формами є важливою складовою сучасних WEB додатків, спрямованих на взаємодію з користувачами. У багатьох випадках, для досягнення повноцінного REST-подібного підходу у взаємодії з даними, особливо в рамках CRUD (створення, читання, оновлення, видалення) операцій, використовують HTTP методи, такі як PUT і DELETE. Однак стандарт HTML-форм в даний час не підтримує пряму взаємодію з цими методами.[29]

Однією з головних причин відсутності підтримки методів PUT та DELETE в HTML-формах є те що в минулому, коли HTML з'явився, було менше акценту на повноцінний REST-підхід, який зазвичай вимагає використання різних HTTP-методів для відповідних операцій.

```
@RestEndpoint(resource = "user/{userId}")
public class UserController {
    @RestGetAll()
    public List<User> getAllUsers() {...}
    @RestGetById()
    public User getUser(Integer userId) {...}
    @RestUpdate()
    public User getEditUser() {...}
    @RestPut()
    public User getAddUser(Integer userId) {...}
    @RestDelete()
    public User getRemoveUser(Integer userId) {...}
}

@RestEndpoint(resource = "user/{userId}/address/{addressId}")
class AddressController {
    @RestGetAll()
    public List<AddressData> getAllAddresses(Integer userId) {...}
    @RestGetById()
    public AddressData getAddress(Integer userId, Integer addressId) {...}
    @RestUpdate()
    public AddressData getEditAddress(Integer userId, Integer addressId) {...}
    @RestPut()
    public AddressData getAddAddress(Integer userId, Integer addressId) {...}
    @RestDelete()
    public AddressData getRemoveAddress(Integer userId, Integer addressId) {...}
}
```

Рисунок 2.8 – Приклад неможливості не коректної реалізації архітектурного підходу REST у нашій розробці

В той час, коли WEB стандарти формувалися, основним завданням було надати простий механізм взаємодії з користувачами оскільки в більшості випадків

можна було обійтися лише з використанням методів GET і POST, додали підтримку лише їх.

Потреба у застосування методів PUT та DELETE в HTML-формах може виникнути, наприклад, при розробці додатків, які повністю використовують REST API для взаємодії з сервером та дотримання принципів RESTful архітектури. Таким чином, відсутність підтримки цих методів може створювати додаткові обмеження та ускладнення в реалізації повноцінного RESTful підходу і вести до не оптимальних архітектурних рішень.

Сучасні фреймворки не пропонують рішення цієї проблеми. Її можна вирішити додавши логіку що перехоплює POST запити і перевіряє чи містять вони параметр певний додатковий параметр, наприклад «_method».

Цей параметр буде приймати значення значення «put» «get», «delete» та «post». Після чого фронт контролер фреймворку на його основі буде приймати рішення який метод контролеру викликати для обробки запиту.

На рисунку 2.9 зображено приклад того як буде виглядати код html з застосуванням застосування цього параметру. Завдяки цьому підходу користувач має можливість ефективно розподіляти імена для ресурсів.Це рішення дозволить покращити якість архітектури додатків і як наслідок знизити вартість їх підтримки.

```
<form id="new-parameter" class="form" method="post"
      action="${pageContext.request.contextPath}/parameter">
    <input type="hidden" name="_method" value="put"/>
    <input name="queueCode" value="${queueDto.code}" type="hidden"/>
    <tr>
        <td>weight function</td>
```

Рисунок 2.9 – Застосування методу пут для форми створення нового параметру черги

2.6 Висновки розділу

В даному розділі було проаналізовано чому існуючі фреймворки не сфокусовані на мінімізації часу запуску. Після цього було проаналізовано сам процес запуску та виявлено, що прискорити запуск можна за рахунок оптимізації етапів пов’язаних з обробкою компонентів.

Запропоновано зміни в цих етапах котрі дозволять досягти прискорення. Основна ідея цих зміни мінімізація, або прибрання функціоналу що не є необхідним або перенесення його з етапу завантаження серверу на подальші етапи роботи серверу.

Ці зміни ведуть до компромісів у питаннях швидкості обробки запитів та раннього виявлення помилок у конфігурації. У якості часткового рішення цих проблем є можливість ввести властивості для гнучкого налаштування модульного включення кроків обробки компонентів у процес старту.

У якості поділу запропоновано такі блоки як власне завантаження загальних компонентів, REST компонентів, звичайних WEB компонентів та компонентів для роботи за протоколом WEB Socket.

Ще одним компромісом який вимагають ці зміни відсутність підтримки кругових залежностей між компонентами. З точки зору теми дисертації це не є недоліком. Навіть у випадку можливості їх реалізації від цих залежностей в рамках наших цілей потрібно було б відмовитись або хоча б увести в конфігурацію можливість їх включити за необхідності. Оскільки застосування кругових залежностей є поганою практикою та свідчить про некоректність проектування архітектури.

Було виявлено недолік у підтримці існуючими фреймворками архітектурного стилю REST. Він також зайву свободу в рамках контролерів.

Для рішення цього було запропоновано зміну підходу до API фреймворку щоб змусити розробників проектувати коректні REST контролери.

Також виявлено другий архітектурний недоліків у підтримці REST та мові HTML, а саме відсутність модифікацій для підтримки методів PUT та DELETE в обробці WEB форм. Пояснено яким чином це веде до не можливості застосовування REST підходу на боці серверу при обробці цих форм. Було запропоновано метод рішення цієї проблеми.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ

3.1 Реалізація роботи з WEB

Фреймворк підтримує роботу за чистим HTTP протоколом, за архітекрурою REST та за протоколом WEB socket. На рисунку 3.1 схематично зображено як відбувається обробка WEB запиту.

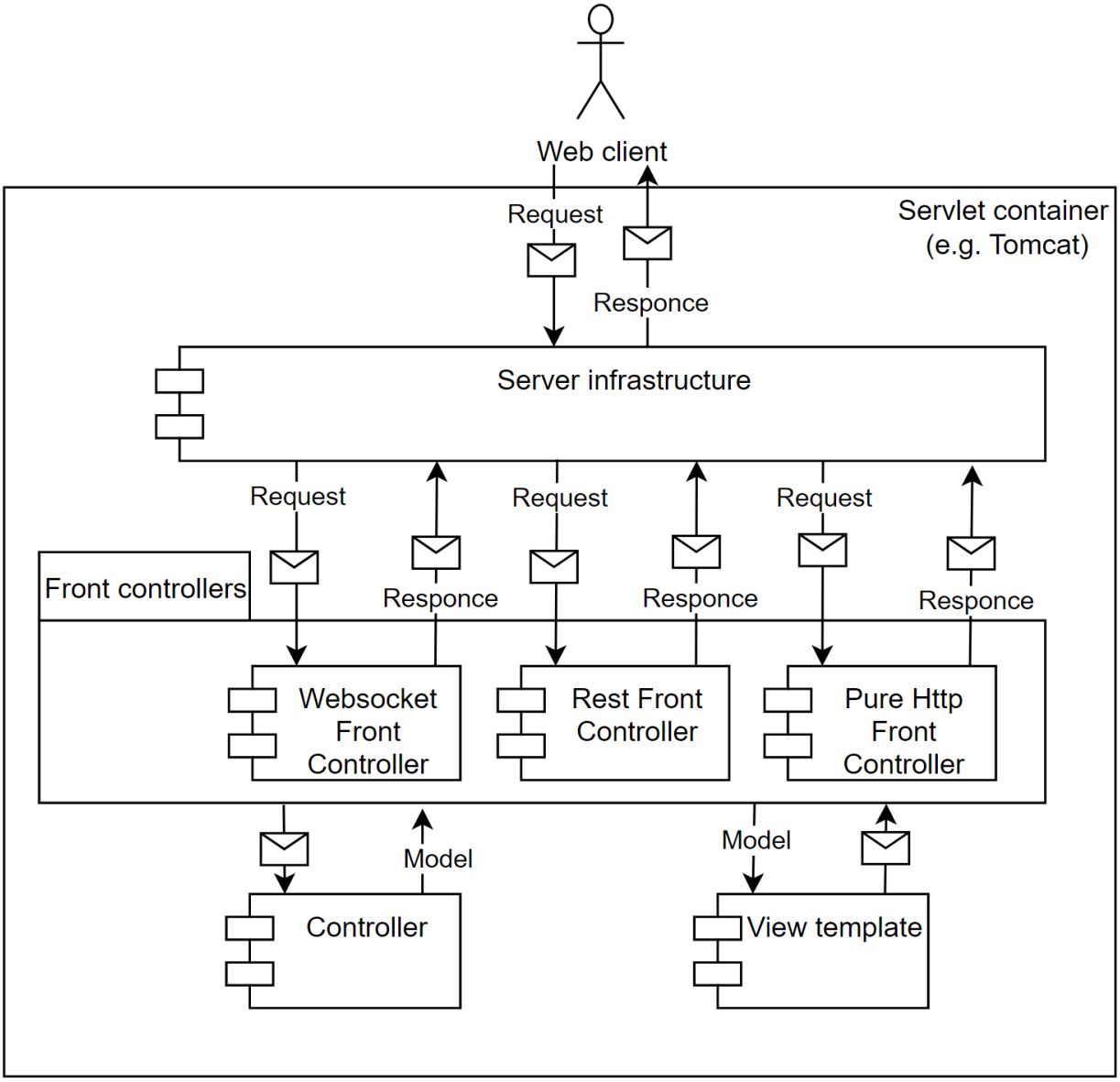


Рисунок 3.1 – Узагальнена діаграма патерну обробки запитів Фронт контролер

Обробка запиту починається з його отримання сервером (або контейнером сервлетів), наприклад Tomcat. Після цього на базі конфігурації в рамках стандартних вимог специфікації Java EE запит передається на фронт контролер що відповідає за обробку цього типу мережевої взаємодії.

Фронт контролери є частиною розробленого фреймворку. Вони виступають клієнтами в рамках патерну команда, їх функція полягає у виконанні стандартної обробки запиту:

- визначення за адресою запиту контролера що відповідальний за обробку цього запиту.
- вилучення з мережевого повідомлення інформації про параметри запиту та приведення їх до формату декларованого у API відповідного типу WEB обробки.
- передача обробленого запиту на обробку відповідному контролеру та отримання від нього відповіді у вигляді результуючої сутності.

У випадку якщо обробка запиту не вимагає перетворення у візуальне представлення фронт контролер перетворює результат отриманий від контролера у формат що відповідає специфікації Java EE та передає його серверу. Сервер в свою чергу відправляє результат клієнту.

У випадку якщо потрібне перетворення у візуальне представлення WEB сторінки фронт контролер знов виступає клієнтом відповідно до патерну команди уже по відношенню до шаблонів відображення. Обирає шаблон відображення що відповідає за обробку ключа повернутого на обробку від контролера та передає йому на обробку модель. Шаблон відображення у відповідь повертає html код створений на базі даних з моделі. Після цього результуюча сторінка передається серверу для повернення клієнту.

3.1.1 Реалізація роботи за чистим HTTP протоколом та з підтримкою архітектурного стилю REST

Перед початком обробки запиту для забезпечення безпеки від XSS атак відпрацьовує фільтер «XSSFilter». Він замінює всі скрипт подібні символи на їх коди при виконанні запиту.

Фронт контролером для обробки запитів за HTTP типом взаємодії відповідає клас «PureHttpFrontController», а за обробку запитів в архітектурному стилі REST відповідає «RestFrontController». Спільний функціонал цих контролерів винесено в батьківський клас «FrontController» що в свою чергу опосередковано реалізує інтерфейс «Servlet». На рисунку 3.2 зображено відношення між ними.

GenericServlet дозволяє згідно специфікації Java Servlet Specification назначати його відповідальним за обробку вхідних HTTP запитів. Розділення який запит якому обробнику належить відбувається з допомогою першого кроку адреси ресурсу. За замовчуванням це «http» та «rest» відповідно.

Отримавши запит на обробку фронт контролер базуючись на адресі та інформації з запиту отримує від «ApplicationContext» з допомогою методів «getRestCommand» та «getHttpCommand» обробник що відповідає за обробку даного запиту. Їх взаємодію та названі методи інтерфейсу ApplicationContext можна побачити на рисунку 3.2.

Після чого фронт контролер ґрунтуючись на інформації про контролер з допомогою класу «RestUrlUtilService» формує на основі запиту масив параметрів. Після цього передає обробку запиту конкретному контролеру.

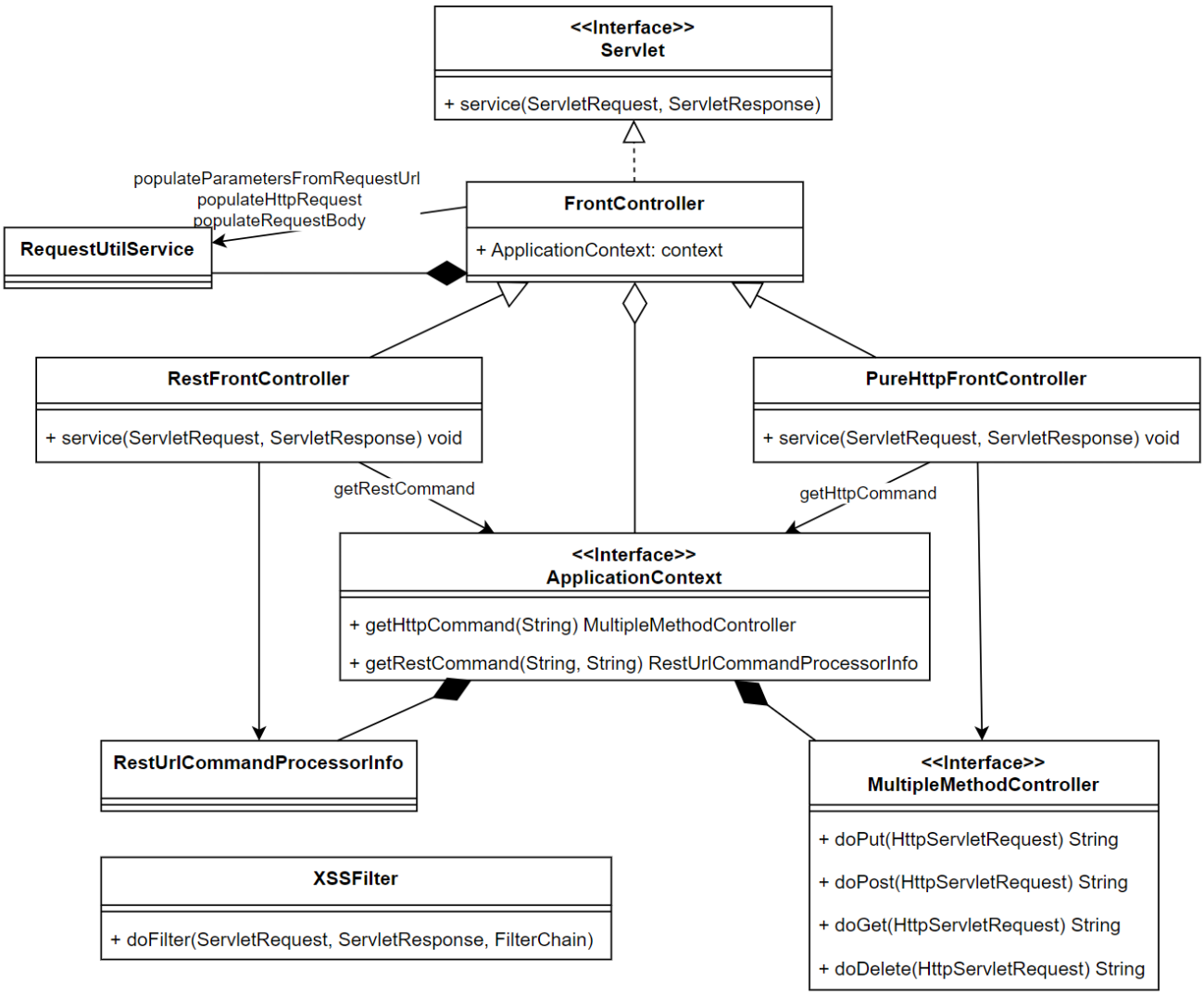


Рисунок 3.2 – Діаграма ключових класів обробки HTTP та REST запитів

3.1.2 Реалізація роботи за протоколом WEB Socket

Робота за протоколом WEB Socket сильно відрізняється від описаних вище видів мережевої взаємодії. Це логічний наслідок того що HTTP протокол працює по принципу послідовних запитів та відповідей в той час як сокети дають можливість асинхронного обміну повідомленнями в обох напрямках.

Незважаючи на це у фреймворку вдалось досягти подібної конфігурації та організації роботи. Для забезпечення цього додано поняття типу повідомлення. Базуючись на ньому визначається контролер відповідальний за його обробку.

Мережеві повідомлення мають відповідати такому формату – код повідомлення, вертикальна риска та JSON об’єкт повідомлення. Наприклад «SUBSCRIBE_TO_ID|{"idsToSubscribe":[1]}».

За представлення у коді повідомлень відповідають класи «SocketReceivedMessage» та «SocketSendMessage». Обидва класи мають текстове поле що зберігає тип повідомлення, а також стрічку json об’єкту та об’єкт результату запиту до серверу відповідно.

За конвертацію тексту повідомлення у об’єкт відповідає «MassageDecoder». За конвертацію об’єкта повідомлення від серверу у текст відповідає «MassageEncoder». На рисунку 3.3 зображено струкуру класів повідомлень, декодеру та енкодеру. Декодер та енкодер реалізують інтерфейси відповідно специфікації Jakarta EE.

Фронт контролером виступає клас «WebSocketFrontController». Він аналогічно до фронт контролерів описаних раніше з допомогою контексту отримує обробника повідомлення за його типом. Окрім цього він використовує контекст для подальшого отримання класу що очікує отримати контролер відповідальний за запит, а також отримання за типом відповіді від контролеру коду цього повідомлення. Див рисунок 3.3.

3.1.3 Аутентифікація та сповіщення між WEB Socket сесіями

Для аутентифікації користувача було створено «LoginTcpController». Перше повідомлення має бути повідомленням типу «LOGIN». Код користувача та пароль

отримані в ньому контроллер передає «LoginWebSocketService». І у випадку якщо сервіс повертає що доступ такому користувачу дозволено аді користувача додається в сесію. Реалізацію «LoginWebSocketService» має виконати користувач залежно у тому виді який задовольнить його потреби. На рисунку 3.3 продемонстрована ця взаємодія.

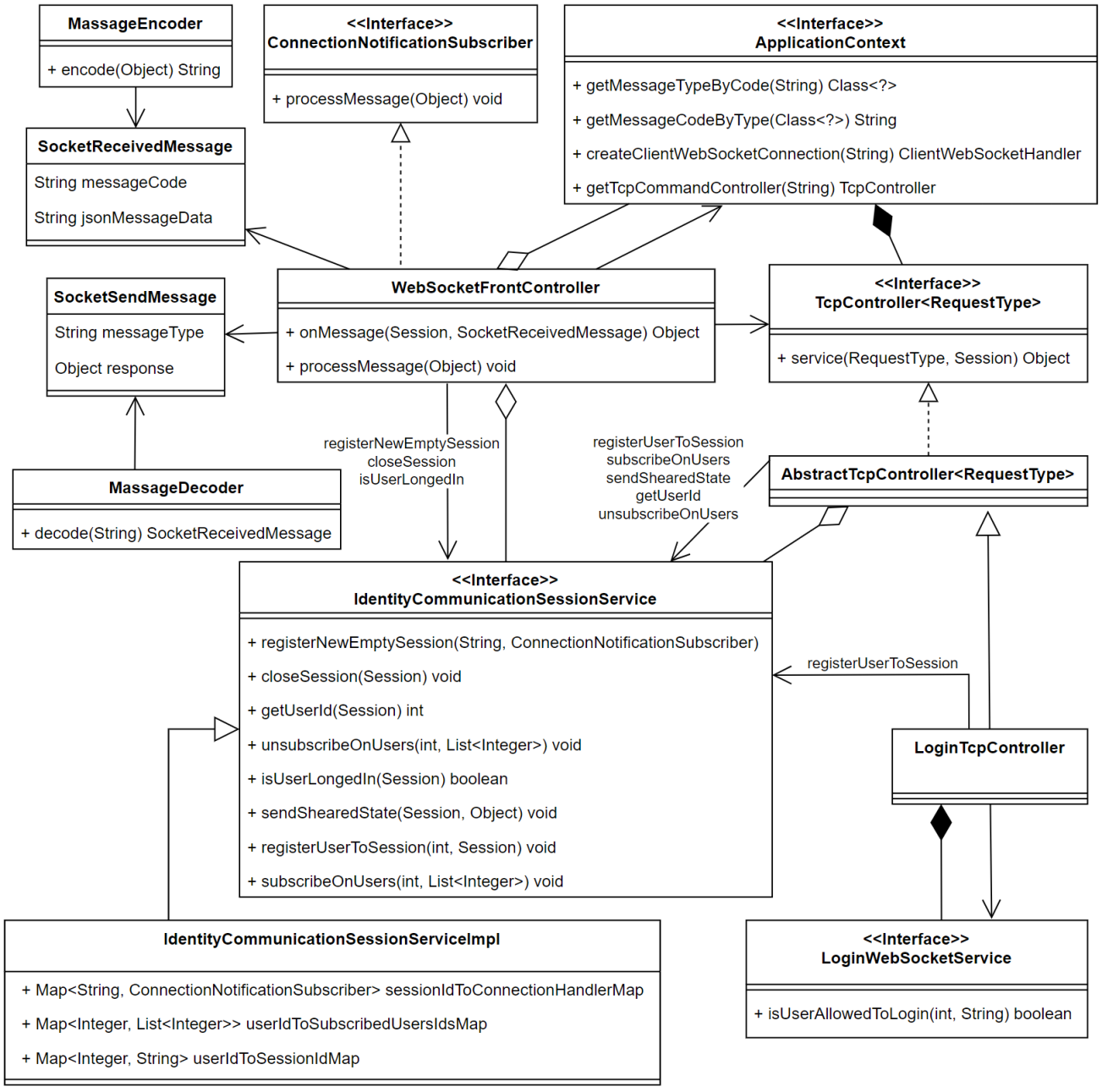


Рисунок 3.3 - Діаграма ключових класів обробки WEB Socket запитів

У випадку якщо буде спроба виконати будь який інший запит до того як буде виконано успішну аутентифікацію буде повернута помилка.

Інтерфейс «IdentityCommunicationSessionService» надає можливості підписання та відмови від повідомлень інших користувачів, а також відповідає за відправлення сповіщень іншим користувачам.

Для забезпечення цього функціоналу в ньому є 3 словники див рисунок 3.3. Словник «userIdToSessionIdMap» зберігає інформацію про ідентифікатор сесії користувача за ідентифікатором самого користувача. Завдяки ній при відкритті нового з’єднання сервіс завжди знатиме ідентифікатор саме поточної сесії навіть якщо стара ще не закрилась.

Словник «userIdToSubscribedUsersIdsMap» зберігає ідентифікатори всіх користувачів що зацікавлені в сповіщеннях від конкретного користувача.

І останній словник «sessionIdToConnectionHandlerMap» пов’язує ідентифікатор сесії користувача та екземпляр класу фронт контролера що пов’язаний з заданою сесією і прихований під інтерфейсом «ConnectionNotificationSubscriber», див рисунок 3.3.

Завдяки цьому підходу обробка сповіщень передається фронт контролеру та далі обробляється з точки зору користувача так само як і мережеві повідомлення. Це дає можливість пере використовувати одні і ті самі контролери та повідомлення і дуже сильно спростити конфігурацію. По суті користувачу не важливо при розробці чи приходить повідомлення з мережі чи від іншої сесії.

3.2 Контекст додатку

Контекст додатку - це оточуюче середовище, де працює клієнтський додаток. По суті це набір інформації та ресурсів, необхідних для функціонування додатку. У розробленому фреймворку точкою в ходу в нього є інтерфейс «ApplicationContext». Він є ядром контейнера інверсії залежностей який забезпечує керування компонентами та їх залежностями в додатку.

Контекст відповідає за:

- створення компонентів на основі конфігурації з допомогою анотацій.
- впровадження залежностей в компоненти. Тобто автоматичне надання компонентам необхідних об'єктів та ресурсів, замість того, щоб вони самостійно їх створювали.
- забезпечення доступу до компонентів: Контекст додатку дозволяє отримати компоненти за допомогою їхніх ідентифікаторів або класів, що дозволяє зручно управляти об'єктами в додатку.
- забезпечення доступу до налаштувань додатку. З допомогою методу «getPropertyValue» можна отримати за ключем значення властивості із конфігураційного файлу «application.properties».

Загалом, контекст додатку є централізованим місцем де визначається, налаштовуються та отримується доступ до компонентів додатку. Централізація місця досягається з допомогою застосування патерну Singletone. ApplicationContext має статичний метод «getContext» котрий повертає всім один і той самий екземпляр контексту.

Оскільки робота без контексте є неможливою створення сінглтону зроблено жадібно у статичному блоку ініціалізації класу для уникнення в подальшому зайвої перевірки на те чи створено контекст при кожному запиті до методу «getContext».

Контекст має підтримку чотирьох особливих видів компонентів що мають особливості пов’язані мережевою взаємодією. Для них контекст надає окремі кеші та можливості.

3.1.1 Архітектура управління компонентами.

Згаданий вище контекст не створює компонентів а лише управляє їх життєвим циклом. Згідно з архітектурними принципами солід ця логіка винесена в фабрику «ObjectFactory». На рисунку 3.4 можна побачити їх зв’язок.

Інформація про конфігурацію отримується з допомогою відкритої бібліотеки «Java runtime metadata analysis» [30]. Вона дозволяє знизити кількість бойлер коду пов’язаного з отриманням метаданих по додатку.

Для гнучкості додатку було створено адаптер у вигляді інтерфейсу «Config» що надає абстракцію для доступу до конфігурації. І створено його реалізацію «JavaConfig» яка покладаючись на клас «org.reflections.Reflections» вище згаданої біліотеки реалізує потрібні конфігураційні методи. В подальшому за потреби це дасть можливість не міняючи код фреймворку додати інший вид конфігурації, наприклад xml.

За впровадження значень атрибутів компоненту відповідає інтерфіейс «ObjectConfigurator». Цей функціональний інтерфейс приймає об’єкт, його тип та контекст. Контекст потрібеноскільки він відповідає за управління змінними оточення та іншими копонентами що потрібні для наповнення налаштовуванго компоненту. На рисунку 3.4 зображено два наслідники цього класу що відповідають за впровадження посилань на обєкти інших компонентів та заповнення текстових атрибутів класу. Це

«InjectStringPropertyAnnotationObjectConfigurator» та «InjectStringPropertyAnnotationObjectConfigurator» відповідно.

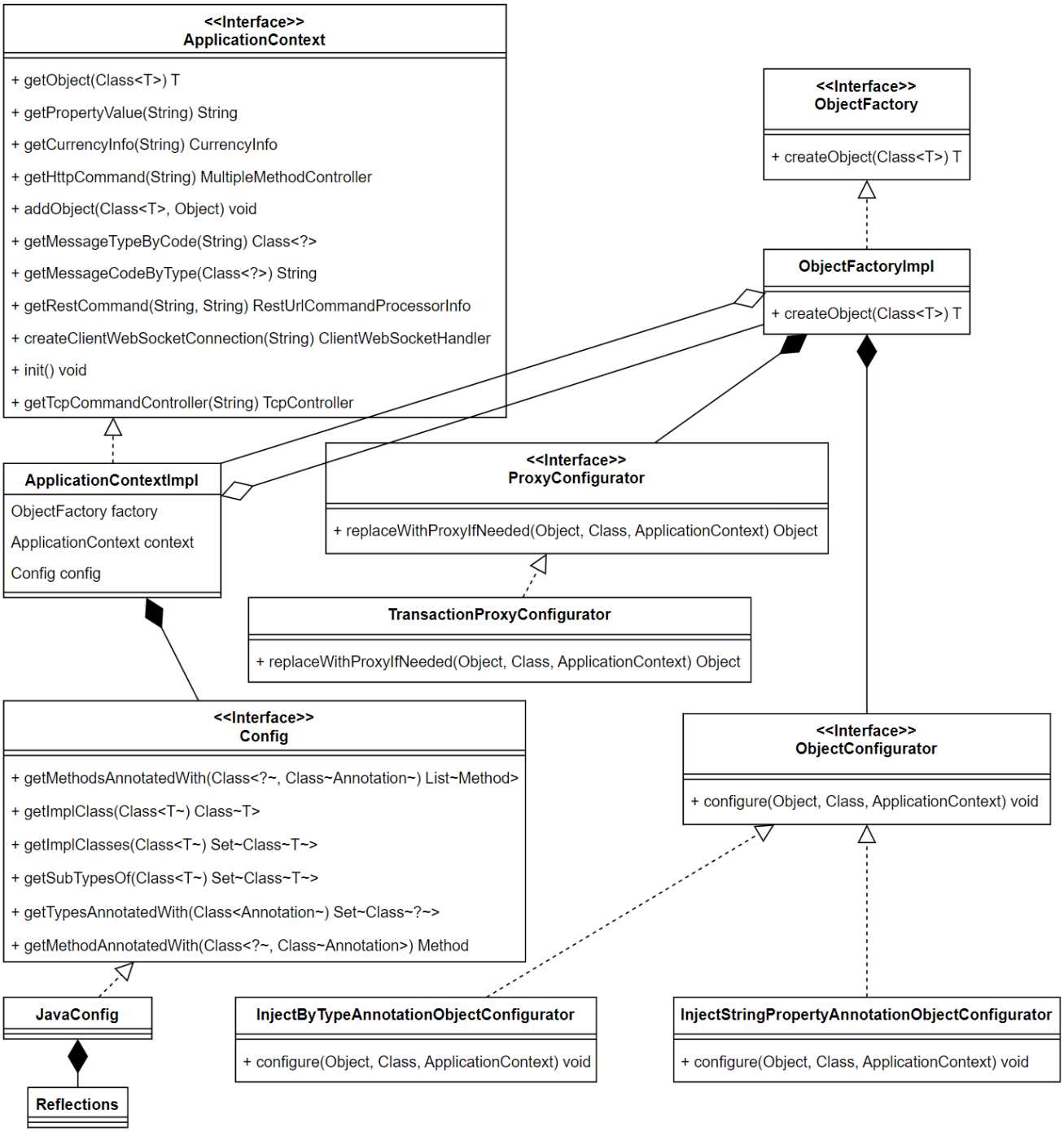


Рисунок 3.4 – Діаграма класів підтримки життєвого циклу компонентів

За створення проксі об’єктів для додання додаткової логіки відповідає інтерфейс «ProxyConfigurator». Наразі створена єдина його реалізація що відповідає за додання до методів сервісів що повинні працювати в рамках транзакцій логіки відкриття та закриття транзакцій.

3.2.1 Контроль життєвого циклу компонентів

З допомогою методу getObject інтерфейсу ApplicationContext відбувається отримання та якщо потрібно створення компоненту за його типом. Тип бажаного об’єкту передається в метод у якості параметру.

Контекст містить в собі словник пар типу класу та вже створеного компоненту для нього. Цей словник слугує для реалізації патерну Singletone. При створенні компоненту що відмічений анотацією @Singletone відбувається збереження його в цей словник.

Спочатку відбувається перевірка чи компонент уже присутній в словнику, якщо так він повертається. Якщо ні з допомогою конфігурації отримується конкретний тип реалізації запитуваного класу або інтерфейсу та передається у фабрику для створення компоненту. На рисунку 3.5 це зображено як перші три кроки.

У кроці створення об’єкту фабрикою криється основна робота. Для забезпечення принципів SOLID вона рознесена по різним допоміжним класам. Спочатку фабрика як відповідальна за створення об’єкту створює порожній екземпляр класу.[26]

Після цього у випадку якщо клас помічений анотацією «NeedConfig», створений екземпляр передається по ланцюжку обов’язків утвореному з класів що реалізують інтерфейс «ObjectConfigurator». Див рисунок 3.4. Для позначення що

таке впровадження потрібне атрибут має бути помічений анотаціями @InjectStringProperty або @InjectByType, для впровадження текстового поля або посилального типу відповідно. Конфігуратори знаходять всі відмічені поля та заповнюють їх.

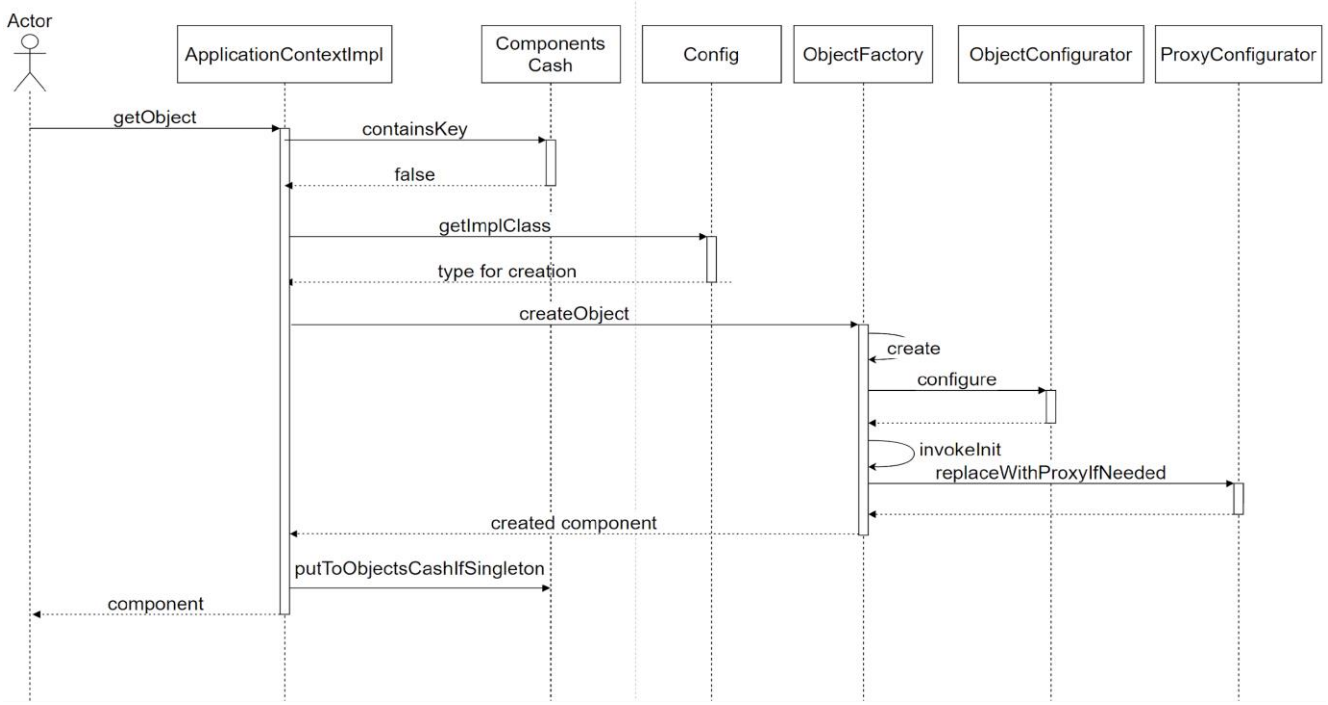


Рисунок 3.5 – Діаграма послідовності створення компоненту при його відсутності в кеші компонентів

Далі фабрика викликає всі ініціалізаційні методи помічені анотацією @PostConstruct, якщо такі є. Ініціалізаційні методи застосовуються в якості конструкторів після впровадження залежностей, оскільки часто для налаштування потрібно мати залежності котрих на етапі виклику конструктору немає бо ще не було виконано їх заповнення конфігураторами.

Останнім кроком у створенні компоненту є додання у випадку необхідності обгорток. Для цього об’єкт знов передається ланцюжком обов’язків створеним уже із класів що реалізують інтерфейс «ProxyConfigurator».

Після цього об’єкт повертається контексту котрий перевіряє чи відмічений компонент є поміченим анотацією «@Singleton», якщо так то зберігає його в кеш. Далі компонент відається тому хто в ньому був зацікавлений.

3.1.2 Підтримка контекстом роботи за чистим HTTP протоколом

Контролери що відповідають за обробку чистих HTTP запитів відмічаються анотацією «@HttpEndpoint» та реалізують інтерфейс «MultipleMethodController». Контекст зберігає окремий кеш для цих контролерів.

Кеш зберігається в словнику де в якості ключа виступає шлях пошукового запиту котрий цей контролер має обробляти, а в якості значення клас обробнику. Сам об’єкт обробника у випадку якщо він є сінглтоном керується за загальними правилами керування екземплярів інших компонентів.

Для отримання контролеру по шляху пошукового запиту є метод «getHttpCommand». При його виклику спершу відбувається перевірка чи вже за кешовано для заданого ключа контролер. У випадку якщо ні шукається клас компоненту відмічений анотацією «@HttpEndpoint» з співпадаючим з шляхом запиту параметром «value».

Оскільки під час пошуку обробника відповідального за обробку запиту ми вимушені послідовно перевіряти класи що реалізують інтерфейс для економії часу у наступних запитах ми зберігаємо в кеш всі контролери котрі довелось обробити у процесі пошуку відповідального за конкретний запит.

3.1.3 Підтримка контекстом роботи за REST протоколом

Компоненти що є контролерами запитів за REST протоколом є дещо складнішими. Для їх конфігурації застосовуються анотації «@RestEndpoint», «@RestGetById», «@RestGetAll», «@RestPut», «@RestUpdate» та «@RestDelete». Перша відповідає за відмітку класу як REST контролеру, а інші за відмічення відповідних REST методів.

Аналогічно до чистого HTTP контекст має метод «getRestCommand» який відповідає за надання обробника запиту за його адресою.

REST запити на відміну від HTTP передають змінні в стрічці запиту. Для подальшого їх застосування в конфігураційній стрічці ресурсів вони іменуються та ідентифікуються з допомогою фігурних дужок. На рисунку 3.6 продемонстровано приклад конфігураційної стрічки.

Через параметри у пошуковій стрічці створення компонентів обробки REST запитів має свої особливості. На рисунку 3.6 запропоновано графічне представлення алгоритму їх створення.

Якщо компонент не був створений раніше з його конфігурації отримується стрічка ресурсу з параметрами. Після чого з неї видаляються всі параметри і стрічка що залишилась використовується в якості ключа для кешу контролерів. Див приклад на рисунку 3.6.

З видалених параметрів утворюється масив кодових назв змінних та їх порядкового номеру в ланцюжку ресурсу. Ця інформація буде потрібна для передання значень що будуть знаходитись в запиті на відповідних позиціях в якості змінних контролерам. Ці дання зберігаються в масиві типу RestUrlVariableInfo. Його структура зображена на рисунку 3.6.

Після цього створюється об’єкт типу RestProcessorCashInfo котрий виступає в якості значення що зберігається в кеші контролерів. В нього зберігається інформація про змінні ресурсу, клас контролеру що відповідає за обробку цього ресурсу та посилання на методи що відповідають за кожен тип REST запиту. Структура класу також зображена на рисунку 3.6.

Після цього стандартним чином створюється компонент за класом. Далі створений компонент обробника разом із методом що відповідає за обробку заданого запиту та змінними передається фронт контролеру.

3.2.4 Підтримка контекстом роботи WEB Socket

Подібно до HTTP та REST контекст надає метод для отримання контролеру обробки запиту і для WEB Socket. Метод «getTcpCommandController» повертає команду за типом. Окрім цього є методи «getMessageTypeByCode» та «getMessageCodeByType», що повертають тип повідомлення за його кодом та код повідомлення за типом відповідно.

В конфігураційній анотації «@TcpEndpoint» відсутній параметр типу повідомлення яке він обробляє, це визначення досягається транзитивно через тип повідомлення котрим параметризується інтерфейс клас «TcpController». Завдяки цьому досягається відсутність дублювання цього параметру та як наслідок можливих помилок.

Також завдяки цьому з мінімальними додатковими витратами обчислювальних ресурсів паралельно з обробкою та кешуванням контролерів відбувається і керування повідомлень. Що в свою чергу ефективніше знижує час на обробку подальших запитів.

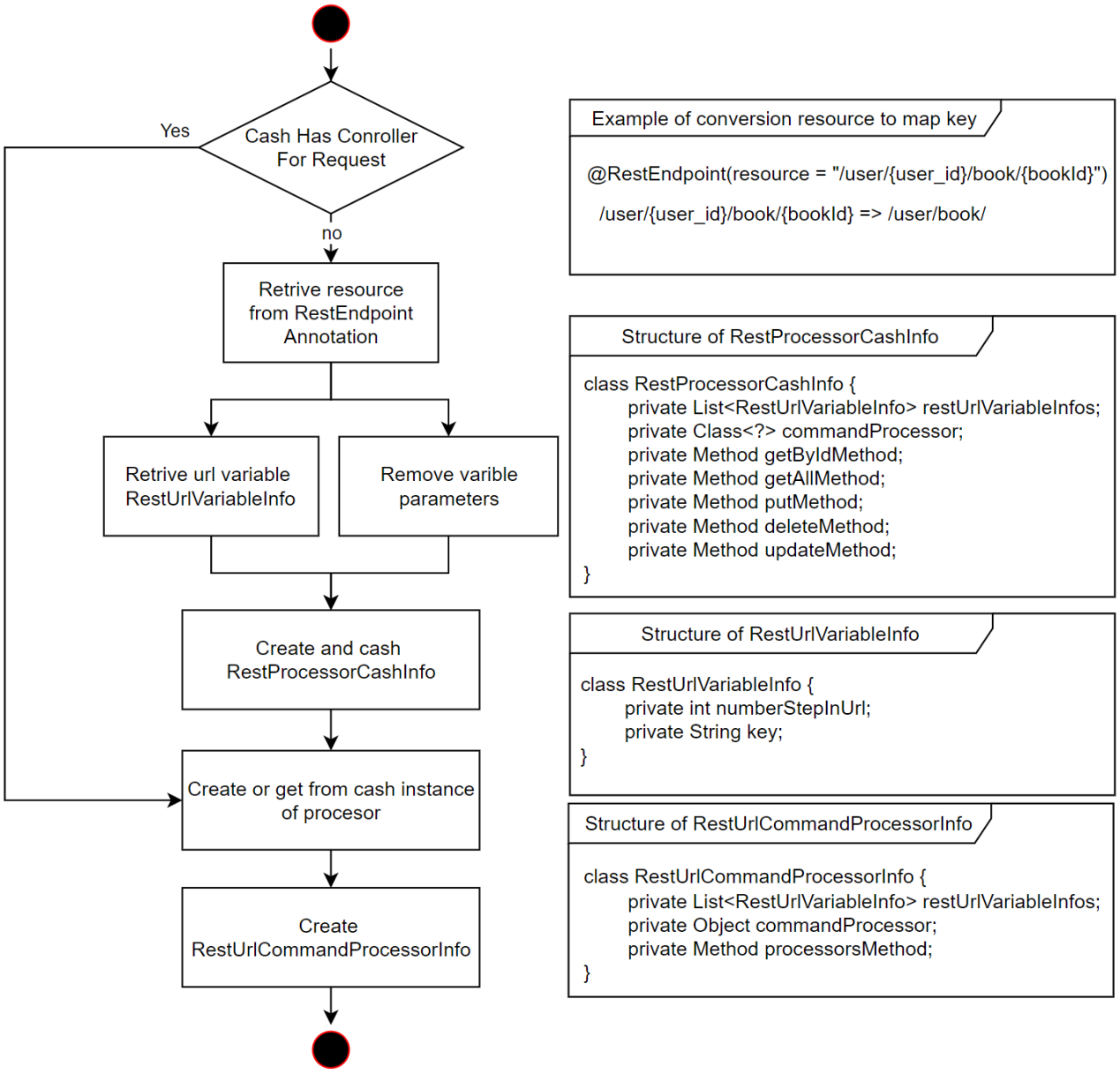


Рисунок 3.6 – Алгоритм створення компоненту типу REST за запитом

3.2.5 Кешування при старті серверу

Кешування компонентів згадане раніше за необхідності може відбуватись і під час створення контексту. Для мінімізації часу запуску та гнучкого налаштування для кожного кешу створено окрему конфігураційну властивість.

На рисунку 3.7 зображено приклад конфігурації. У порядку зображеному на рисунку властивості дозволяють вмикати кешування сінглтон компонентів, кешування компонентів мережевих повідомлень, компонентів обробки http повідомлень, компонентів протоколу REST та компонентів TCP. Зображена на рисунку конфігурація вмикає обчислення всіх кешів при старті.

```
#sturtup settings
infrastructure.include.in.start.singleton=true
infrastructure.include.in.start.network.dto=true
infrastructure.include.in.start.http.controllers=true
infrastructure.include.in.start.rest.controllers=true
infrastructure.include.in.start.tcp.controllers=true
```

Рисунок 3.7 – Приклад конфігурації обробки кешів при створенні контексту

3.3 Управління транзакціями бази даних

Для з’єднання з базою даних використовується реалізація пулу підключень від apache commons. Пул підключень є ефективним рішенням для контролю взаємодії з базою даних та максимізацією ефективності роботи з нею. Отримання підключення для виконання запиту відбувається не напряму а через інтерфейс «ConnectionManager».

Це потрібно оскільки для забезпечення транзакцій у фрейморку повинно бути єдине контрольоване ним місце отримання підключень до бази даних. Також це потрібно щоб забезпечити незалежність програмної реалізації від бібліотеки взаємодії з базою даних.

Для забезпечення транзакції в рамках різних викликів методів бази даних потрібно забезпечити одне підключення на один поток серверу. Для цього клас

реалізації інтерфейсу використовує бібліотечний параметризований тип «ThreadLocal». Що дозволяє зберігати в посилання єдиний об'єкт підключення для кожного потоку. На рисунку 3.8 представлено реалізацію інтерфейсу що зветься «ConnectionManagerImpl» та приватне поле «connectionThreadLocal» в якому зберігається підключення відповідного потоку.

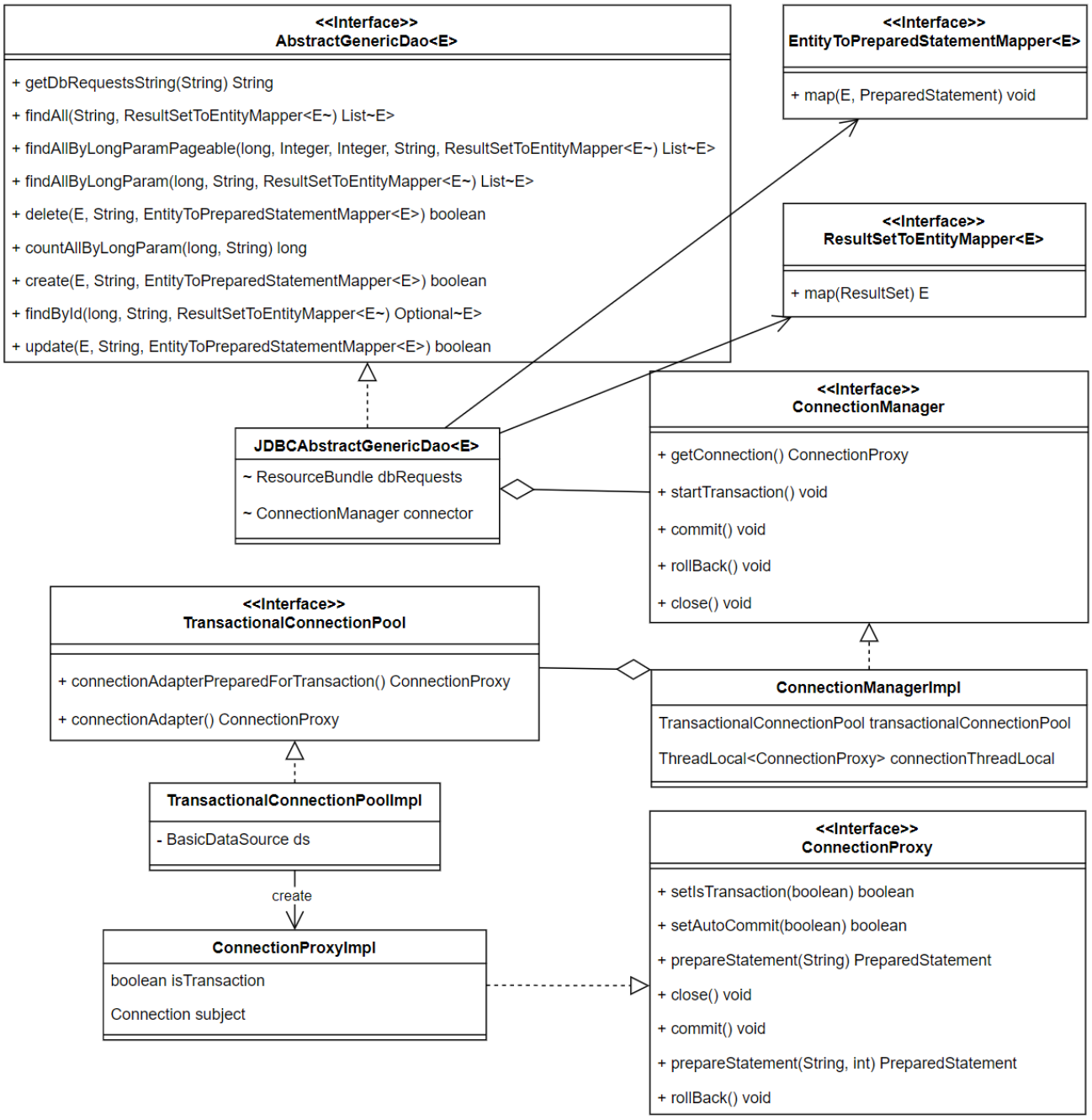


Рисунок 3.8 – Діаграма класів роботи з базою даних

Для отримання нового підключення «ConnectionManagerImpl» використовує інтерфейс «TransactionalConnectionPool» що обгортає бібліотечну реалізацію пулу підключень згадану раніше. Саме підключення, відповідно до ідеї гнучкості коду, також обгортається у інтерфейс «ConnectionProxy». Взаємодія цих класів зображена на рисунку 3.8.

Користувачу для роботи з БД достатньо з допомогою впровадження залежностей отримати в свій клас інтерфейс «ConnectionManager». Або унаслідувати клас «AbstractGenericDao» котрий пропонує реалізацію деяких базових методів та вже має впроваджений менеджер підключень.

Для забезпечення роботи транзакцій «TransactionProxyConfigurator» додає в проксі класу що має транзакційні методи логіку що перед викликом транзакційної логіки з допомогою того самого «ConnectionManager» створює транзакційне підключення до бази даних для цього потоку та відкриває транзакцію.

Після виконання методу конфігуратор викликає збереження та закриття транзакції. А також у випаду не передпачуваного завершення методу виключенням відкатує транзакцію назад. Та для забезпечення невидимості втручання повторно передає нагору те саме виключення що виникло у клієнтському коді.

3.4 Підхід до виконання запитів та базові сценарії їх виконання реалізовані у фреймворку

Вже згаданий раніше клас «JDBCAbstractGenericDao» пропонує базові можливості для реалізації конфігураційного підходу до виконання запитів та власне реалізацію базових CRUD операцій для абстрактного об’єкту. Достатньо параметризувати його відповідним типом, передати запит та реалізацію

перетворювачів параметрів запиту у «PreparedStatement» та результату «ResultSet» у об’єктну модель даних. На рисунку 3.8 зображено відповідні інтерфейси конверторів.

Також цей клас реалізує метод «getDbRequestsString» отримання запиту за його кодом у конфігураційному файлі. Це дозволяє без змін у коді за необхідності міняти запити до БД.

На рисунку 3.9 представлено приклад оновлення імені користувача. З нього видно мінімальну кількість коду необхідну для цієї операції. Достатньо запиту до бази даних та створити наповнювач запиту до бази даних на базі сутності користувача. Цього достатньо щоб абстрактний клас DAO виконав запит до бази даних.

```
@Override
public boolean updateUser(UserModel userModel) {
    return update(userModel, getDbRequestsString(UPDATE_USER_ID), getModelEntityToPreparedStatementMapper());
}

private static EntityToPreparedStatementMapper<UserModel> getModelEntityToPreparedStatementMapper() {
    return (entity, preparedStatement) -> {
        preparedStatement.setInt( parameterIndex: 2, entity.getId());
        preparedStatement.setString( parameterIndex: 1, entity.getName());
    };
}
```

Рисунок 3.9 – Запит на оновлення імені користувача в базі даних

3.5 Висновки розділу

В розділі було описано та обґрунтовано архітектурні рішення прийняті для забезпечення сформованих у попередніх розділах та у завданні задач. Описано максимальну однотипність обробки WEB запитів з допомогою патернів команда та фронт контролер.

Описано реалізацію управління компонентами для мінімізації обчислювальних витрат особливо при старті. Було приділено увагу мінімізації

уповільнення обробки запитів за рахунок оптимізації процесу кешування компонентів що відповідають за обробку WEB запитів усіх типів.

Також розкрито підтримку роботи з базою даних. Підтримку транзакцій та підходу до управління обробкою запитів з допомогою CRUD методів абстрактного класу DAO та конфігурації. Показано як такий підхід дає додаткову гнучкість та одночасно мінімізує складності пов’язані з відмовою від об’єктно-реляційних перетворень з допомогою рефлексії.

4 ПІДТВЕРДЖЕННЯ КОРЕЛЯТИВНОСТІ МЕТОДУ ТА МЕТИ

4.1 Оцінка зниження часу запуску

Для визначення досягнення зниження часу запуску порівняно час запуску розробленого фреймворку з уже існуючими фреймворками. Для ефективного виконання експериментів було створено генератор проектів щоб автоматизувати створення тестових додатків для різних фреймворків та кількостей залежностей у додатку.

У таблиці 4.1 запропоновано конфігурацію системи в якій проводилось тестування.

Таблиця 4.1 – конфігурація тестового середовища.

Параметр	Значення
Операційна система	Windows 10
Розрядність операційної системи	64
Кількість процесорів	1
Кількість ядер в процесорах	4
Кількість логічних процесорів	8
Частота процесорів	3.00 GHz
Об’єм оперативної пам’яті	32 GB
Частота оперативної пам’яті	3200 Hz
Тип жорсткого диску	SSD

Тришарова архітектура є базовим архітектурним підходом організації додатків. Вона пропонує мінімальну кількість шарів для ефективного поділу функціоналу додатку. Її типова структура складається з п’яти компонентів – контролер обробки запитів, інтерфейс сервісу та сам сервіс що виконує бізнес логіку і ДАО інтерфейс та його реалізація для взаємодії з БД.

Перший тестовий сценарій це дослідження впливу на час запуску при жадній ініціалізації різних фреймворків, залежно від кількості ланцюжків контролер, сервіс, ДАО. На рисунку 4.1 можна побачити наскільки розроблене в рамках дисертації рішення є ефективнішим з точки зору мінімізації часу запуску в порівнянні з існуючими фреймворками. При кількості типових ланцюжків від 100 до 1000 час запуску нижчий приблизно у 7 разів в порівнянні з фреймворком Spring та у 13 разів в порівнянні з WEB сервером Glass Fish.

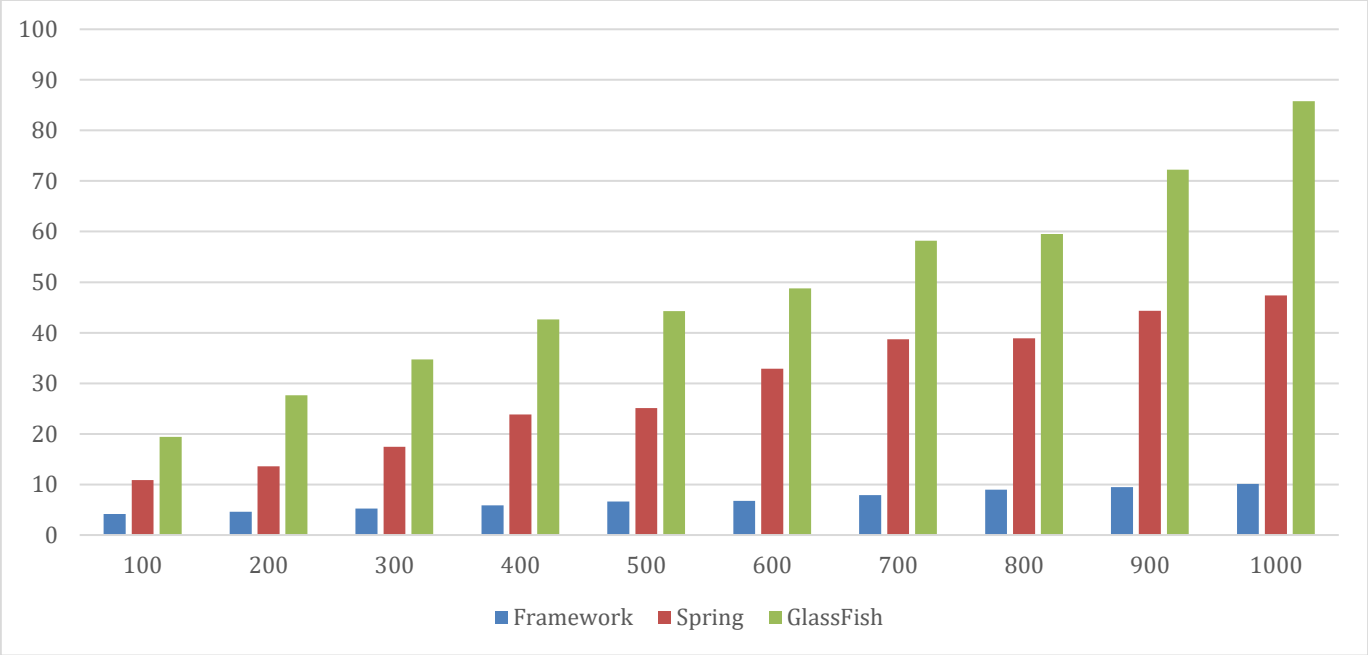


Рисунок 4.1 – Гістограма часу запуску додатків з однаковим функціоналом найрозповсюдженіших фреймворків та розробки даної дисертації

З ростом кількості ланцюжків розрив тільки зростає. Так у випадку 3000 ланцюжків час запуску фреймворку складає 21.241 що на 133 секунди швидше ніж Spring та на 246 секунд швидше ніж GlassFish.

Очевидно що з таким часом запуску який мають існуючі фреймворки їх неможливо застосовувати для WEB сайтів. Жоден користувач не буде чекати

десятки секунд відповіді на свій запит. У випадку службових WEB сайтів подібний час очікування також не є прийнятним.

Затримка в секунди також не є приємною, але в рамках оптимізації витрат є прийнятною для службових цілей та для комерційних стайтів що обмежені у бюджеті. При лінивому завантаженні значення найгіршого часу відповіді знижено приблизно на 25 відсотків. На рисунку 4.2. видно відносну різницю часу запуску при лінивому та жадібному завантаженнях.

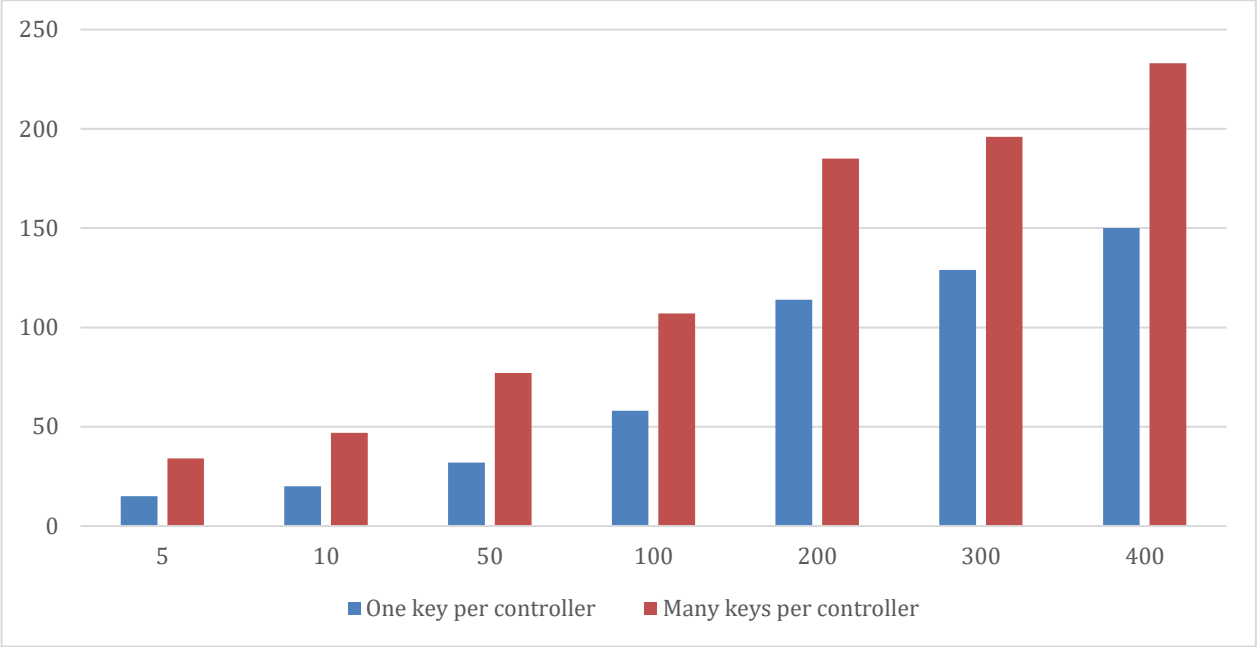


Рисунок 4.2 – Час запуску при лінивому та жадібному завантаженнях залежно від кількості ланцюжків

Додатково треба враховувати що перші запити будуть повільнішими. В найгіршому випадку для обробки першого запиту доведеться виконати повну ініціалізацію та як результат сумарний час відповіді буде рівним часу жадібного завантаження.

В середньому час відповіді на запит був би в двічі нижчий за теорією ймовірностей оскільки пошук йде почергово по класам зі складністю O(n) в

рамках модуля, а значить вірогідність що потрібний компонент опиниться в першій половині рівна вірогідності що він опиниться в другій половині. Попередньо іде відкидання частини компонентів за модулями і відповідно частина компонентів відкидається за логарифмічною складністю. При рівномірному розподілі компонентів в рамках всіх 4 модулів, сумарний час запуску та відповіді на перший запит в середньому буде приблизно на 15% швидшим.

Окрім цього завдяки фоновій багато поточній ініціалізації з кожною секундою роботи серверу цей час стрімко знижується і вже через додатковий час рівний приблизно 33 відсоткам від лінивого часу запуску час відповіді на перший запит не відрізнятиметься від часу відповіді при жадібній ініціалізації. Тож в цей момент він зрівняється з часом відповіді існуючих фреймворків.

4.2 Ефективність зниження часу запуску за рахунок архітектурного рішення підтримки REST

Для визначення ефективності зниження часу запуску за рахунок архітектурного рішення в рамках підтримки REST було проведено кілька експериментів.

Було створено тестовий проект з модифікованих REST контролерів що мають по 5 методів відповідних REST архітектурі з текстовим параметром ключа що відповідає адресі запиту який цей метод має обробляти.

Аналогічний експеримент було проведено з ініціалізацією контролерів що відповідають концепції запропонованій в даній дисертації. На рисунку 4.3 запропоновано результати часу потрібного на ініціалізацію хештаблиці для команди в мілісекундах.

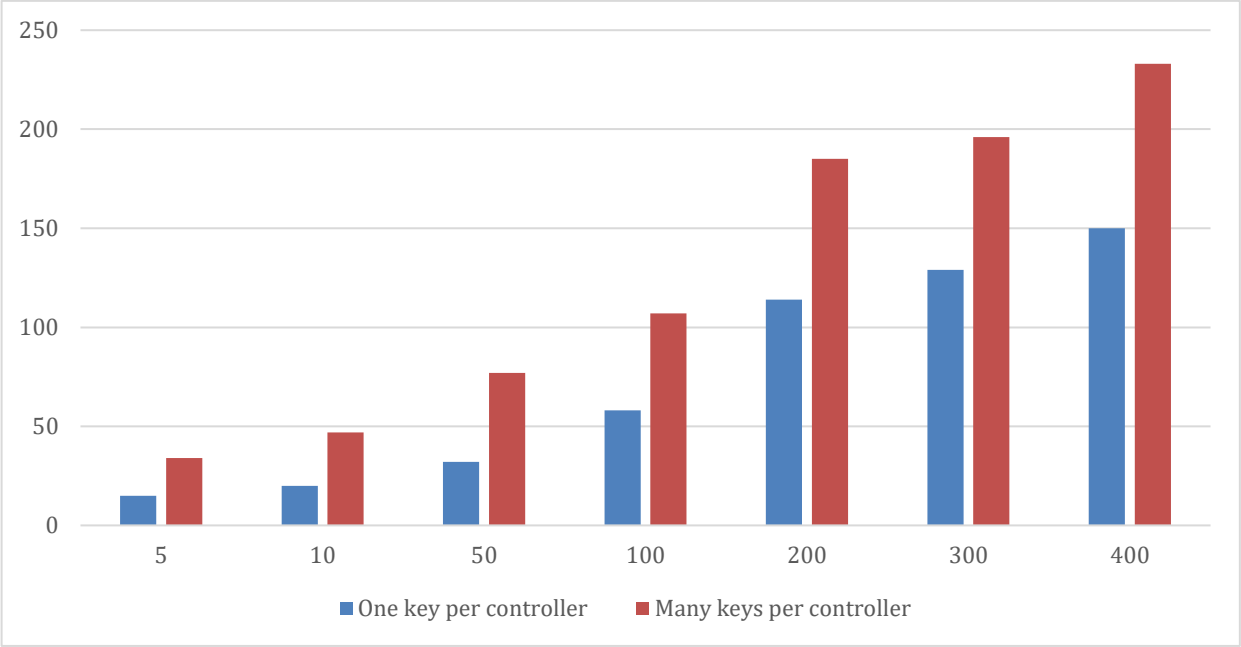


Рисунок 4.3 – Час ініціалізації хеш-таблиці команди для контролерів в рамках існуючої концепції та концепції запропонованої в дисертації

З графіку видно суттєву перевагу запропонованого архітектурного рішення в порівнянні з існуючим підходом реалізованим в аналогічних фреймворках. Також можна помітити що за рахунок оптимізації що проводить java машина при великій кількості подібних дій з ростом кількості контролерів відносний розрив по часу незначно зменшується.

Після чого для цих же тестових сценаріїв було виміряно ефективність по оперативній пам’яті. Запропоноване рішення є приблизно 2.2 рази ефективнішим за існуюче рішення. На рисунку 4.4 можна побачити результат вимірів у байтах.

Незважаючи на таку різницю у пам’яті тест часу отримання обробника за ключем показав практично однаковий результат. Це очікувано оскільки оскільки апроксимовано час пошуку у хештаблиці не залежить від кількості елементів в ній та має алгоритмічну складність O(1).

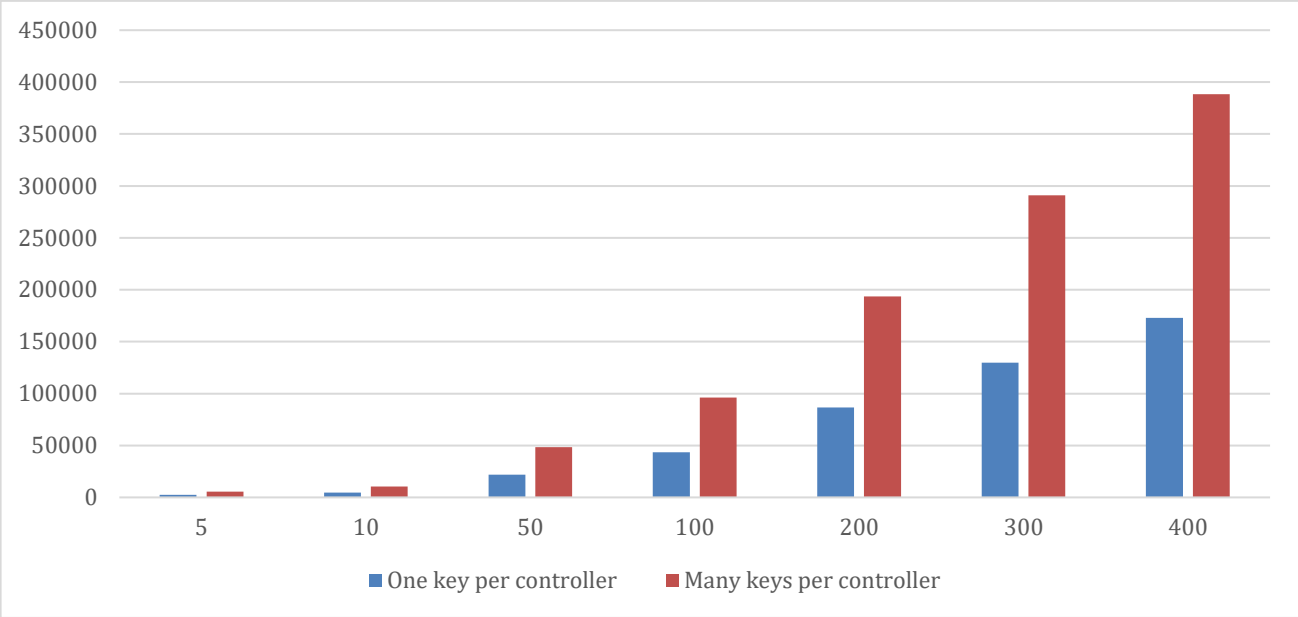


Рисунок 4.4 – Об’єм пам’яті що займає хеш-таблиця команди для контролерів в рамках існуючої концепції та концепції запропонованої в дисертації

4.3 Зниження витрат на утримання за рахунок покращення підтримуваності коду

Прямий виграш у людино годинах порахувати неможливо оскільки він буде відрізнятись від проекту до проекту через специфіку задач та рівень компетенції спеціалістів що його розробляють. Проте ми можемо проаналізувати ступінь його ефективності для цільових типів проектів.

Очевидно що у подібній розробці в першу чергу зацікавлені ті проекти котрі обмежені у фінансуванні. На таких проектах часто економлять і на технічних спеціалістах. Менш кваліфіковані фахівці роблять більше архітектурних помилок. Тому для цих проектів ефективність запропонованих рішень буде вищою.

З точки зору специфіки проектів архітектурні рішення у підтримці REST дадуть найбільший ефект в рамках службових WEB додатків. Оскільки одним з дуже розповсюджених сценаріїв роботи цих додатків є ручне маніпулювання

інформацією в базі даних. Наприклад коли користувач вказав не вірну адресу в замовленні, чи це замовлення взагалі треба видалити і створити нове в телефонному режимі. Для реалізації цих операцій потрібно створювати велику кількість форм для CRUD операцій з сутностями бази даних.

Розширена підтримка WEB форм в купі з коректними контролерами для архітектурного стилю REST дадуть значне зниження витрат на підтримку цих рішень для цього типу проектів.

Архітектурне рішення передачі повідомлень між сесіями WEB сокетів та групами цих сесій робить ефективним застосування цього фреймворку для цільової групи низько бюджетних онлайн ігор. А також різних реактивних рішень для обробки даних.

4.4 Оцінка потенціалу зниження в витрат за рахунок застосування спотових машин

Аналіз доцільно провести з урахуванням особливостей різних задач. Оскільки для різних задач будуть відрізнятись необхідні комп’ютерні потужності для забезпечення роботи. Далі розглянуто ті задачі у яких є доцільним застосування запропонованого у дисертації рішення.

4.4.1 Невеликі WEB сайти

Для забезпечення функціонування таких додатків буде достатньо мінімальної конфігурації призначеної для цього типу задач. Провайдер хмарних послуг рекомендує для задач цього типу віртуальні машини сімейства призначеного для загальних задач. В них пропонується баланс обчислювальних

потужностей, оперативної пам’яті та мережевих ресурсів тому і добре підходять для задач що вимагають їх у рівних пропорціях одною з яких є WEB сайти.[31]

Най новішим та най ефективнішим поколінням сімейства загальних конфігурацій що пропонує AWS є конфігурація «m7g». Мінімальною за об’ємом ресурсів є «m7g.medium». Вона має лише одне ядро процесору 4 гігабайти оперативної пам’яті та конкурентну швидкість передачі даних.[31]

Її вартість складає $357,4 доларів на рік за звичайним тарифним планом та 147,2 у випадку спотового тарифного плану. Інформація актуальна на кінець 2023 року [20,32]

Таким чином для цієї задачі застосування фреймворку в купі із спотовими машинами AWS дає можливість знизити витрати на утримання у 2,43 рази. Із частотою переривання менше 5 відсотків[33] Аналогічна конфігурація в google cloud буде коштувати 416,8 та 126 доларів на рік, тобто в 3,3 рази дешевше.[19] Для Azure 364,4 та 91,1 відповідно що в 4 рази дешевше.[34]

4.4.2 WEB застосунки для управління бізнес процесами

Для великих WEB додатків призначених для кінцевого користувача вірогідність застосування запропонованої розробки досить не велика. Проте його застосування в рамках додатків для організації внутрішніх бізнес процесів. Наприклад систем організації підтримки замовлень, управління балансами та номенклатурою товарів на складі, тощо.

Для цих застосунків також як і для невеликих WEB сайтів краще всього підходять віртуальні машини загальної конфігурації, але більшої потужності. Оскільки додатки такого типу можуть бути дуже великими, для дослідження було

взято най потужнішу конфігурацію «c7g.16xlarge».[31] Вона має 64 ядра 128 гігабайт пам’яті та високу мережеву пропускну здатність.

За рік це складе 20323,2 долари та 8059,2 долари для серверу за запитом та для спотового серверу відповідно. Тобто для цієї задачі при застосуванням сервісу AWS можна досягти зниження витрат у 2,52 рази. Найближча подібна конфігурація запропонована Microsoft Azure коштує 19804,23 за запитом та 2887,29 для спотових машин тобто в 6,8 разів дешевше.[36] Для Google cloud це 25008,05 та 7565,49, що у 3,3 рази менше.[19]

4.4.3 Stateless застосунки для блокової обробки даних

Фреймфорк може додати додаткову ефективність при застосуванні у задачах пакетної обробки великих даних коли не потрібно збереження стану. Тобто управляючий сервіс медіатор відправляє блок даних для обробки після чого отримує у відповідь агрегований результат обробки блоку та виконує подальші дії.

Для таких задач і без запропонованої розробки часто застосовуються спотові машини. Проте при дійсно великих потужностях кожна секунда яку сервер замість обробки даних витрачає на перезапуск це додаткові втрати. І відповідно нижчий час запуску тут може дозволити знизити витрати шляхом вищого коефіцієнту корисної дії в рамках використання обчислювальних ресурсів.

Для подібних задач най вірогідніше використання серверів оптимізованих за обчислювальними потужностями. Окрім цього в цих цілях можуть застосовуватись і сервери оптимізовані за оперативною пам’яттю для задач де потрібно зберігати багато проміжних даних в процесі обробки.

Для найпотужнішої оптимізованої під обчислення конфігурації «p5.48xlarge» що пропонує AWS вартість за годину складає 98.32 доларів за

запитом та 77.6921 за потовим тарифом. При цьому частота переривання складає більше 20 відсотків.[33] Для конфігурації оптимізованої по пам’яті «r7gd.16xlarge» частота переривань складає 10-15 відсотків.[33] Вартість на 60 відсотків нижча.[33]

Інформацію про кількість переривань на одиницю часу AWS не надає, але вона є значною і відповідно виграш ефективно використаного робочого часу також значний.

4.5 Перевірка зручності для розробки в рамках переносу тестового WEB додатку

Для цієї перевірки було взято розроблений кілька років тому тренувальний проект. Він надає доступ до послуг по отриманню і відправці вантажів. Не зарегістрованому користувачу доступна сторінка калькулятору вартості та часу, необхідних для виконання доставки з заданими параметрами. Зарегістрований користувач у себе в кабінеті може створити запит на доставку вантажу, оплатити сформовані для нього рахунки, підтвердити отримання доставок адресованих йому та переглянути статистику платежів.

Додаток побудовано за класичною тришаровою архітектурою з застосуванням MVC патерну для побудови представлень. Містить 13 контролерів, 6 сервісів та 5 класів доступу до бази даних, по одному для кожної ключової сутності. Структура бази даних запропонована на рисунку 4.5.

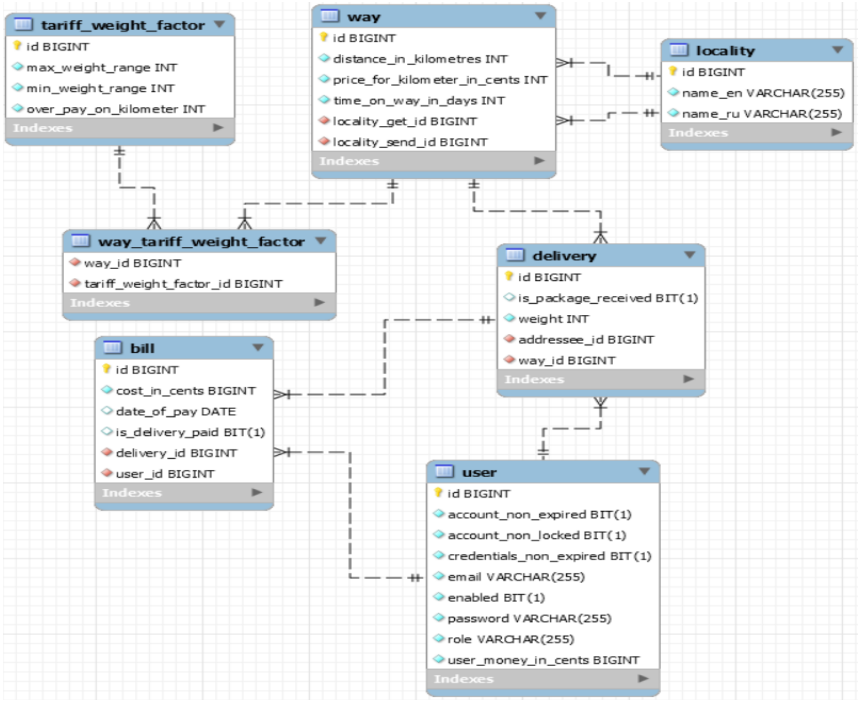


Рисунок 4.5 – ER діаграма сутностей бази даних

За створення візуального представлення по архітектурі MVC відповідають 18 jsp сервлетів. На рисунку 4.6 запропоновано представлення для сторінки відправлення доставки. З неї можна отримати додаткове суб'єктивне візуальне уявлення про порядок розміру WEB додатку застосованого для тесту.

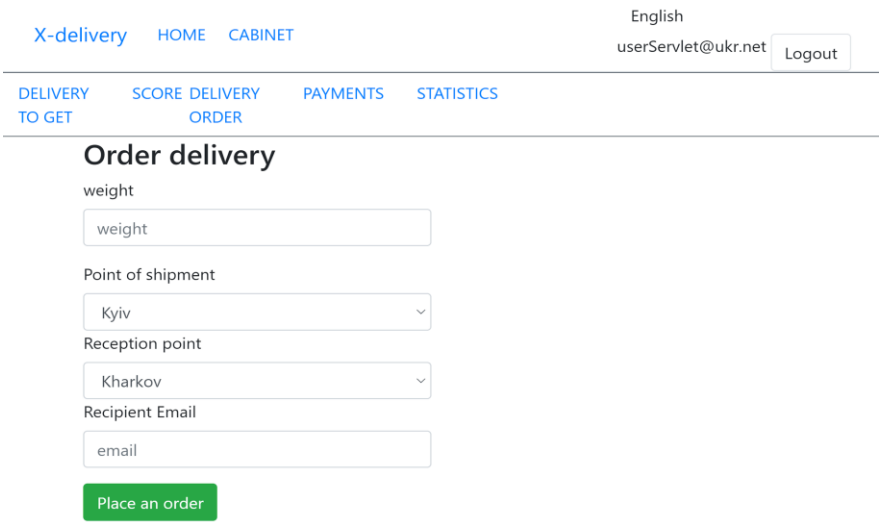


Рисунок 4.6 – Сторінка відправлення доставки тестового додатку

Це функціонал типового не великого сайту – інформаційні сторінки, досить прості розрахунки та інформація призначена індивідуально для кожного користувача. Експеримент перенесення цього проекту на фреймворк запропонований як результат даної дисертації дасть можливість оцінити кількість зусиль для його перенесення. Після чого можна буде зробити висновок про раціональність перенесення уже створених з застосунків на базі інших фреймворків.

Процес переносу склався з таких етапів:

- копіювання коду вихідного проекту у новий проект.
- заміна з допомогою можливостей середовища розробки всіх анотацій фреймворку Spring на відповідні їм анотації.
- агрегування анотацій кінцевих точок з методів контролерів на рівень класу.
- створити реалізації репозиторіїв, SQL запити, та конвертери до команд в рамках репозиторіїв.

На рисунку 4.7 зображено логи з запуску що демонструють виграш у часі запуску цього тестового серверу з використанням Spring та результату дисертації. Проект на фреймворку Spring запускається повільніше в 2,1 рази.

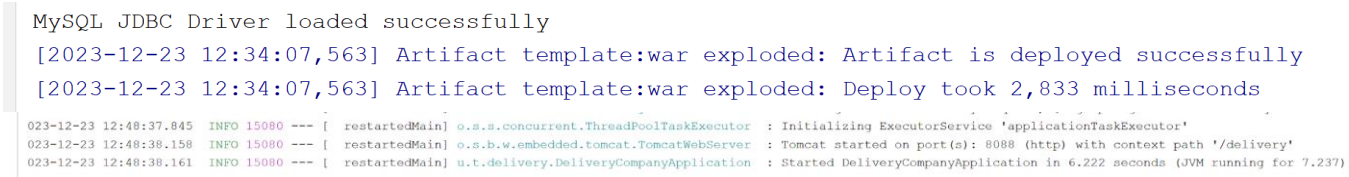


Рисунок 4.7 – Журнал часу запуску WEB додатку управління доставками з допомогою фреймворку дисертації та фреймворку Spring

Процес перетворення зайняв близько 5 годин. Людино-година роботи розробника достатньої кваліфікації в Україні коштує близько 10 доларів на годину. Тобто вартість перенесення невеликого сайту складає порядку 50 доларів.

Базуючись на інформації викладеній вище виграш в утриманні невеликого WEB сайту складає порядку 210 доларів на рік тобто після 4 місяців роботи вартість перенесення буде компенсована.

4.6 Висновки розділу

Запропоноване рішення дійсно дає зниження часу запуску. Близько двох разів в порівнянні з кращою розповсюдженою альтернативою якою є фреймворк Spring.

Покращена та розширена підтримка архітектури REST не тільки знижує час ініціалізації та об’єм пам’яті кешу для контролерів більше ніж в двічи, а й веде до покращення якості клієнтського коду і як наслідок підтримуваності.

Аналітично було перевірено що покращення підтримуваності запропоновані даним в роботі матимуть найбільший позитивний ефект якраз в тих сферах де даний фреймворк може бути ефективно застосований.

З результату дослідження ціноутворення на віртуальні машини підтверджено, що зниження витрат може досягати 80 відсотків при застосуванні спотових віртуальних машин. Разом із зниженим часом запуску це дає можливість значно знизити витрати на утримання серверів у сфері WEB застосувань де до цього їх застосування було практично неможливим.

Зниження витрат також може бути отримано і в сферах де і раніше застосовувались спотові віртуальні машини за рахунок більшого часу корисної

роботи серверу що швидше запускається. Для дуже потужних конфігурацій в абсолютних цифрах ці суми можуть бути досить великими.

5 РОЗРОБКА СТАРТАП ПРОЕКТУ

5.1 Опис ідеї проекту

Таблиця 5.1 – Опис ідеї стартап проекту

Зміст ідеї	Напрямки застосування	Вигоди для користувача
Фреймворк що дозволяє знизити витрати на утримання Java WEB серверів.	За рахунок мінімального часу запуску є можливість використовувати для WEB серверу спотові віртуальні машини.	Зниження витрат на хмарний сервер більше ніж у два рази. Нижчий час від рішення про запуск нових змін до початку їх роботи.
	Неможливість допущення деяких розповсюджених архітектурних помилок завдяки.	Зниження витрат на внесення змін у існуючий код. Зниження витрат за рахунок найму менш кваліфікованих розробників.
	Покращення та розширення підтримки архітектурних сценаріїв що упущені в існуючих фреймворках.	

Таблиця 5.2 – Визначення сильних, слабких та нейтральних характеристик

Характеристика	Потенційні товари / концепції конкурентів		
	Мій Фреймворк	Spring	Реалізації Jakarta EE
Час запуску серверу	Strong	Weak	Weak
Можливості інтеграції з різними інструментами	Weak	Strong	Neutral
Стандартизованість	Neutral	Neutral	Strong
Ціна ліцензії	Strong	Strong	Weak
Протидія архітектурним помилкам	Strong	Weak	Weak

Продовження таблиці 5.2

Характеристика	Потенційні товари / концепції конкурентів		
	Мій Фреймворк	Spring	Реалізації Jakarta EE
Якість підтримки REST архітектури	Strong	Neutral	Neutral
Зручність конфігурування	Strong	Strong	Neutral
Розповсюдженість	Weak	Strong	Neutral
Допоміжні супутні інструменти	Weak	Strong	Neutral

5.2 Технологічний аудит ідеї проекту

Таблиця 5.3 – Технологічна здійсненність проекту

№ п/п	Можливість проекту	Технології її реалізації	Наявність технологій	Доступність технологій
1	Підтримка архітектури MVC.	JSP, Java Reflection API, servlet api	Наявна	Доступна
2	Підтримка контролерів за архітектурним стилем REST.	Java Reflection API, servlet api	Наявна	Доступна
3	Підтримка роботи в асинхронному режимі в рамках протоколу TCP	WebSocket Java API, Java Reflection	Наявна	Доступна
4	Взаємодія з реляційними базами даних.	JDBC API, JDBC Driver	Наявна	Доступна
Висновок: для реалізації проекту наявні і доступні всі необхідні технології, тому його реалізувати можливо.				

5.3 Аналіз ринкових можливостей запуску стартап-проекту

Таблиця 5.4 – Попередня характеристика потенційного ринку стартап-проекту

№ п/п	Показники стану ринку (найменування)	Характеристика
1	Кількість головних гравців, од	10
2	Загальний обсяг продаж, USD/year	530,44 мільйона доларів США на 2022 та, за прогнозами, досягне 9 034,10 мільйонів доларів США до 2030 року.[37]
3	Динаміка ринку (якісна оцінка)	Зростає
4	Наявність обмежень для входу (вказати характер обмежень)	Немає
5	Специфічні вимоги до стандартизації та сертифікації	Немає

Таблиця 5.5 – Характеристика потенційних клієнтів стартап-проекту

No п/п	Потреба, що формує ринок	Цільова аудиторія (цільові сегменти ринку)	Особливості поведінки потенційних цільових груп клієнтів	Вимоги споживачів до товару
1	Утримання Java серверів.	Невеликі WEB сайти з обмеженим бюджетом	В силу обмеженості бюджетів ставлять перший пріоритет на оптимізацію ресурсів.	Низька вартість, наявність базових функцій
		Низько бюджетні онлайн ігри		
		Службові WEB додатки	Не вимагають надання максимального комфорту користувачів та готові миритися з певними жертвами для зниження витрат.	Коректність роботи, надійність, ефективна підтримка

Таблиця 5.6 – Фактори загроз

№ п/п	Фактор	Зміст загрози	Можлива реакція компанії
1	Особливості ціноутворення спотових віртуальних машин	При досягненні значного успіху попит на спотові віртуальні машини виросте що приведе до зниження рентабельності ефективності оптимізації витрат з допомогою фреймворку.	Зміна правил ліцензування для збільшення рентабельності користувачів.
2	Низькі можливості в рамках інтеграції з сторонніми інструментами	Для багатьох користувачів відсутність можливості легкої інтеграції з потрібними їм інструментами може стати не прийнятною	Надати можливість безкоштовного застосування фреймворку тим хто доповнить фреймворк необхідним для інтеграції з стороннім інструментом функціоналом.

Таблиця 5.7 – Фактори можливостей

№ п/п	Фактор	Зміст можливості	Можлива реакція компанії
1	Економічна криза	Компанії намагаються оптимізувати витрати щоб не збанкрутіти	Додання нових можливостей що є найпотрібнішими у стилі подібному до реалізації в найпопулярніших конкурентів щоб полегшити перехід.
2	Стрімке зростання ринку WEB додатків	Велика кількість проектів що бажають швидко вийти на ринок	Активно розповсюджувати інформацію про те як швидко та ефективно цього можна досягти з використанням запропонованого фреймворку.

Таблиця 5.8 – Ступеневий аналіз конкуренції на ринку

Особливості конкурентного середовища	В чому проявляється дана характеристика	Вплив на діяльність підприємства (можливі дії компанії, щоб бути конкурентоспроможною)
1. Вказати тип конкуренції - Олігополія	На ринку доступні подібні за функціоналом сервери, але серверу з такою концепцією та аналогічним функціоналом немує.	Забезпечити базовий функціонал, а також і якість і стабільність роботи. Додання нових функцій не повинно негативно вплавати на ключову концепцію оскільки єдина конкурентна перевага це знижена вартість підтримка, а конкуренція за функціоналом є неможливою.
2. За рівнем конкурентної боротьби - глобальний	Хмарні сервери застосовуються по всьому світу. Конкурентна перевага мінімального часу запуску актуальна у цьому масштабі.	Запропонувати легкий перехід з уже існуючих рішень для вже існуючої аудиторії
3. За галузевою ознакою - міжгалузева	Розробка призначена для будь яких WEB серверів.	Розвивати унікальні сильні сторони – мінімальний час запуску та звуження поля для архітектурних помилок.
4. Конкуренція за видами товарів: - товарно-родова	Розробка пропонує типовий для WEB фреймфорків функціонал з не функціональною відмінністю у вигляді зниженого часу запуску та архітектурних покращень	Якість базового функціоналу та розвиток переваг що відрізняють продукт від ринкових
5. За характером конкурентних переваг - нецінова	Продукт пропонує знижений час запуску та концепцію прибирання зайвої свободи у розробнику коду.	Підтримка та системна робота по роз’ясненню та допомозі у переході з існуючих аналогів.
6. За інтенсивністю - не марочна	Конкуренція відбувається за якісними характеристиками.	Надання якісної переваги у часі запуску та полегшенні підтримки.

Таблиця 5.9 – Аналіз конкуренції в галузі за М. Портером

Складові аналізу	Прямі конкуренти в галузі	Потенційні конкуренти	Постачальники	Клієнти	Товари-замінники
	Spring, Jakarta EE	Spark Java	Відсутні	Проекти що шукають можливостей мінімізації витрат на утримання	Включення прямими конкурентами з допомогою конфігурації подібних можливостей
Висновки	В прямих конкурентів немає тих можливостей що пропонуються	Конкурує в полі розподілених обчислень що не є пріоритетною областю.	-	Якість та надійність роботи. Функціональні і не функціональні переваги	Додати подібні функції в товари замінники буде дуже не рентабельно і займе багато часу.

Таблиця 5.10 – Обґрунтування факторів конкурентоспроможності

No п/п	Фактор конкурентоспроможності	Обґрунтування (наведення чинників, що роблять фактор для порівняння конкурентних проектів значущим)
1	Нижчий час запуску серверу	Клієнтам відкривається можливість застосування спотових віртуальних машин, що в свою чергу дає можливість знизити витрати на утримання серверу більше ніж в 2 рази.
2	Знижена кількість можливостей для помилок	Є можливість наймати більш дешевих працівників, та знизити витрати на внесення додаткових змін завдяки зниженню кількості архітектурних помилок.

Таблиця 5.11 – Порівняльний аналіз сильних та слабких сторін ПЗ

№ п/п	Фактор конкурентоспроможності	Бали 1-20	Рейтинг товарів-конкурентів у порівнянні з розроблюваним ПЗ						
			-3	-2	-1	0	+1	+2	+3
1	Нижчий час запуску серверу	20	+						
2	Знижена кількість можливостей для помилок	15		+					

Таблиця 5.12 – SWOT-аналіз стартап проекту

Сильні сторони:	Слабкі сторони:
- низький час запуску - знижена кількість можливостей для помилок	- можливість інтеграції з іншими інструментами - відсутність допоміжних інструментів
Можливості:	Загрози:
- економічна криза - стрімке зростання ринку WEB додатків	- в наслідок успіху різниця між вартістю серверів на вимогу та спотовими може скоротитись. - недостатність для клієнтів можливостей в рамках інтеграції з сторонніми інструментами

Таблиця 5.13 – Альтернативи ринкового впровадження стартап проекту

№ п/п	Альтернатива (орієнтовний комплекс заходів ринкової поведінки)	Ймовірність отримання ресурсів	Строки реалізації
1	Укладення партнерських угод з провідними виробниками WEB фреймворків для інтеграції запропонованих ідей як опції в їх фреймворку.	Низька	12 місяців
2	Проведення презентацій та доповідей на важливих конференціях для залучення уваги до фреймворку.	Висока	6 місяців
3	Запуск краудфандингової кампанії для збору коштів на активне рекламування та впровадження фреймворку.	Середня	4 місяці
4	Укладення угод з навчальними установами та організаціями для включення вашого фреймворку у навчальні програми.	Середня	12 місяців

5.4 Розроблення ринкової стратегії проекту

Таблиця 5.14 – Вибір цільових груп потенційних споживачів

№ п/п	Опис профілю цільової групи потенційних клієнтів	Готовність споживачів прийняти продукт	Орієнтовний попит в межах цільової групи (сегменту)	Інтенсивність конкуренції в сегменті	Простота входу в сегмент
1	Власники WEB додатків обмежені у бюджеті	Готові	Високий	Середня	Просто
2	Власники WEB додатків для внутрішньої інфраструктури ведення бізнесу	Потребують переговорів	Високий	Висока	Складно

Продовження таблиці 5.14

№ п/п	Опис профілю цільової групи потенційних клієнтів	Готовність споживачів прийняти продукт	Орієнтовний попит в межах цільової групи (сегменту)	Інтенсивність конкуренції в сегменті	Простота входу в сегмент
3	Власники великих WEB додатків	Не готові	Низький	Висока	Не можливо
4	Проекти обчислень великих даних	Потребують переговорів	Середній	Висока	Складно
Обрано цільову групу: Власники WEB додатків обмежені у бюджеті.					

Таблиця 5.15 – Визначення базової стратегії розвитку

№ п/п	Обрана альтернатива розвитку проекту	Стратегія охоплення ринку	Ключові конкурентоспроможні позиції відповідно до обраної альтернативи	Базова стратегія розвитку
1	Орієнтація на власників низько бюджетних WEB сайтів	Стратегія концентрованого маркетингу	Зниження прямих витрат на утримання та на внесення змін в існуючий код	Стратегія спеціалізації

Таблиця 5.16 – Визначення базової стратегій конкурентної поведінки

№ п/п	Чи є стартап «першопрохідцем» на ринку?	Чи буде дана компанія шукати нових споживачів, або забирати існуючих у конкурентів?	Чи буде дана компанія копіювати основні характеристики товару конкурентів і які?	Стратегія конкурентної поведінки
1	Ні	Пошук нових користувачів та конкуренція за існуючих у конкурентів	Так. Копіюються основні функціональні можливості і додається унікальна характеристика мінімальний час запуску.	Стратегія займання конкурентної ніші

Таблиця 5.17 – Визначення стратегії позиціонування

№ п/п	Вимоги до товару від цільової аудиторії	Базова стратегія розвитку	Ключові конкурентоспроможні позиції стартап-проекту	Вибір асоціацій, які мають сформувати комплексну позицію власного проекту
1	Зниження вартості утримання серверу, надійність, коректність роботи, технічна підтримка, подібність до уже існуючих рішень	Стратегія спеціалізації	Користувачам потрібен фреймворк з мінімальним часом запуску щоб знизити витрати шляхом застосування спотових серверів.	Економія коштів, Мінімум помилок, Простота

5.5 Розроблення маркетингової програми проекту

Таблиця 5.18 – Визначення ключових переваг концепції потенційного товару

№ п/п	Потреба	Вигода, яку пропонує товар	Ключові переваги перед конкурентами (існуючі, або такі, що потрібно створити)
1	Зниження витрат	Мінімальний час запуску додатку і як наслідок можливість застосування спотових віртуальні машин. Зниження кількості архітектурних помилок в коді додатку.	Нова концепція мінімального часу запуску. Архітектурні покращення по зниженні свободи розробника з метою унеможливлення типових архітектурних помилок

Таблиця 5.19 – Опис трьох рівнів моделі товару

Рівні товару	Сутність та складові		
I. Товар за задумом	Фреймворк для зниження витрат на утримання Java WEB серверів		
II. Товар у реальному виконанні	Властивості/характеристики	М/Нм	Вр/Тх/Тл/Е/Ор
	1. Дешевизна 2. Оптимізація	1. М 2. М	1. Тл 2. Тх
	Якість: математичне та програмне забезпечення		
	Марка: F&K framework		
III. Товар із підкріпленням	Користувач може ознайомитися з можливостями використання цього забезпечення шляхом перегляду рекламних та навчальних відеороликів та виступів на конференціях. Після користувач отримує супровід від підтримки.		
За рахунок чого потенційний товар буде захищено від копіювання: захист інтелектуальної власності			

Таблиця 5.20 – Формування системи збуту

№ п/п	Специфіка закупівельної поведінки цільових клієнтів	Функції збуту, які має виконувати постачальник товару	Глибина каналу збуту	Оптимальна система збуту
1	Надання безкоштовної версії для використання в рамках циклу розробки та тестування	Легкість отримання, прозорість ціноутворення, зручність оплати	Низька	Безпосередня

Таблиця 5.21 – Концепція маркетингових комунікацій

№ п/п	Специфіка поведінки цільових клієнтів	Канали комунікацій, що використовують цільові клієнти	Ключові позиції, обрані для позиціонування	Завдання рекламного повідомлення	Концепція рекламного звернення
1	Орієнтація на надійний базовий функціонал та економію	Сайти компаній, спеціалізовані блоги, конференції	Здешевлення утримання серверів	Пояснити як за рахунок зниження часу запуску та кількості архітектурних помилок дає зниження вартості утримання	«Мінімізуй витрати на сервер», «Ефективно використовуй комп’ютерний і людський ресурс»

5.6 Висновки розділу

В розділі було проведено аналіз ринку, де передбачається пропонувати розроблений фреймворк. Було розглянуто переваги та недоліки у порівнянні з аналогами та конкурентами. На основі цього було виявлено та враховано можливі ризики та варіанти розвитку.

Потенціал для комерціалізації є досить високим оскільки прямих конкурентів у вузькій ніші низько бюджетних WEB додатків немає. Попит є достатнім та має потенціал до росту завдяки тому що наразі економіка почала відновлення після кризи.

Для впровадження доцільно обрати альтернативу просвітницької та роз’яснювальної діяльності на тематичних площадках. Проведення презентацій та доповідей на конференціях для залучення уваги до фреймворку. Публікація статей на тематичних сайтах та форумах.

ВИСНОВКИ

В даній роботі було розглянуто потенціал можливості здешевити вартість утримання Java WEB серверів. В процесі дослідження було виявлено та реалізовано два напрямки у яких є потенціал для зниження витрат.

Першим напрямком є застосування спотових віртуальних машин що є більш ніж у 2 рази дешевшими в порівнянні з машинами за запитом. Було винайдено концепцію побудови WEB фреймворку таким чином щоб значно знизити час запуску зберігши необхідний функціонал.

Це було досягнуто за рахунок застосування протилежного класичному підходу побудови фреймворків, а саме зміщенні фокусу зі швидкості обробки запиту на швидкість запуску. Завдяки ряду оптимізаційних рішень наслідки компромісів були успішно мінімізовані.

Результуючий час запуску у порівнянні з найближчими аналогами виявився в рази меншим.

Другим виявленим напрямком зниження витрат на утримання стало архітектурне обмеження можливостей користувача для уникнення розповсюджених архітектурних помилок. Їх мінімізація дозволить знизити витрати на підтримку шляхом полегшення внесення змін в уже існуючий код та меншій кількості дефектів під час розробки.

В рамках цього напрямку було ліквідовано циклічні залежності, некоректне поєднання ключів для REST точок та контролерів. Також було розширено підтримку архітектурного стилю REST шляхом компенсуванням недоліку мови HTML наданням можливості відправки запитів WEB формами до REST контролерів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1) Singh, Chamkaur, Neeraj Sharma, and Narender Kumar. "Analysis of software maintenance cost affecting factors and estimation models." Int. J. Sci. Technol. Res 8.9 (2019): 276-281.

2) Schmidt, Douglas C., Aniruddha Gokhale, and Balachandran Natarajan. "Frameworks: Why they are important and how to apply them effectively." ACM Queue magazine 2.5 (2004).

3) Dustin, Elfriede, Thom Garrett, and Bernie Gauf. Implementing automated software testing: How to save time and lower costs while raising quality. Pearson Education, 2009.

4) Dhar, Subhankar. "From outsourcing to Cloud computing: evolution of IT services." Management research review 35.8 (2012): 664-675.

5) Buttar, Ahmed Mateen, et al. "Optimization of DevOps Transformation for Cloud-Based Applications." Electronics 12.2 (2023): 357.

6) Das, Sumit, et al. "Applications of artificial intelligence in machine learning: review and prospect." International Journal of Computer Applications 115.9 (2015).

7) Goncalves, Antonio. Beginning Java EE 7. Apress, 2013.

8) Gutierrez, Felipe. Introducing spring framework: a primer. Apress, 2014.

9) Jacobi, Jonas, and John R. Fallows. "Pro JSF and Ajax." (2006).

10) Reelsen, Alexander. Play Framework Cookbook. Packt Publishing, 2011.

11) Singh, Harmeet, and Syed Imtiyaz Hassan. "Effect of solid design principles on quality of software: An empirical assessment." International Journal of Scientific & Engineering Research 6.4 (2015): 1321-1324.

12) Deck, Paul. "Spring MVC: a tutorial." (2016).

13) Reese, George. Database Programming with JDBC and JAVA. " O'Reilly Media, Inc.", 2000.

14) Dobles, Ignacio, Alexandra Martínez, and Christian Quesada-López. "Comparing the effort and effectiveness of automated and manual tests." 2019 14th Iberian Conference on Information Systems and Technologies (CISTI). IEEE, 2019.

15) Sether, Ayob. "Cloud computing benefits." Available at SSRN 2781593 (2016).

16) Saraswat, Manish, and R. C. Tripathi. "Cloud computing: Comparison and analysis of cloud service providers-AWs, Microsoft and Google." 2020 9th international conference system modeling and advancement in research trends (SMART). IEEE, 2020.

17) Zhang, Qi, et al. "Dynamic service placement in geographically distributed clouds." IEEE Journal on Selected Areas in Communications 31.12 (2013): 762-772.

18) Laatikainen, Gabriella, Arto Ojala, and Oleksiy Mazhelis. "Cloud services pricing models." Software Business. From Physical Products to Software Services and Solutions: 4th International Conference, ICSOB 2013, Potsdam, Germany, June 11-14, 2013. Proceedings 4. Springer Berlin Heidelberg, 2013.

19) Linux Virtual Machines Pricing. (2024). https://azure.microsoft.com/en-us/pricing/details/virtual-machines/linux/

20) Amazon EC2 Spot Instances Pricing. (2024). Amazon Web Services. Inc. https://aws.amazon.com/ec2/spot/pricing/

21) Compute Engine pricing. (2024). Google Cloud. Retrieved January 07, 2024 from https://cloud.google.com/compute/all-pricing

22) Muñoz, Mirna, and Mario Negrete Rodríguez. "A guidance to implement or reinforce a DevOps approach in organizations: A case study." Journal of Software: Evolution and Process (2021): e2342.

23) Cosmina, Iuliana, et al. Pro Spring 5: An in-depth guide to the Spring framework and its tools. Apress, 2017.

24) Seemann, M., & van Deursen, S. (2019). Dependency Injection Principles, Practices, and Patterns. Simon and Schuster.

25) Dey, Tamal. "A comparative analysis on modeling and implementing with MVC architecture." International Journal of Computer Applications 1 (2011): 44-49.

26) Helm, Richard, et al. Design patterns: Elements of reusable object-oriented software. Quebec: Braille Jymico Incorporated, 2000.

27) Li, Yue, Tian Tan, and Jingling Xue. "Understanding and analyzing java reflection." ACM Transactions on Software Engineering and Methodology (TOSEM) 28.2 (2019): 1-50.

28) Wilde, Erik, and Cesare Pautasso, eds. REST: from research to practice. Springer Science & Business Media, 2011.

29) Hickson, Anthony. HTML, The living standard. Lulu. com, 2014.

30) Репозиторій бібліотеки ava runtime metadata analysis https://github.com/ronmamo/reflections/

31) Instance types. (2024). Amazon Web Services. Inc. https://aws.amazon.com/ec2/instance-types/

32) Spot Instance advisor. (2024). Amazon Web Services. Inc. https://aws.amazon.com/ec2/spot/instance-advisor/

33) VM instance pricing. (2024). Google Cloud. https://cloud.google.com/compute/vm-instance-pricing

34) Java Web Frameworks Software Market Size And Forecast. (2023). Verified Market Research. https://www.verifiedmarketresearch.com/product/java-web-frameworks-software-market/

ДОДАТКИ

ДОДАТОК А

код основних частин програми

Код файлу ApplicationContextImpl.java що є точкою входу до ключового функціоналу фймворку.

```
public class ApplicationContextImpl implements ApplicationContext {

    private static final Logger log = LogManager.getLogger(ApplicationContextImpl.class);

    public static final String URL_STEP_DELIMITER = "/";

    public static final String REPLACEMENT_REGEX_FOR_REST_ENDPOINT_VARIABLE_PLACEHOLDER = "/";

    public static final String URL_STEP_SPLIT_REGEX = "\\" + URL_STEP_DELIMITER;

    public static final String REST_URL_VARIABLE_REGEX = "\\{[^/]*\\}";

    public static final String REST_ENDPOINT_VARIABLE_PLACEHOLDER_REGEX = "\\" + URL_STEP_DELIMITER + REST_URL_VARIABLE_REGEX + "\\" + URL_STEP_DELIMITER;

    public static final String REST_ENDPOINT_LUST_VARIABLE_PLACEHOLDER_REGEX = "\\" + URL_STEP_DELIMITER + REST_URL_VARIABLE_REGEX;

    public static final String REPLACEMENT_REGEX_FOR_REST_ENDPOINT_LUST_VARIABLE_PLACEHOLDER = "/";

    private final Map<Class<?>, Object> objectsCash;

    private final Map<String, Class<?>> controllerMap;//todo consider refactor this to hold type instead of link to object //todo consider rewrite it to annotation based approach

    private final Map<String, RestProcessorCashInfo> urlMatchPatternToCommandProcessorInfo;

    private final Map<String, CurrencyInfo> currencies;

    private final Map<String, Class<?>> messageTypeTcpCommandControllerMap;

    private final Map<String, Class<?>> messageTypeToMessageClass;

    private final Map<Class<?>, String> massageClassToCode;

    private final Set<Class<?>> processedCalsesSet = Collections.newSetFromMap(new ConcurrentHashMap<>());

    private final Class defaultEndpoint = PhantomController.class;

    private Config config;

    private ObjectFactory factory;

    private final ResourceBundle applicationConfigurationBundle = ResourceBundle.getBundle("application");


    private static final ApplicationContext singleToneApplicationContext;


    static {

        Map<Class<?>, Object> paramMap = new ConcurrentHashMap<>();

        ApplicationContext context = new ApplicationContextImpl(paramMap,

                new ConcurrentHashMap<>(), new CurrencyInfoFromFileLoader(), new ConcurrentHashMap<>(), new ConcurrentHashMap<>(), new ConcurrentHashMap<>());//todo ivan perhaps we do not need here concurrent hash maps

        ObjectFactory objectFactory = new ObjectFactoryImpl(context);

        context.setFactory(objectFactory);

        context.init();

        singleToneApplicationContext = context;
```

```
    }

    public static ApplicationContext getContext() {
        return singleToneApplicationContext;
    }

    private ApplicationContextImpl(Map<Class<?>, Object> preparedCash,
                                   Map<String, Class<?>> controllersPrepared,
                                   CurrencyInfoLoader currencyInfoLoader, Map<String, Class<?>> messageTypeTcpCommandControllerMap,
Map<String, Class<?>> messageTypeToMessageClass, Map<Class<?>, String> massageClassToCode) {
        this.messageTypeTcpCommandControllerMap = messageTypeTcpCommandControllerMap;
        this.messageTypeToMessageClass = messageTypeToMessageClass;
        this.massageClassToCode = massageClassToCode;
        log.debug("");

        this.controllerMap = controllersPrepared;
        this.objectsCash = preparedCash;
        currencies = currencyInfoLoader.getCurrencyInfo();
        urlMatchPatternToCommandProcessorInfo = new HashMap<>();
    }

    @Override
    public String getPropertyValue(String key) {
        return applicationConfigurationBundle.getString(key);
    }

    public void init() {
        objectsCash.put(ApplicationContext.class, this);
        log.debug("");
        initSingletonAnnotatedObjects(true);
        initNetworkDto(true);
        initHttpControllers(true);
        initRestControllers(true);
        initTcpEndpoints(true);
        if (Boolean.parseBoolean(getPropertyValue("infrastructure.initialize.components.in.parallel"))){
            Thread thread = new Thread(()->initSingletonAnnotatedObjects(false));
            thread.setPriority(Integer.parseInt(getPropertyValue("infrastructure.initialize.components.in.parallel.priority")));
            thread.start();
            thread = new Thread(()->initNetworkDto(false));
```

```
            thread.setPriority(Integer.parseInt(getPropertyValue("infrastructure.initialize.components.in.parallel.priority")));
            thread.start();
            thread = new Thread(()->initHttpControllers(false));
            thread.setPriority(Integer.parseInt(getPropertyValue("infrastructure.initialize.components.in.parallel.priority")));
            thread.start();
            thread = new Thread(()->initRestControllers(false));
            thread.setPriority(Integer.parseInt(getPropertyValue("infrastructure.initialize.components.in.parallel.priority")));
            thread.start();
            thread = new Thread(()->initTcpEndpoints(false));
            thread.setPriority(Integer.parseInt(getPropertyValue("infrastructure.initialize.components.in.parallel.priority")));
            thread.start();
        }
        startDemonThreads();
    }

    private void startDemonThreads() {
        if (!Boolean.parseBoolean(getPropertyValue("infrastructure.include.in.start.demon.threads"))){
            return;
        }
        for (Class<?> clazz : getConfig().getTypesAnnotatedWith(DemonThread.class)) {
            final Runnable runnable = (Runnable) getObject(clazz);
            new Thread(runnable).start();
        }
    }

    private void initTcpEndpoints(boolean shouldCheckLazyLoad) {
        if (shouldCheckLazyLoad && !Boolean.parseBoolean(getPropertyValue("infrastructure.include.in.start.tcp.controllers"))){
            return;
        }
        for (Class<?> clazz : getConfig().getSubTypesOf(AbstractTcpController.class)) {
            cashTcpControllerData(clazz);
        }
    }

    private void initRestControllers(boolean shouldCheckLazyLoad) {
        if (shouldCheckLazyLoad && !Boolean.parseBoolean(getPropertyValue("infrastructure.include.in.start.rest.controllers"))){
            return;
        }
        for (Class<?> clazz : getConfig().getTypesAnnotatedWith(RestEndpoint.class)) {
```

```
                RestEndpoint annotation = clazz.getAnnotation(RestEndpoint.class);
                for (String resourceFromAnnotation : annotation.resource()) {
                    String[] urlSteps = resourceFromAnnotation.split(URL_STEP_SPLIT_REGEX);
                    urlSteps = Arrays.copyOfRange(urlSteps, 1, urlSteps.length);
                    String restCommandPattern = resourceFromAnnotation.replaceAll(REST_ENDPOINT_VARIABLE_PLACEHOLDER_REGEX,
REPLACEMENT_REGEX_FOR_REST_ENDPOINT_VARIABLE_PLACEHOLDER)
                            .replaceAll(REST_ENDPOINT_LUST_VARIABLE_PLACEHOLDER_REGEX,
REPLACEMENT_REGEX_FOR_REST_ENDPOINT_LUST_VARIABLE_PLACEHOLDER);
                    cashRestCommandProcessor(clazz, restCommandPattern, urlSteps);
                    processedCalsesSet.add(clazz);
                }
            }
        }

        private void initHttpControllers(boolean shouldCheckLazyLoad) {
            if (shouldCheckLazyLoad && Boolean.parseBoolean(getPropertyValue("infrastructure.include.in.start.http.controllers"))) {
                for (Class<?> clazz : getConfig().getTypesAnnotatedWith(HttpEndpoint.class)) {
                    HttpEndpoint annotation = clazz.getAnnotation(HttpEndpoint.class);
                    Arrays.stream(annotation.value()).forEach(endpointUrl -> controllerMap.put(endpointUrl, clazz));
                    processedCalsesSet.add(clazz);
                }
            }
        }
        private void initNetworkDto(boolean shouldCheckLazyLoad) {
            if (shouldCheckLazyLoad && Boolean.parseBoolean(getPropertyValue("infrastructure.include.in.start.network.dto"))) {
                for (Class<?> clazz : getConfig().getTypesAnnotatedWith(NetworkDto.class)) {
                    NetworkDto annotation = clazz.getAnnotation(NetworkDto.class);
                    messageTypeToMessageClass.put(annotation.massageCode(), clazz);
                    massageClassToCode.put(clazz, annotation.massageCode());
                    processedCalsesSet.add(clazz);
                }
            }
        }
        private void initSingletonAnnotatedObjects(boolean shouldCheckLazyLoad) {
            if (shouldCheckLazyLoad && Boolean.parseBoolean(getPropertyValue("infrastructure.include.in.start.singleton"))) {
                for (Class<?> clazz : getConfig().getTypesAnnotatedWith(Singleton.class)) {
                    Singleton annotation = clazz.getAnnotation(Singleton.class);
                    if (!annotation.isLazy()) {
                        log.debug("created" + clazz.getName());
                        getObject(clazz);
```

```
            }
        }
    }
}

@Override
@SneakyThrows
public ClientWebSocketHandler createClientWebSocketConnection(String serverPath) {
    return new ClientWebSocketHandler(new URI("serverPath"));
}

public CurrencyInfo getCurrencyInfo(String langKey) {
    return currencies.get(langKey);
}

@Override
public <T> void addObject(Class<T> typeKey, Object object) {
    if (!objectsCash.containsKey(typeKey)) {
        synchronized (objectsCash) {
            if (!objectsCash.containsKey(typeKey)) {
                objectsCash.put(typeKey, object);
            }
        }
    }

}

public <T> T getObject(Class<T> typeKey) {
    if (objectsCash.containsKey(typeKey)) {
        return (T) objectsCash.get(typeKey);
    }
    synchronized (objectsCash) {
        if (objectsCash.containsKey(typeKey)) {
            return (T) objectsCash.get(typeKey);
        }
        Class<? extends T> implClass = typeKey;

        if (typeKey.isInterface()) {
            implClass = getConfig().getImplClass(typeKey);
```

```
        }
        T toReturn;
        try {
            toReturn = factory.createObject(implClass);
        } catch (InvocationTargetException | NoSuchMethodException | InstantiationException | IllegalAccessException e) {
            throw new ReflectionException(e);
        }
        putToObjectsCashIfSingleton(typeKey, implClass, toReturn);
        return toReturn;
    }
}

public MultipleMethodController getHttpCommand(String linkKey) {
    log.debug("");

    if (controllerMap.containsKey(linkKey)) {
        return (MultipleMethodController) getObject(controllerMap.get(linkKey));
    }
    synchronized (controllerMap) {
        if (controllerMap.containsKey(linkKey)) {
            return (MultipleMethodController) getObject(controllerMap.get(linkKey));
        }
        Set<Class<?>> typesToLookFor = getConfig().getTypesAnnotatedWith(HttpEndpoint.class).stream()
                .filter(type -> !processedCalsesSet.contains(type))
                .collect(Collectors.toSet());
        for (Class<?> clazz : typesToLookFor) {
            HttpEndpoint annotation = clazz.getAnnotation(HttpEndpoint.class);
            Arrays.stream(annotation.value()).forEach(endpointUrl -> controllerMap.put(endpointUrl, clazz));
            processedCalsesSet.add(clazz);
            for (String i : annotation.value()) {
                if (i.equals(linkKey)) {
                    MultipleMethodController toReturn = (MultipleMethodController) getObject(clazz);
                    return toReturn;
                }
            }
        }
    }
    return (MultipleMethodController) getObject(defaultEndpoint);
}
```

```
    @Override
    public RestUrlCommandProcessorInfo getRestCommand(String requestUrl, String requestMethod) {
        log.debug("");

        KeyInfoRestRequest keyInfoRestRequest = getKeyInfoRestRequest(requestUrl);
        RestProcessorCashInfo restProcessorCashInfo = urlMatchPatternToCommandProcessorInfo.get(keyInfoRestRequest.resourceKey);
        if (restProcessorCashInfo != null) {
            return RestUrlCommandProcessorInfo.builder()
                    .commandProcessor(getObject(restProcessorCashInfo.getCommandProcessor()))
                    .processorsMethod(RequestUtilService.retrieveMethodForProcessRequest(keyInfoRestRequest.lustUrlStep, requestMethod,
restProcessorCashInfo,
                                                                                          keyInfoRestRequest.resourceKey))
                    .restUrlVariableInfos(restProcessorCashInfo.getRestUrlVariableInfos())
                    .build();
        }

        synchronized (urlMatchPatternToCommandProcessorInfo) {
            restProcessorCashInfo = urlMatchPatternToCommandProcessorInfo.get(keyInfoRestRequest.resourceKey);
            if (restProcessorCashInfo != null) {
                return RestUrlCommandProcessorInfo.builder()
                        .commandProcessor(getObject(restProcessorCashInfo.getCommandProcessor()))
                        .processorsMethod(
                                RequestUtilService.retrieveMethodForProcessRequest(keyInfoRestRequest.lustUrlStep, requestMethod,
restProcessorCashInfo,
                                                                                   keyInfoRestRequest.resourceKey))
                        .restUrlVariableInfos(restProcessorCashInfo.getRestUrlVariableInfos())
                        .build();
            }
        }
        return getNotCashedYetRestUrlCommandProcessorInfo(requestUrl, requestMethod);
    }

    private RestUrlCommandProcessorInfo getNotCashedYetRestUrlCommandProcessorInfo(String requestUrl, String requestMethod) {
        Set<Class<?>> typesToLookFor = getConfig().getTypesAnnotatedWith(RestEndpoint.class).stream()
                .filter(type -> !processedCalsesSet.contains(type))
                .collect(Collectors.toSet());
        for (Class<?> clazz : typesToLookFor) {
            RestEndpoint annotation = clazz.getAnnotation(RestEndpoint.class);
            for (String resourceFromAnnotation : annotation.resource()) {
```

```
                String[] urlSteps = resourceFromAnnotation.split(URL_STEP_SPLIT_REGEX);
                urlSteps = Arrays.copyOfRange(urlSteps, 1, urlSteps.length);
                String restCommandKey = resourceFromAnnotation.replaceAll(REST_ENDPOINT_VARIABLE_PLACEHOLDER_REGEX,
                                                        REPLACEMENT_REGEX_FOR_REST_ENDPOINT_VARIABLE_PLACEHOLDER)
                        .replaceAll(REST_ENDPOINT_LUST_VARIABLE_PLACEHOLDER_REGEX,
REPLACEMENT_REGEX_FOR_REST_ENDPOINT_LUST_VARIABLE_PLACEHOLDER);
                cashRestCommandProcessor(clazz, restCommandKey, urlSteps);
                typesToLookFor.add(clazz);
                KeyInfoRestRequest keyInfoRestRequest = getKeyInfoRestRequest(requestUrl);
                if (keyInfoRestRequest.resourceKey.equals(restCommandKey)) {

                  return RestUrlCommandProcessorInfo.builder()
                        .commandProcessor(getObject(clazz))
                        .restUrlVariableInfos(getRestUrlVariableInfos(urlSteps))
                        .processorsMethod(
                              getConfig().getMethodAnnotatedWith(
                                    clazz,
                                    RequestUtilService.getRestMethodAnnotation(keyInfoRestRequest.lustUrlStep, requestMethod, restCommandKey))
                        ).build();
                  }
                }
              }
              return RestUrlCommandProcessorInfo.builder().commandProcessor(new Rest404Controller())
                    .processorsMethod(getConfig().getMethodAnnotatedWith(
                          Rest404Controller.class, RestGetAll.class))
                    .build();
            }

            private KeyInfoRestRequest getKeyInfoRestRequest(String requestUrl) {
              String[] split = requestUrl.split(URL_STEP_SPLIT_REGEX);
              String restUrlWithoutVariables = "";
              for (int i = 1; i < split.length; i += 2) {
                restUrlWithoutVariables += URL_STEP_DELIMITER + split[i];
              }
              restUrlWithoutVariables += URL_STEP_DELIMITER;
              return new KeyInfoRestRequest(split[split.length - 1] + URL_STEP_DELIMITER, restUrlWithoutVariables);
            }

            private static class KeyInfoRestRequest {
```

```
    public final String lustUrlStep;
    public final String resourceKey;

    public KeyInfoRestRequest(String lustUrlStep, String resourceKey) {
        this.lustUrlStep = lustUrlStep;
        this.resourceKey = resourceKey;
    }
}

private String removeLustIdParameter(String resourceFromAnnotation) {
    String resourceWithoutLustIdParameter = resourceFromAnnotation;//отбрасываем последний парметр в урле
    int lastIndex = resourceWithoutLustIdParameter.lastIndexOf(URL_STEP_DELIMITER);
    if (lastIndex != -1) {
        resourceWithoutLustIdParameter = resourceFromAnnotation.substring(0, lastIndex);
    }
    return resourceWithoutLustIdParameter;
}

private List<RestUrlVariableInfo> getRestUrlVariableInfos(String[] urlSteps) {
    List<RestUrlVariableInfo> restUrlVariableInfos = new ArrayList<>();
    for (int j = 0; j < urlSteps.length; j++) {
        if (urlSteps[j].matches(REST_URL_VARIABLE_REGEX)) {
            restUrlVariableInfos.add(new RestUrlVariableInfo(j, urlSteps[j].substring(1, urlSteps[j].length() - 1)));
        }
    }
    return restUrlVariableInfos;
}

private void cashRestCommandProcessor(Class<?> clazz, String resourceUrlKey, String[] urlSteps) {
    RestProcessorCashInfo restUrlCommandProcessorInfo = new RestProcessorCashInfo();
    restUrlCommandProcessorInfo.setCommandProcessor(clazz);
    restUrlCommandProcessorInfo.setGetByIdMethod(getConfig().getMethodAnnotatedWith(clazz, RestGetById.class));
    restUrlCommandProcessorInfo.setGetAllMethod(getConfig().getMethodAnnotatedWith(clazz, RestGetAll.class));
    restUrlCommandProcessorInfo.setUpdateMethod(getConfig().getMethodAnnotatedWith(clazz, RestUpdate.class));
    restUrlCommandProcessorInfo.setPutMethod(getConfig().getMethodAnnotatedWith(clazz, RestPut.class));
    restUrlCommandProcessorInfo.setDeleteMethod(getConfig().getMethodAnnotatedWith(clazz, RestDelete.class));
    restUrlCommandProcessorInfo.setRestUrlVariableInfos(getRestUrlVariableInfos(urlSteps));
    urlMatchPatternToCommandProcessorInfo.put(resourceUrlKey, restUrlCommandProcessorInfo);
}
```

```
@Override
public TcpController getTcpCommandController(String requestMessageType) {
    if (messageTypeTcpCommandControllerMap.containsKey(requestMessageType)) {
        return (TcpController) getObject(messageTypeTcpCommandControllerMap.get(requestMessageType));
    }
    synchronized (messageTypeTcpCommandControllerMap) {
        if (messageTypeTcpCommandControllerMap.containsKey(requestMessageType)) {
            return (TcpController) getObject(messageTypeTcpCommandControllerMap.get(requestMessageType));
        }
        Set<Class<?>> typesToLookFor = getConfig().getSubTypesOf(AbstractTcpController.class).stream()
                .filter(type -> !processedCalsesSet.contains(type))
                .collect(Collectors.toSet());
        for (Class<?> clazz : typesToLookFor) {
            String massageCode = cashTcpControllerData(clazz);
            if (requestMessageType.equals(massageCode)) {
                TcpController toReturn = (TcpController) getObject(clazz);
                return toReturn;
            }
        }
    }
    return getObject(Error404DefaultController.class);
}

private String cashTcpControllerData(Class<?> clazz) {
    ParameterizedType genericSuperclass = (ParameterizedType) clazz.getGenericSuperclass();
    Class<?> actualTypeArgument = (Class<?>) genericSuperclass.getActualTypeArguments()[0];
    String massageCode = actualTypeArgument.getAnnotation(NetworkDto.class).massageCode();
    messageTypeTcpCommandControllerMap.put(massageCode, clazz);
    putTcpMessageToCash(actualTypeArgument, massageCode);
    processedCalsesSet.add(clazz);
    return massageCode;
}

private void putTcpMessageToCash(Class<?> actualTypeArgument, String massageCode) {
    messageTypeToMessageClass.put(massageCode, actualTypeArgument);
    massageClassToCode.put(actualTypeArgument, massageCode);
    processedCalsesSet.add(actualTypeArgument);
}
```

```
@Override
public Class<?> getMessageTypeByCode(String messageCode) {
    Class<?> messageClass = messageTypeToMessageClass.get(messageCode);
    if (messageClass == null) {
        synchronized (messageTypeToMessageClass) {
            messageClass = messageTypeToMessageClass.get(messageCode);
            if (messageClass == null) {
                Set<Class<?>> typesToLookFor = getConfig().getTypesAnnotatedWith(NetworkDto.class).stream()
                        .filter(type -> !processedCalsesSet.contains(type))
                        .collect(Collectors.toSet());
                for (Class<?> clazz : typesToLookFor) {
                    String massageCodeFromAnnotation = clazz.getAnnotation(NetworkDto.class).massageCode();
                    putTcpMessageToCash(clazz, massageCodeFromAnnotation);
                    if (messageCode.equals(massageCodeFromAnnotation)) {
                        return clazz;
                    }
                }
            } else {
                return Error404Dto.class;
            }
        }
    }
    return messageClass;
}

@Override
public String getMessageCodeByType(Class<?> messageType) {
    String messageCode = massageClassToCode.get(messageType);
    if (messageCode == null) {
        synchronized (massageClassToCode) {
            messageCode = massageClassToCode.get(messageType);
            if (messageCode == null) {
                messageCode = messageType.getAnnotation(NetworkDto.class).massageCode();
                putTcpMessageToCash(messageType, messageCode);
            }
        }
    }
    return messageCode;
```

```
    }
    public void setFactory(ObjectFactory factory) {
        this.factory = factory;
    }
    public Config getConfig() {
        if (this.config == null) {
            synchronized (this) {
                if (this.config == null) {
                    this.config = new JavaConfig(getPropertyValue("infrastructure.package.to.annotation.scan"));
                }
            }
        }
        return this.config;
    }
    private <T> void putToObjectsCashIfSingleton(Class<T> type, Class<? extends T> implClass, T instance) {
        if (implClass.isAnnotationPresent(Singleton.class)) {
            objectsCash.put(type, instance);
        }
    }
}
```

Код файлу ObjectFactoryImpl.java що відповідає за створення об’єктів.

```
public class ObjectFactoryImpl implements ObjectFactory {
    private static final Logger log = LogManager.getLogger(ObjectFactoryImpl.class);

    private final ApplicationContext context;
    private final List<ObjectConfigurator> configurators = new ArrayList<>();
    private final List<ProxyConfigurator> proxyConfigurators = new ArrayList<>();

    public ObjectFactoryImpl(ApplicationContext context) {
        log.debug("");

        this.context = context;

        try {
            for (Class<? extends ObjectConfigurator> aClass : context.getConfig().getSubTypesOf(ObjectConfigurator.class)) {
                configurators.add(aClass.getDeclaredConstructor().newInstance());
            }
```

```
            for (Class<? extends ProxyConfigurator> aClass : context.getConfig().getSubTypesOf(ProxyConfigurator.class)) {
                proxyConfigurators.add(aClass.getDeclaredConstructor().newInstance());
            }
        } catch (InstantiationException | IllegalAccessException | InvocationTargetException | NoSuchMethodException ignored) {
        }
    }

    @Override
    public <T> T createObject(Class<T> implClassKey) throws InvocationTargetException, NoSuchMethodException, InstantiationException, IllegalAccessException {
        log.debug("");

        T toReturn = create(implClassKey);
        configure(toReturn);
        invokeInit(implClassKey, toReturn);
        toReturn = wrapWithProxyIfNeeded(implClassKey, toReturn);
        return toReturn;
    }

    private <T> T create(Class<T> implClass) throws InstantiationException, IllegalAccessException, InvocationTargetException, NoSuchMethodException {
        return implClass.getDeclaredConstructor().newInstance();
    }

    private <T> void configure(T instance) {
        Class currentClass = instance.getClass();
        while (currentClass.isAnnotationPresent(NeedConfig.class)) {
            Class finalCurrentClass = currentClass;
            configurators.forEach(objectConfigurator -> objectConfigurator.configure(instance, finalCurrentClass, context));
            currentClass = currentClass.getSuperclass();
        }
    }

    private <T> void invokeInit(Class<T> implClass, T instance) {
        Arrays.stream(implClass.getMethods()).filter(method -> method.isAnnotationPresent(PostConstruct.class))
                .forEach(method -> {
                    try {
                        method.invoke(instance);
                    } catch (IllegalAccessException | InvocationTargetException e) {
                        throw new ReflectionException();
```

```
                }
            });
    }

    private <T> T wrapWithProxyIfNeeded(Class<T> implClass, T instance) {
        for (ProxyConfigurator proxyConfigurator : proxyConfigurators) {
            instance = (T) proxyConfigurator.replaceWithProxyIfNeeded(instance, implClass, context);
        }
        return instance;
    }
}
```

Код файлу FrontController.java

```
public abstract class FrontController extends GenericServlet {

    protected ApplicationContext context;
    protected static final String JSON_RESPONSE_KEY = "infrastructure.web.json.response.prefix";
    private static final String INFRASTRUCTURE_APPLICATION_URL_REDIRECT_PREFIX = "infrastructure.web.url.redirect.prefix";

    @Override
    public void init() {
        try {
            Class.forName("com.mysql.cj.jdbc.Driver");
            System.out.println("MySQL JDBC Driver loaded successfully");
        } catch (ClassNotFoundException e) {
            System.out.println("Failed to load MySQL JDBC Driver");
            e.printStackTrace();
        }
        final ServletContext servletContext = getServletContext();
        servletContext.setAttribute(LOGGED_USER_NAMES, new ConcurrentHashMap<String, HttpSession>());
        context = ApplicationContextImpl.getContext();
    }


    protected String getMethodCode(HttpServletRequest request) {
        if (request.getContentType() != null && request.getContentType().equals("application/x-www-form-urlencoded") &&
request.getParameter("_method") != null) {
            switch (request.getParameter("_method")) {
```

```
                case "GET":
                    return "GET";
                case "PUT":
                    return "PUT";
                case "DELETE":
                    return "DELETE";
                case "POST":
                default:
                    return "POST";
            }
        }
        return request.getMethod();
    }

    protected void passOver(HttpServletRequest request, HttpServletResponse response, String page) throws IOException, ServletException {
        if (page.contains(context.getPropertyValue(INFRASTRUCTURE_APPLICATION_URL_REDIRECT_PREFIX))) {
            response.sendRedirect(page.replace(context.getPropertyValue(INFRASTRUCTURE_APPLICATION_URL_REDIRECT_PREFIX),
"""));
        } else if (page.startsWith(context.getPropertyValue(JSON_RESPONSE_KEY))) {
            response.setContentType("application/json");
            response.setCharacterEncoding("UTF-8");
            PrintWriter out = response.getWriter();
            out.print(page.replaceFirst(context.getPropertyValue(JSON_RESPONSE_KEY), """));
            out.flush();
        } else {
            request.getRequestDispatcher(page).forward(request, response); //стандартная работа по пробросу на джспеху
        }
    }
}
```

Код файлу RestFrontController.java що виступає фронт контролером для запитів за архітектурою REST.

```
public class RestFrontController extends FrontController {

    private static final RequestUtilService REST_URL_UTIL_SERVICE = new RequestUtilService();
    private static final String INFRASTRUCTURE_APPLICATION_REST_URL_PREFIX = "infrastructure.application.rest.url.prefix";
    private static final String INFRASTRUCTURE_APPLICATION_URL_BASE_PATH = "infrastructure.application.url.base.path";

    @Override
    public void service(ServletRequest servletRequest, ServletResponse servletResponse) throws ServletException, IOException {
```

```
        HttpServletRequest request = (HttpServletRequest) servletRequest;
        HttpServletResponse response = (HttpServletResponse) servletResponse;
        String logicUrlPath = request.getRequestURI().replaceFirst(
            ".*" + context.getPropertyValue(INFRASTRUCTURE_APPLICATION_URL_BASE_PATH), "");
        passOver(request, response, processRestRequest(logicUrlPath, request, response));
    }

    private String processRestRequest(String logicUrlPath, HttpServletRequest request,
                                      HttpServletResponse response) {
        logicUrlPath = logicUrlPath.replaceFirst(context.getPropertyValue(INFRASTRUCTURE_APPLICATION_REST_URL_PREFIX), "");
        RestUrlCommandProcessorInfo restCommandProcessorInfo = context.getRestCommand(logicUrlPath, getMethodCode(request));

        Object[] parametersToPassInInvocation = populateMethodParametersValues(logicUrlPath, request, response,
restCommandProcessorInfo);

        try {
            Object invoke = restCommandProcessorInfo.getProcessorsMethod().invoke(restCommandProcessorInfo.getCommandProcessor(),
parametersToPassInInvocation);
            return context.getPropertyValue(JSON_RESPONSE_KEY) + new Gson().toJson(invoke);
        } catch (IllegalAccessException | InvocationTargetException e) {
            e.printStackTrace();
            return "null";
        }
    }

    private static Object[] populateMethodParametersValues(String logicUrlPath, HttpServletRequest request, HttpServletResponse response,
RestUrlCommandProcessorInfo restCommandProcessorInfo) {
        Object[] parametersToPassInInvocation = REST_URL_UTIL_SERVICE.createParametersArray(restCommandProcessorInfo);
        REST_URL_UTIL_SERVICE.populateParametersFromRequestUrl(logicUrlPath, restCommandProcessorInfo,
parametersToPassInInvocation);
        REST_URL_UTIL_SERVICE.populateHttpRequest(request, restCommandProcessorInfo, parametersToPassInInvocation);
        REST_URL_UTIL_SERVICE.populateHttpResponse(response, restCommandProcessorInfo, parametersToPassInInvocation);
        REST_URL_UTIL_SERVICE.populateRequestBody(request, restCommandProcessorInfo, parametersToPassInInvocation);
        return parametersToPassInInvocation;
    }
}
```

Код файлу PureHttpFrontController.java що виступає фронт контролером для запитів за HTTP протоколом.

```
public class PureHttpFrontController extends FrontController {

    private static final Logger log = LogManager.getLogger(PureHttpFrontController.class);
```

```
    private static final String INFRASTRUCTURE_APPLICATION_URL_BASE_PATH = "infrastructure.application.url.base.path";

    @Override
    public void service(ServletRequest servletRequest, ServletResponse servletResponse) throws ServletException, IOException {
      HttpServletRequest request = (HttpServletRequest) servletRequest;
      HttpServletResponse response = (HttpServletResponse) servletResponse;
      processGeneralHttpRequest(response, request);
    }

    private void processGeneralHttpRequest(HttpServletResponse servletResponse, HttpServletRequest request) throws IOException,
ServletException {
      switch (getMethodCode(request)) {
        case "GET":
          doGet(request, servletResponse);
          break;
        case "POST":
          doPost(request, servletResponse);
          break;
        case "PUT":
          doPut(request, servletResponse);
          break;
        case "DELETE":
          doDelete(request, servletResponse);
          break;
      }
    }

    private void doGet(HttpServletRequest request,
                HttpServletResponse response)
        throws IOException, ServletException {
      log.debug("servlet called with request - " + request.getRequestURI());

      passOver(request, response, getMultipleMethodCommand(request).doGet(request));
    }

    private void doPost(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException {
      log.debug("servlet called with request - " + request.getRequestURI());
```

```
        passOver(request, response, getMultipleMethodCommand(request).doPost(request));
    }

    private void doPut(HttpServletRequest request, HttpServletResponse response)
            throws IOException, ServletException {
        log.debug("servlet called with request - " + request.getRequestURI());

        passOver(request, response, getMultipleMethodCommand(request).doPut(request));
    }

    private void doDelete(HttpServletRequest request, HttpServletResponse response)
            throws IOException, ServletException {
        log.debug("servlet called with request - " + request.getRequestURI());

        passOver(request, response, getMultipleMethodCommand(request).doDelete(request));
    }

    private MultipleMethodController getMultipleMethodCommand(HttpServletRequest request) {
        return context
                .getHttpCommand(request.getRequestURI().replaceFirst
                        (".*" + context.getPropertyValue(INFRASTRUCTURE_APPLICATION_URL_BASE_PATH), ""));
    }
}
```

sКод файлу TransactionProxyConfigurator.java що відповідає за додання транзакцій до методів.

```
public class TransactionProxyConfigurator implements ProxyConfigurator {
    private static final Logger log = LogManager.getLogger(TransactionProxyConfigurator.class);

    @Override
    public Object replaceWithProxyIfNeeded(Object instance, Class instanceType, ApplicationContext context) {
        for (Method method : instanceType.getMethods()) {
            if (method.isAnnotationPresent(Transaction.class)) {
                if (instanceType.getInterfaces().length == 0) {
                    return Enhancer.create(instanceType, new net.sf.cglib.proxy.InvocationHandler() {
                        @Override
                        public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
                            return getInvocationHandlerLogic(method, args, instance, context);
                        } });
```

```
            }
            return Proxy.newProxyInstance(instanceType.getClassLoader(), instanceType.getInterfaces(),
                    (proxy, met, args) -> getInvocationHandlerLogic(met, args, instance, context));
        }
    }
    return instance;
  }

  private Object getInvocationHandlerLogic(Method method, Object[] args, Object t, ApplicationContext context) throws Throwable {
    log.debug("getInvocationHandlerLogic");
    try {
      if (t.getClass().getMethod(method.getName(), method.getParameterTypes()).isAnnotationPresent(Transaction.class)) {
        return doTransactionMethodCall(method, args, t, context.getObject(ConnectionManager.class));
      }
    } catch (NoSuchMethodException | SQLException | InstantiationException ex) {
      log.debug(ex);
    }
    try {
      return method.invoke(t, args);
    } catch (Throwable e) {
      throw e;
    }
  }

  private Object doTransactionMethodCall(Method method, Object[] args, Object t, ConnectionManager connectionManager)
      throws Throwable {
    try {
      connectionManager.startTransaction();
      Object result = method.invoke(t, args);
      connectionManager.commit();
      connectionManager.close();
      return result;
    } catch (Throwable e) {
      connectionManager.rollBack();
      connectionManager.close();
      throw e;
    }
  }
}
```

Код файлу InjectStringPropertyAnnotationObjectConfigurator.java що відповідає за впровадження текстових значень в атрибути класу.

```
public class InjectStringPropertyAnnotationObjectConfigurator implements ObjectConfigurator {
    private static final Logger log = LogManager.getLogger(InjectStringPropertyAnnotationObjectConfigurator.class);

    @Override
    public void configure(Object instance, Class instanceType, ApplicationContext context) {
        for (Field field : instanceType.getDeclaredFields()) {
            InjectStringProperty annotation = field.getAnnotation(InjectStringProperty.class);
            if (annotation != null) {
                String value = annotation.value().isEmpty() ?
                        context.getPropertyValue(field.getName()) :
                        context.getPropertyValue(annotation.value());
                field.setAccessible(true);
                try {
                    field.set(instance, value);
                } catch (IllegalAccessException e) {
                    throw new ReflectionException("impossible we made accessible true");
                }
            }
        }
    }
}
```

Код файлу InjectByTypeAnnotationObjectConfigurator.java що відповідає за впровадження залежностей за типом.

```
public class InjectByTypeAnnotationObjectConfigurator implements ObjectConfigurator {
    @Override
    public void configure(Object instance, Class instanceType, ApplicationContext context) {

        for (Field field : instanceType.getDeclaredFields()) {
            if (field.isAnnotationPresent(InjectByType.class)) {
                field.setAccessible(true);
                Object object = context.getObject(field.getType());
                try {
                    field.set(instance, object);
                } catch (IllegalAccessException e) {
                    throw new ReflectionException("impossible we made accessible true");
                }
```

```
            }
        }

    }
}
```

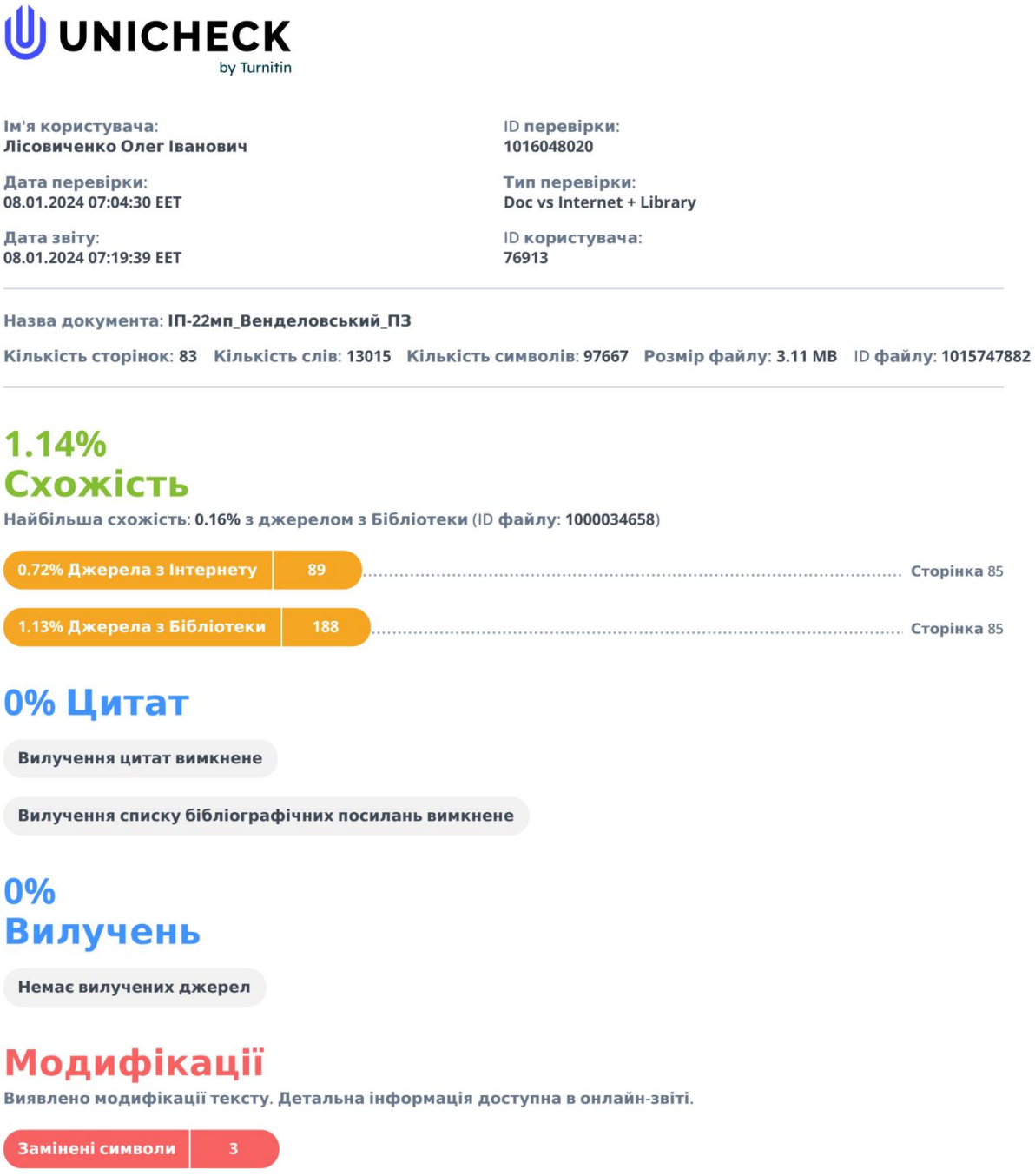
Код файлу WebSocketFrontController.java що є фронт контролером для запитів за протоколом Web Socket.

```
@ServerEndpoint(value = "/socket", decoders = MassageDecoder.class, encoders = MassageEncoder.class)
public class WebSocketFrontController implements ConnectionNotificationSubscriber {
    private ApplicationContext applicationContext;
    private IdentityCommunicationSessionService identityCommunicationSessionService;
    private Session session;

    public WebSocketFrontController() {
        applicationContext = ApplicationContextImpl.getContext();
        applicationContext.addObject(WebSocketFrontController.class, this);
        identityCommunicationSessionService = applicationContext.getObject(IdentityCommunicationSessionService.class);
    }



    @OnOpen
    public void onOpen(
            Session session) throws IOException {
        this.session = session;
        identityCommunicationSessionService.registerNewEmptySession(session.getId(), this);
    }

    @OnMessage
    public Object onMessage(Session session, SocketReceivedMessage request) {
        String requestMessageCode = request.getMessageCode();
        if (!requestMessageCode.equals("LOGIN") && !identityCommunicationSessionService.isUserLongedIn(session)) {
            return new ErrorDto("You are not logged in so your request was declined");
        }

        Class messageTypeByCode = applicationContext.getMessageTypeByCode(requestMessageCode);
        final TcpController tcpController = applicationContext.getTcpCommandController(requestMessageCode);
        return tcpController.service(messageTypeByCode.cast(convertJsonMassageToDto(request)), session);
    }
```

```
    @OnClose
    public void onClose(Session session) {
        identityCommunicationSessionService.closeSession(session);
    }

    @OnError
    public void onError(Session session, Throwable throwable) {
        identityCommunicationSessionService.closeSession(session);
    }

    @SneakyThrows
    @Override
    public void processMessage(Object request) {
        String messageCode = applicationContext.getMessageCodeByType(request.getClass());
        final TcpController tcpController = applicationContext.getTcpCommandController(messageCode);
        final Object response = tcpController.service(request, session);
        session.getBasicRemote().sendObject(response);
    }

    @SneakyThrows
    private Object convertJsonMassageToDto(SocketReceivedMessage socketReceivedMessage) {
        final Class messageTypeByCode = applicationContext.getMessageTypeByCode(socketReceivedMessage.getMessageCode());
        return new Gson().fromJson(socketReceivedMessage.getJsonMessageData(), messageTypeByCode);
    }
}
```

ДОДАТОК Б

Результати перевірки роботи на співпадіння



UNICHECK
by Turnitin

Ім'я користувача:
Лісовиченко Олег Іванович

ID перевірки:
1016048020

Дата перевірки:
08.01.2024 07:04:30 EET

Тип перевірки:
Doc vs Internet + Library

Дата звіту:
08.01.2024 07:19:39 EET

ID користувача:
76913

Назва документа: ІП-22мп_Венделовський_ПЗ

Кількість сторінок: 83 **Кількість слів: 13015** **Кількість символів: 97667** **Розмір файлу: 3.11 MB** **ID файлу: 1015747882**

1.14%
Схожість

Найбільша схожість: 0.16% з джерелом з Бібліотеки (ID файлу: 1000034658)

0.72% Джерела з Інтернету **89** ..... **Сторінка 85**

1.13% Джерела з Бібліотеки **188** ..... **Сторінка 85**

0% Цитат

Вилучення цитат вимкнене

Вилучення списку бібліографічних посилань вимкнене

0%
Вилучень

Немає вилучених джерел

Модифікації

Виявлено модифікації тексту. Детальна інформація доступна в онлайн-звіті.

Замінені символи **3**