

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

Інститут прикладного системного аналізу

Кафедра системного проектування

До захисту допущено:

Завідувач кафедри

_____ Анатолій ПЕТРЕНКО

«__» _____ 20__ р.

Дипломна робота

на здобуття ступеня бакалавра

**за освітньо-професійною програмою «Інтелектуальні сервіс-орієнтовані
розподілені обчислювання»**

спеціальності 122 «Комп'ютерні науки»

**на тему: «Інструментарій для опису екосистеми мікросервісів з
використанням графової бази даних»**

Виконав (-ла):

студент (-ка) IV курсу, групи ДА-71

Стефура Олег Ярославович _____

Керівник:

к.т.н., доцент кафедри СП

Булах Богдан Вікторович _____

Консультант з економічного розділу:

к.т.н. економічних наук, доцент,

Рощина Надія Василівна _____

Рецензент:

в.о. зав. каф. ММСА, к.т.н., доц

Тимощук Оксана Леонідівна _____

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

Студент (-ка) _____

Київ – 2021 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Інститут прикладного системного аналізу
Кафедра системного проектування

Рівень вищої освіти – перший (бакалаврський)

Спеціальність – 122 «Комп’ютерні науки»

Освітньо-професійна програма «Інтелектуальні сервіс-орієнтовані розподілені обчислювання»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Анатолій ПЕТРЕНКО

« ____ » _____ 20__ р.

ЗАВДАННЯ
на дипломну роботу студенту
Стефури Олега Ярославовича

1. Тема роботи **«Інструментарій для опису екосистеми мікросервісів з використанням графової бази даних»**, керівник роботи Булах Богдан Вікторович, к.т.н., доцент, затверджені наказом по університету від « ____ » _____ 20__ р. № _____
2. Термін подання студентом роботи _____
3. Вихідні дані до роботи
4. Зміст роботи
5. Перелік ілюстративного матеріалу (із зазначенням плакатів, презентацій тощо)

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Економічний	Рощина Надія Василівна, канд.економ.наук, доцент		

7. Дата видачі завдання _____

Календарний план

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітка
1.	Затвердження теми БДР	15.04.2021 – 21.04.2021	Виконано
2.	Ознайомлений зі структурою БДР згідно з Положенням про державну атестацію студентів НТУУ «КПІ»	15.04.2021 – 28.04.2021	Виконано
3.	Ознайомлений з ДСТУ 3008-95 та стандарти ЄСПД	21.04.2021-28.04.2021	Виконано
4.	Проведення дослідження за темою БДР під керівництвом керівника	28.04.2021 – 05.05.2021	Виконано
5.	Завершення роботи над першим варіантом частини БДР	05.05.2021 – 19.05.2021	Виконано
6.	Проведення роботи над експериментальною частиною БДР	05.05.2021 – 19.05.2021	Виконано
7.	Проведення роботи над програмним продуктом	19.05.2021 – 26.05.2021	Виконано
8.	Оформлення БДР та аналіз отриманих результатів	19.05.2021 – 31.05.2021	Виконано

Студент

Олег СТЕФУРА

Керівник

Богдан БУЛАХ

РЕФЕРАТ

Тема: «Інструментарій для опису екосистеми мікросервісів з використанням графової бази даних»

Дипломна робота 121с., 35 рис., 7 табл., 2 додатки, 16 джерел.

МІКРОСЕРВІСНІ ТА СЕРВІС-ОРІЄНТОВАНІ СИСТЕМИ, ГРАФОВІ
БАЗИ ДАНИХ, СИСТЕМНИЙ МОНІТОРИНГ

Об'єкт дослідження – опис та моніторинг мікросервісних та сервіс-орієнтованих систем.

Предмет дослідження – досвід застосування графових баз даних для опису екосистеми мікросервісів.

Мета роботи – проаналізувати предмет дослідження, реалізувати програмний продукт для автоматичного опису мікросервісних архітектур із використанням графової бази даних.

Методи дослідження – методи проектування систем, методи проектування додатків, методи теорії графів.

Актуальність – створення додатку для автоматичного опису мікросервісних архітектур із використанням графових баз даних дозволить отримати додатковий інструмент моніторингу, що дозволить виявляти підозрілу поведінку в таких системах та вчасно запобігати провалам у роботі.

Результат роботи – було створено та протестовано додаток, що автоматично описує мікросервісні системи, побудовані на технологіях екосистеми Java.

Шляхи подальшого розвитку предмету – деталізація представлення системи, що досліджується, розширення спектру технологій, що може бути підданий дослідженню.

ABSTRACT

The theme: "Microservice ecosystem description tool using graph database"

Bachelor thesis: 121p., 35 fig., 7 tabl., 2 append., 14 sources.

MICROSERVICE AND SERVICE-ORIENTED SYSTEMS, GRAPH DATABASES, SYSTEM MONITORING

The object of research is the description and monitoring of microservice and service-oriented systems.

The subject of research is the experience of using graph databases to describe the ecosystem of microservices.

The purpose of the work is to analyze the subject of research, to implement a software product for automatic description of microservice architectures using a graph database.

Research methods - methods of system design, methods of application design, methods of graph theory.

Relevance - creating an application for automatic description of microservice architectures using graph databases will provide an additional monitoring tool that will detect suspicious behavior in such systems and prevent failures in a timely manner.

The result - an application was created and tested that automatically describes microservice systems based on Java ecosystem technologies.

Ways of further development of the subject - detailing the presentation of the system under investigation, expanding the range of technologies that can be studied.

ЗМІСТ

ВСТУП	8
1. РОЗДІЛ 1: ОГЛЯД СУЧАСНИХ МЕТОДІВ ТА ІНСТРУМЕНТІВ АНАЛІЗУ СЕРВІС-ОРІЄНТОВАНИХ ТА МІКРОСЕРВІСНИХ АРХІТЕКТУРНИХ РІШЕНЬ	10
1.1 Проблематика моніторингу великих систем. Маркери моніторингу	10
1.1.1 Метрики програми	11
1.1.2 Метрики платформи	14
1.1.3 Внутрішньосистемні події	15
1.2 Принципи аналізу мікросервісів.....	16
1.2.1 Контроль контейнерів та їх вмісту	16
1.2.2 Експлуатація систем оркестрації для контролю ефективності сервісів	18
1.2.3 Ефективна експлуатація еластичних та багатопозиційних сервісів	19
1.2.4 Моніторинг API.....	20
1.2.5 Синхронні організаційні зміни при впровадженні систем моніторингу	21
1.3 Огляд деяких популярних сучасних інструментів	21
1.3.1 Raygun APM.....	22
1.3.2 Stackify Retrace	25
1.3.3 Prometheus.....	28
1.3.4 AppDynamics.....	29
1.4 Висновки	31
РОЗДІЛ 2. ОГЛЯД ІДЕЇ ЗАСТОСУВАННЯ ТЕОРІЇ ГРАФІВ ДЛЯ ОПИСУ ТА АНАЛІЗУ РОЗПОДІЛЕНИХ ТА МІКРОСЕРВІСНИХ РІШЕНЬ.....	32
2.1 Графові бази даних як інструмент для опису та аналізу формалізованих знань.....	32
2.2 Поняття графу знань	34
2.2.1 Огляд ідеї графу знань.....	34
2.2.2 Приклади застосування графів знань.....	37
2.3 Сучасні технології графових баз даних	38
2.3.1 Поняття графових баз даних	38
2.3.2 Переваги графових баз даних	39
2.3.3 Відомі приклади застосування	41
2.4 Приклад опису мікросервісної архітектури на основі графової моделі	43
2.5 Висновки	49
3. РОЗДІЛ 3: ПРАКТИЧНА РЕАЛІЗАЦІЯ ДОДАТКУ ДЛЯ ОПИСУ ЕКОСИСТЕМИ МІКРОСЕРВІСІВ ІЗ ВИКОРИСТАННЯМ ГРАФОВОЇ БАЗИ ДАНИХ.....	51
3.1 Вимоги до тестового додатку для опису екосистеми мікросервісів.....	51
3.1.1 Формування зрозумілої картини системи	51
3.1.2 Можливість ефективного оперування отриманими даними	52

3.1.3 Масштабованість.....	53
3.1.4 Простота використання	53
3.1.5 Продуктивність	54
3.2 Підбір інструментів для реалізації	54
3.2.1 Графова база даних neo4j	55
3.2.2 Мова запитів Cypher	56
3.2.3 Засоби візуалізації: Neo4j embedded visualizing tool/Neo4j Desktop	57
3.2.5 Бібліотека Object ASM	59
3.3 Реалізація програми	60
3.3.1 Модель програми	60
3.3.2 Ключові фрагменти коду програми	61
3.4 Приклади роботи програми	70
3.6 Оперування отриманим графом	75
3.6.1 Пошук всіх кінцевих точок системи	75
3.6.2 Виявлення місця декларації кінцевих точок	76
3.6.3 Виявлення залежностей між мікросервісами	79
3.6.4 Виявлення завантажених точок	80
3.7 Висновки	80
4. РОЗДІЛ: ФУНКЦІОНАЛЬНО-ВАРТІСНИЙ АНАЛІЗ ПРОГРАМНОГО ПРОДУКТУ	82
4.1 Постановка задачі	82
4.2 Визначення функцій потенційного продукту	83
4.3 Система параметрів програмного продукту.....	87
4.4 Експертна оцінка параметрів	89
4.5 Рівень якості варіантів реалізації функцій	93
4.6 Економічний аналіз варіантів розробки ПП.....	94
4.7 Вибір кращого варіанту ПП техніко-економічного рівня	100
4.8 Висновки	101
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....	102
ДОДАТОК А: КОД ПРОГРАМИ	104

ВСТУП

Моніторинг залишається важливою частиною управління будь-якою ІТ-системою, і особливо – системами на основі розподіленої архітектури, адже проблеми, пов'язані з аналізом мікросервісних архітектур, особливо унікальні та значно відрізняються від типових проблем монолітних систем [1].

Програми, які базуються на використанні архітектури мікросервісів, вимагають моніторингу з тих самих причин, що і будь-який інший тип не тільки розподілених системи, а й будь-яких систем загалом: всі системи з часом починають виходити з ладу, або в їх роботі починає з'являтися поведінка, яка відрізняється від прогнозованої чи запланованої, що зрештою має негативний вплив на користувацький досвід, показники ефективності системи, інколи навіть до безповоротної втрати даних чи іншої шкоди, що є просто неприпустимим. Звісно, запобігання виходу системи з ладу або уникнення неправильного її функціонування завдяки обробці очевидних маркерів є найбільш очевидною причиною застосування засобів моніторингу, але є ще й інша мета. Наприклад, дослідження неочевидних тенденцій.

Наприклад продуктивність. Так, продуктивність системи не є просто двійковим поняттям. Системи не просто "працюють" або "не працюють". Складні системи, навіть моноліти, можуть працювати в неправильному стані досить довго, що впливає на продуктивність, тобто система все ще працює, однак це слово не відображає сповна всього стану [1]. Таким чином, немає очевидного сигналу про те, що система знаходиться в критичному стані, проте можна відслідкувати певні тенденції. Ці деградовані стани часто віщують неминучі крахи системи в майбутньому. Моніторинг поведінки систем може попередити спеціалістів про погіршення стану з часом, що допоможе запобігти повних збоїв в майбутньому.

Системи моніторингу дозволяють з часом збирати цінні дані, які можна використовувати для підвищення продуктивності обслуговування системи,

чи для впровадження покращень задля кращого функціонування або більш надійного захисту. Дані про відмови та показники продуктивності в певні мпроміжки часу можна проаналізувати, щоб знайти закономірності при збоях у системі, які можна співвіднести з певними подіями, встановивши нові види маркерів. Наприклад у випадку, коли дані вказують, що високий відсоток загальних збоїв системи відбувається протягом певного фіксованого часу після нового розгортання, це було б потужним показником того, що процеси розгортання потребують уваги та вдосконалення.

Таким чином, цінність засобів моніторингу не викликає сумніву. Скоріше виникає інше питання – як використовувати наявні рішення моніторингу правильно, або що можна створити, щоб моніторинг був найбільш ефективним. На даний момент існує чимало рішень у цій області, однак потенціал для створення нових досить великий. Зокрема, у зв'язку із широким успіхом розподілених систем, та своєрідній революції у сфері баз даних із нахилом до так званих NoSQL, видається інтуїтивно цікавим використати графові бази даних для задач моніторингу таких систем, враховуючи їх природню спорідненість. Це може мати великий успіх та значну практичну цінність.

1. РОЗДІЛ 1: ОГЛЯД СУЧАСНИХ МЕТОДІВ ТА ІНСТРУМЕНТІВ АНАЛІЗУ СЕРВІС-ОРІЄНТОВАНИХ ТА МІКРОСЕРВІСНИХ АРХІТЕКТУРНИХ РІШЕНЬ

1.1 Проблематика моніторингу великих систем. Маркери моніторингу

Моніторинг - це процес збору, зберігання та відповідної обробки даних про роботу певної системи. Відповідно постає перше запитання - які дані підлягають обробці, або іншими словами – які дані варті уваги в контексті дослідження поведінки системи, що саме варто відстежувати. Великі розподілені системи обробляють величезний обсяг даних щодня, потенційно генеруючи гігабайти нових метаданих про свою поведінку за короткий час (і це попри гігантського обсягу бізнес-даних та інших даних, які обробляються чи генеруються в процесі роботи самої системи). Таким чином існує проблема вибору релевантних даних серед надвеликого обсягу. Очевидно, обробляти всі дані часто є практично неможливою задачею як з точки зору обчислювальних потужностей, так і з точки зору часових затрат. Більше того, обробка абсолютно всього потоку даних системи є просто нераціональним процесом, адже більша частина даних є практично повністю знеціненою в контексті важливості при дослідженні системи. Тож зрештою фінальним є питання: як знайти сигнал серед шуму? Для кожної програми існують свої критерії, за якими можна відносити дані або до шуму, або до важливих характеристик. Це залежить від специфіки області застосування системи, із власне її архітектурою, тощо. Завдання вибору ключових метрик повинно вирішуватися провідними інженерами, які мають глибоке розуміння логіки потоків даних всередині системи та, зрештою, детальне розуміння архітектури системи. Проте є кілька загальних показників, варті уваги, які можна зустріти майже у кожній системі. Облаштовуючи надсистему моніторингу, в першу чергу необхідно подбати про те, що ця система ефективно відслідковує такі ключові метрики і вже

далі брати до уваги специфічні для даної конкретної системи маркери. Вдласне, такі розповсюджені метрики включають наступні.

1.1.1 Метрики програми

Ці показники стосуються конкретної програми. Зазвичай об'єктом дослідження в такому випадку є певний внутрішній параметр, що відноситься до поточного бізнес-домену, чи якийсь специфічний показник, часто створений конкретно для даної платформи, яки базується на, знову ж таки, специфічних даних і є своєрідним збірним поняттям, що дає уявлення про певну складову системи. Якщо програма, наприклад, є частиною трейдингової платформи, стандартною метрикою для неї може бути кількість виконаних операцій купівлі\продажу торгових інструментів клієнтом за минулий день. Система для здійснення онлайн-покупок може відслідковувати середню кількість покупок клієнтами в певні часові моменти. Ці дані верхнього рівня корисні для команд розробників та організації для розуміння функціональної поведінки системи. Якщо в середньому для даної системи характерно, що користувачі ввечері п'ятниці здійснюють 1000 покупок, а в один момент цей показник дорівнює 400, це може бути серйозним індикатором проблем. В більшості систем моніторингу передбачені спеціальні правила, що базуються на певних регресійних характеристиках, у випадку порушення яких відбувається автоматичне сповіщення про підозрілу поведінку. Це є першим кроком до оперативного виявлення збоїв у системі.

Що ж стосовно неспецифічних програмних метрик, то до них часто відносять такі:

1) *Загальна продуктивність системи. Оцінка APDEX*

Методика APDEX – широко розповсюджений міжнародний стандарт оцінки продуктивності інформаційних систем, що являє собою акронім (з англ. **A**pplication **P**erformance **i**n**D**EX – індекс продуктивності програми). Його ідея – описати продуктивність системи в числовому вигляді, тобто

надати певну її числову оцінку. Для її отримання застосовується наступний алгоритм [2]:

а) Виокремлення списку ключових операцій (тобто із загального масиву всіх внутрішніх операцій вибрати тільки ті, які безпосередньо впливають на продуктивність)

б) Приоретизація ключових операцій

в) Визначення цільового часу для кожної із операцій (визначення нормального часу виконання, середньо очікуваного значення)

г) Фізичне вимірювання безпосереднього часу виконання ключових операцій (при цьому важливо проводити вимірювання на достатньому проміжку часу, щоб отримати доволі точне усереднене значення)

д) Отримання власне оцінки APDEX

$$APDEX = (N_{\text{цїл}} + N_{\text{пер}}/2)/N$$

де N – загальна кількість виконан певної ключової операції

$N_{\text{цїл}}$ – кількість виконань певної ключової операції, час виконання яких менший або рівний раніше встановленому цільовому часу

$N_{\text{пер}}$ – кількість виконань певної ключової операції, час виконання кожного з яких перевищує раніше встановлений цільовий час

е) Інтерпретація отриманих числових значень (зазвичай, оцінка відбувається в категорії «добре-погано»)

2) *Середній час відгуку*

Дана метрика відображає середню затримку певної функції. Тут можна оцінювати, наприклад, час відгуку при проведенні операції оплати, замовлення на сайті, час отримання відповіді при запиті до конкретного внутрішнього сервісу системи, тощо

3) *Середня завантаженість системи\елементу системи*

Доволі популярна метрика, що дозволяє перш за все переконатися, чи справляється система із кількістю користувачів, що в один момент її

експлуатують, чи наявна відповідність заявленим нормам. Також можна концентрувати увагу на окремих компонентах системи, відстежуючи появу «вузьких місць». Це створює підстави для подальшої оптимізації або масштабування системи згідно нових умов навантажень.

4) *Частота помилок*

Відстеження кількості внутрішніх помилок або виключних ситуацій (англ. Exceptions) дозволяє отримати картину стабільності системи, в той же час це допомагає перевірити, чи достатньо добре налаштовані механізми захисту системи, чи не має в системі занадто багато вразливих місць, які користувач може випадково чи навмисно експлуатувати (тобто чи наявні в системі місця, де користувач має змогу виконати певні дії, здатні нашкодити їй загальному)

5) *Кількість працюючих екземплярів додатків\компонентів*

Ця метрика доволі поширена у розподілених системах, особливо в контейнеризованих середовищах із застосуванням стратегій реплікації. Вона дозволяє відслідковувати скільки реплік того чи іншого мікросервісу\компоненту системи зараз запущено. Враховуючи передумову створення реплік (відповідь на підвищену навантаженість на даний вузол), це дає ще один спосіб виявити «вузькі місця» системи, перевірити стійкість до підвищеного навантаження. Доступність програми

6) *Маніпуляція з пам'яттю. «Збір сміття»*

Одним із найважливіших технічних показників системи є, безумовно, використання пам'яті. Дуже часто, системи зазнають краху саме через нестачу пам'яті на сервері, тож додаток повністю припиняє свою роботу. Оперативне відслідковування використання пам'яті дозволяє перевірити, чи ефективно додаток використовує надані йому ресурси, що в свою чергу відкриває двері до оптимізації механізмів. Зокрема це може створити прецедент для перевірки механізмів очистки пам'яті, або як їх ще називають – «механізмів очистки сміття», тобто невикористовуваних більше об'єктів чи файлів всередині додатків. Інколи показник використання пам'яті в динаміці дає чітко зрозуміти, що подібні механізми не працюють взагалі.

1.1.2 Метрики платформи

Ці показники містять звіти про основні складові вашої інфраструктури. В першу чергу вони дають змогу відслідковувати рівень функціонування ваших механізмів. Ці метрики знаходяться у контексті дещо вищому, ніж програмні, бо сконцентровані скоріше на компонентах як окремих одиницях і взаємодії між ними. Сюди можна віднести наступне [1]:

1) *Усереднене значення часу виконання для кожного із десяти найчастіше виконуваних запитів до бази даних (БД):* дає приближене значення найбільш ймовірного значення часу очікування на виконання запиту до БД

2) *Усереднене значення часу виконання для запитів до БД, що входять у число 10% найповільніших:* усереднений екстремум часу виконання запиту до БД

3) *Усереднене значення часу виконання для запитів до БД, що входять у число 10% найшвидших:* усереднений інфінум часу виконання запиту до БД

4) *Кількість запитів за певний фіксований проміжок часу, що надходить до тієї чи іншої служби\мікросервісу:* приблизна картина завантаження тієї чи іншої служби\сервісу. Корисна інформація як для навантажувального аналізу, так і при масштабуванні архітектури з метою отримати чітку відповідь – чи варто утворювати додаткові залежності з поточним сервісом, чи краще спробувати знайти інший шлях.

5) *Усереднене значення часу отримання відповіді при надсиланні запиту до певної кінцевої точки (англ. endpoint) системи:* оскільки в розподілених та мікросервісних системах кінцеві точки є дуже поширеним поняттям (як складова основного протоколу міжсервісної взаємодії REST), дослідження часу відповідей кінцевих точок дає інформацію про загальну динаміку міжсервісного спілкування, або дозволяє виявити певні «вузькі місця» у внутрішній комунікації.

6) *Значення коефіцієнту успішного\неуспішного функціонування тієї чи іншої служби\мікросервісу:* дозволяє виявити несправності у бізнес-логіці

чи потенційне некоректне використання того чи іншого сервісу.

У сукупності ці показники забезпечують своєрідну інформаційну панель, яка може дати чітку картину продуктивності та поведінки системи низького рівня. У випадку грамотного аналізу, дані показники можуть попередити про погіршення продуктивності, що може мати негативний вплив на загальну пропускну здатність системи, або призвести до загальносистемного збою. Важливо використовувати дані низького рівня таким чином, щоб передбачити та запобігти ширшим збоям до їх виникнення, тобто детально аналізувати загальні тенденції, щоб виявити певну негативну закономірність, що веде до потенційного краху ще до того, поки цей крах стане реальним або неминучим. Зазвичай, сучасні рішення моніторингу (які будуть описані далі) пропонують широкофункціональну та детальну інформаційну панель для вашого продукту, яка оснований на описаних вище маркерах.

1.1.3 Внутрішньосистемні події

Звичайно, на системи впродовж всього їх життєвого циклу, діють різноманітні зовнішні сили. Наприклад, найпоширенішими та потенційно найбільш загрозливими є нові розгортання системи (або так званий деплоймент, події розгортання). Це момент, коли часто система повністю зупиняється, а її кодова база оновлюється протягом цього часу. У випадку проблем, які можуть виникнути під час цього процесу, система може припинити функціонування без можливості відновлення своєї роботи без зовнішнього втручання спеціалістів. Навіть у разі успішного розгортання, завжди існує небезпека появи нових загроз чи збоїв у роботі, які не відстежувались у попередніх версіях системи. Спеціалістам, що контролюють такі процеси добре відомо, що існує сильний взаємозв'язок між розгортанням нового коду та системними несправностями. Враховуючи такі тенденції, є сенс створити журнал подібних подій та інтегрувати його з даними часових рядів, що реєструються програмами, разом з рештою метрик. Події масштабування, оновлення конфігурації

системи чи інші операційні зміни також є актуальними та несуть в собі певні загрози, тож і їх слід реєструвати. Контроль цих подій також дасть можливість статистично співвіднести їх із поведінкою системи, несподіваними змінами в її роботі, проблемами, що з'явилися неочікувано.

1.2 Принципи аналізу мікросервісів

Звісно, для аналізу систем, як мікросервісних, так і монолітних, в першу чергу варто використовувати аналіз метрик, описаних раніше. Це стандартний підхід, своєрідна база для подальших досліджень. Більшість сучасних інструментів моніторингу основуються саме на цих метриках. Однак, помимо цього, існує чимало принципів та стратегій моніторингу систем відповідно до їх типу, що дозволяє провести більш системний аналіз. Зокрема, притаманно розподіленим та мікросервісним архітектурам, можна виділити, наприклад, наступні п'ять принципів, які дозволять вам врахувати як технологічні зміни, пов'язані з мікросервісами, так і організаційні зміни, пов'язані з ними [3].

1.2.1 Контроль контейнерів та їх вмісту

В наш час, слова «мікросервісна архітектура» та «контейнеризація» дуже часто вживаються в одному контексті, адже тепер технології контейнеризації стали фактично стандартом при розгортанні розподілених архітектур.

Контейнери - це легковісна альтернатива віртуальним машинам для забезпечення ізольованих робочих середовищ для ваших робочих навантажень [4]. Вони набули широкої популярності як будівельні блоки для мікросервісів. Швидкість, портативність та ізольованість контейнерів фактично значною мірою забезпечили успіх моделі мікросервісів. Про переваги контейнеризації сьогодні сказано багато і вже неможливо знайти розробника, що працює в області розподілених рішень та не знайомий із технологіями контейнеризації.

Контейнери - це своєрідні «чорні ящики», які неймовірно корисні для розробки та забезпечують високий рівень портативності в процесі усього життєвого циклу продукту: від стадії розробки до стадії виробництва, від робочої машини розробника до хмарних середовищ [3]. Але в процесі експлуатації, аналізу систем та усунення несправностей сервісів чи компонентів, ці самі «чорні ящики» ускладнюють загальну операційну діяльність, так як постійно змушують задумуватися над питанням: що працює в контейнері? Як працює програма/код?

Отже головним завданням в цьому контексті є забезпечення якісного орієнтування всередині контейнерів, в тому, що відбувається всередині них, а не простоконтроль того факту, що вони працюють. Зазвичай, для подібних завдань існує агентно-орієнтований підхід: створюється «агент», який живе в просторі користувача хоста або віртуальної машини та збирає необхідну інформацію [3]. Проте цей підхід у випадку розподілених контейнеризованих середовищ показує себе не дуже добре, адже це йде у розріз із однією із головних переваг контейнерів – вони являють собою невеликі, а саме головне - ізольовані процеси із якомога меншою кількістю залежностей, в той час коли «агент» - це, звичайно, нова залежність. Також, запуск тисяч агентів моніторингу навіть при невеликому розгортанні - це досить коштівний процес у контексті використання ресурсів та велика проблема для «оркестрації» мікросервісів.

Виходячи з цього, в даному випадку виникають два потенційні рішення: 1) стимулювати розробників безпосередньо інструментувати свій код, або 2) користуватися універсальним підходом до інструментації на рівні ядра, щоб мати можливість відслідковувати всю активність додатків та контейнерів на ваших хостах [3].

1.2.2 Експлуатація систем оркестрації для контролю ефективності сервісів

Ще одним завданням, що входить до спектру моніторингу мікросервісних систем, є виявлення закономірностей в експлуатаційних даних в контейнерному середовищі. Звісно, показники одного контейнера мають набагато нижче граничне значення, ніж сукупна, а значить усереднена інформація з усіх контейнерів, що разом утворюють собою функцію чи, іншими словами, сервіс системи. Особливо це стосується інформації на рівні додатків, наприклад: які запити до кінцевих точок сервісів мають найменший час отримання відповіді, чи які запити до баз даних найбільше асоціюються з помилками. Проте також це ще й стосується моніторингу на рівні інфраструктури, наприклад, контейнери яких сервісів використовують найбільше ресурсів, крім виділених спільних ресурсів процесора, які контейнери найбільше асоціюються із високими показниками навантажень, великою кількістю реплік, тощо.

Сьогодні багато складних системах дедалі частіше в процесі розгортання програмного забезпечення покладаються на системи оркестрації, які забезпечують перехід від логічного плану системи до фізичних контейнерів. До таких систем можна віднести такі інструменти, як Kubernetes, Mesosphere DC/OS та Docker Swarm. Зазвичай розробники використовують систему оркестрації для того, щоб, з однієї сторони, визначити самі мікросервіси та, з іншої сторони, визначити поточний стан кожного сервісу в розгорнутому середовищі.

Тож перед розробниками систем ставиться завдання вдосконалити сповіщення системи оркестрації таким чином, щоб мати змогу збирати якомога більше даних про характеристики життєвого циклу сервісів. Ці сповіщення - це своєрідна система захисту, яка першою допомагає оцінити, чи є якісь зовнішні впливи на додаток. Але, звісно, оперувати цими сповіщеннями є складним, або й взагалі неможливим завданням, якщо ваша система моніторингу не є контейнерною.

Контейнерні рішення використовують метадані оркестрації для динамічної обробки даних контейнерів та додатків з метою обчислення метрик моніторингу на основі кожного сервісу індивідуально. Існують різні рівні ієрархії, які можуть бути піддані аналізу, їх відмінність прив'язана до відмінностей систем оркестрації. Наприклад, для Kubernetes це простір імен, ReplicaSets, Pods і контейнери. Агрегація даних на кожному з цих рівнів є важливою для логічного усунення несправностей, незалежно від фізичного розгортання контейнерів, що утворюють сервіс [3].

1.2.3 Ефективна експлуатація еластичних та багатопозиційних сервісів

У контейнеризованих середовищах швидкість змін природньо є дуже високою та значно вищою, ніж, наприклад, у контейнеризованих, і це необхідно враховувати. Варто пам'ятати про так звані еластичні, або багатопозиційні сервіси, яким притаманна висока динаміка зміни, підлаштування під оточення, що міняється. Вони на разі доволі часто зустрічаються у багатьох системах. Системи моніторингу повинні бути підлаштовані відповідно, інакше така динаміка може зашкодити їх нормальному функціонуванню.

Часто раніше, моніторинг застарілих систем вимагав ручного налаштування метрик та перевірок, для яких приходилося індивідуально розгортати кожен із сервісів. Це налаштування могло бути доволі специфічним та досить схожим на визначення окремих показників, які потрібно зібрати, або налаштування колекції даних на основі того, яка програма працює в конкретному контейнері. Очевидно, такий підхід недопустимий у великих системах (адже вимагає занадто багато уваги та сили), хоча це все ще може бути прийнятним у невеликих масштабах. Враховуючи темпи зростання, все таки варто робити акцент на дослідження великих систем. Відповідно до цього, стратегія моніторингу розподілених та сервісно-орієнтованих систем повиненна мати можливість вчасно рости та

зменшуватися згідно із темпом змін еластичних сервісів, без різномісних додаткових ручних налаштувань з боку людини людини.

Звісно, основний акцент варто робити на виробничому середовищі, адже саме там моніторинг є найбільш важливим та доцільним. На виробничому середовищі моніторинг повинен вибудовуватись у напрямку складного розгортання, яке охоплює кілька центрів обробки даних або кілька хмар [3]. Це якраз і спонукає до впровадження системи моніторингу, яка може оперативно охоплювати ці різні місця, а також безперебійно працювати в динамічних системах, що містять контейнери.

1.2.4 Моніторинг API

У розподілених середовищах мікросервісів API є одним із найважливіших термінів. Вони є, по суті, єдиними елементами сервісу, які доступні іншим командам. Це запорука міжсервісної взаємодії, яка власне і відкриває дорогу самій ідеї розподіленості. Фактично формати відповідей та узгодженість API складають так звану “внутрішню угоду на рівні обслуговування”, навіть якщо таку не визначено формально.

Безумовно, моніторинг API – це одна із основних задач моніторингу як такого. Моніторинг API може мати різний вигляд, але, очевидно, він повинен виходити за межі двійкових перевірок в стилі «працює/не працює». Наприклад, цінним є розуміння найбільш часто використовуваних кінцевих точок як функції від часу. Це дозволяє командам побачити, чи не змінилося щось важливе у використанні сервісів, чи це відбулося через зміну дизайну чи зміну користувача.

Можна розглянути найповільніші кінцеві точки вашого сервісу, оскільки вони можуть сигналізувати про значні проблеми або, щонайменше, вказати на області, які потребують найбільшої оптимізації у вашій системі.

Нарешті, можливість відстеження викликів сервісів через вашу систему представляє ще одну важливу можливість. Цей тип профілювання, як

правило, використовується розробниками та допомагає зрозуміти загальну взаємодію з користувачем.

1.2.5 Синхронні організаційні зміни при впровадженні систем моніторингу

Не дивлячись на те, інформація, приведена вище, була зосереджена на технологічних аспектах мікросервісного моніторингу, як і будь-яка інша технологічна ідея, це стосується не тільки програмних бітів, але й людей.

Тим, хто знайомий із законом Конвея, відомо, що дизайн систем визначається організаційною структурою команд, що їх будують [3]. Активне впровадження більш швидкого, гнучкого програмного забезпечення, забезпечене розвитком розподілених архітектур, змусило команди задуматися про аналогічну реструктуризацію своєї організаційної структури, тобто ці ідеї були перенесені в тому числі і на організаційний рівень команд.

Отже, якщо та чи інша організація хоче ефективно використовувати нові підходи в програмному забезпеченні, запропоновані мікросервісними архітектурами, то і її команди, що працюють з цими технологіями, повинні в свою чергу самі в певному сенсі відображати мікросервіси. Що це означає на практиці? Менші команди, слабо зв'язані, які можуть обирати свій напрямок індивідуально до тих пір, поки він все ще узгоджується із загальним напрямком організації.

З точки зору моніторингу, це відкриває двері для створення індивідуальних платформ моніторингу, кожною із яких керує певна команда, якій доступні свої системи сповіщень, інформаційні панелі, метрики та показники, при цьому вона не позбавлена загального бачення продукту.

1.3 Огляд деяких популярних сучасних інструментів

Визначна популярність мікросервісних та розподілених архітектур спричинила появу великої кількості інструментів для їх аналізу. На ринку існує чимало рішень, кожне з яких має свої сильні та слабкі сторони, є і явні

аутсайтери. Більшість із них базується на метриках, які були приведено раніше. Далі приведено декілька найпопулярніших рішень.

1.3.1 Raygun APM

Платформа APC Raygun - приклад цілісної, основаної на хмарних середовищах системи, яка забезпечує як інструменти та процеси збору інформації, так і інформаційну панель для візуалізації даних метрик мобільних та веб-додатків. За її допомогою можна ефективно відслідковувати масу метрик із категорії поведінки користувачів, продуктивності, помилок та крахів системи.

Розробники-користувачі відмічають простоту установки, інтуїтивно-зрозумілий інтерфейс, низький поріг входу, широкий спектр метрик а також властивість «все в одному місці». Нижче, на рисунку 1.1, приведено приклад інформаційної панелі системи.

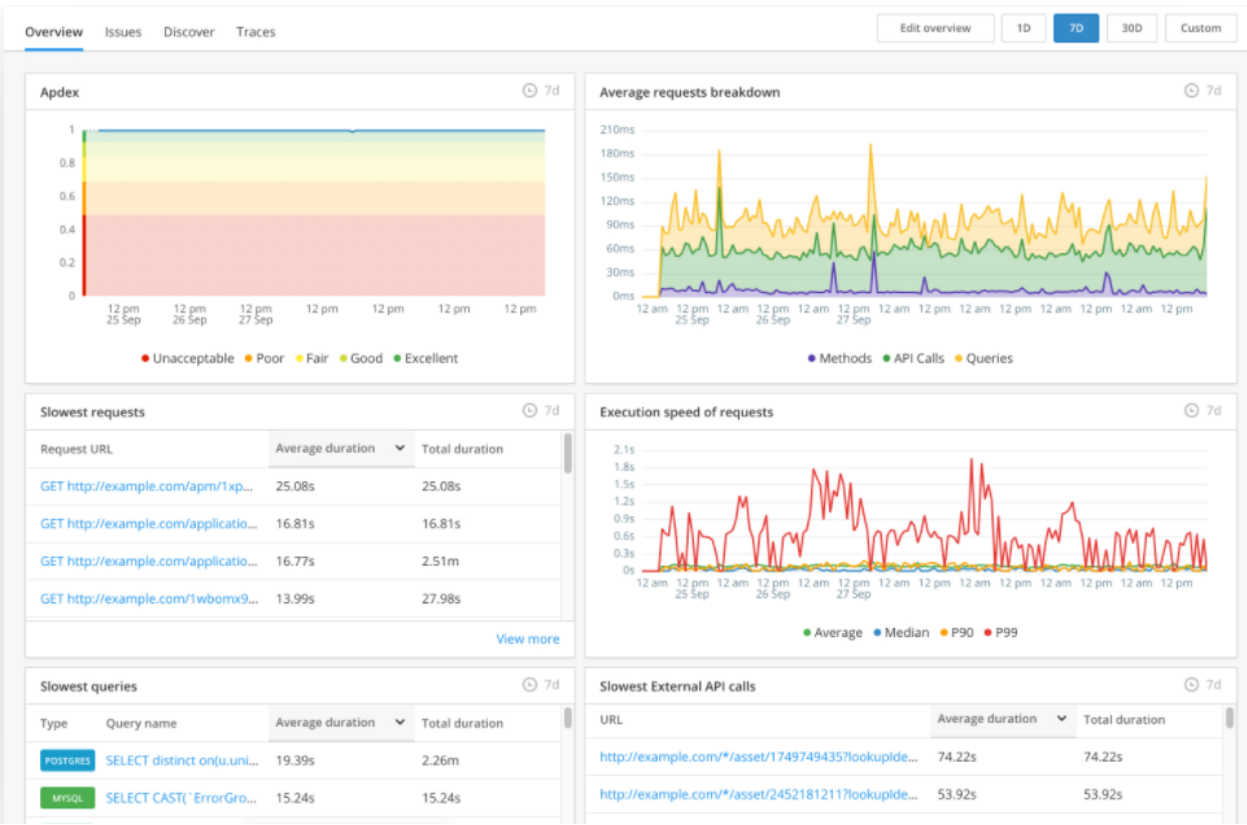


Рисунок 1.1 – Вікно «Огляд» (англ. «Overview») на інформаційній панелі Raygun APM [5]

На рисунку вище можна побачити візуалізацію багатьох метрик, про які йшла мова раніше. Зокрема тут представлені графа APDEX, середній час відгуку запитів, список найповільніших запитів до БД, кінцевих точок чи зовнішніх API сервісів. Помимо цього на вкладці «Проблеми» (англ. «Issues») можна отримати детальну інформацію стосовно проблемних місць системи (рис.1.2), наприклад певні запити до кінцевих точок, що зайняли занадто багато часу, чи наприклад закінчились невдачею.

Add filter +		1D	7D	30D	Custom
Overview Issues Discover Traces					
My issues 4		Active 12	Resolved 0	Ignored 0	Permanently ignored 0
☐	Type	Issue name	First seen	Last seen	Traces
☐	QUERY	SELECT pg_sleep(6) execution >= 2000ms Query took longer than 2000ms	7 days ago	a few seconds...	7,479
☐	TRACE	GET https://testapp.com/slow_endpoint Trace took longer than 3000ms	7 days ago	a few seconds...	7,479
☐	API	https://postman-echo.com/delay/* exe... External api took longer than 2000ms	7 days ago	2 minutes ago	7,476

Рисунок 1.2 – Огляд знайдених проблем системи на інформаційній панелі «Issues» Raygun APM [5]

Панель «Відкриття» (англ. «Discover») надає широку інформацію про популярні запити до API, сервісів, БД, виклики методів системи (рис.1.3). Тут, наприклад, можна проаналізувати запити, що виконувалися найдовше, чи запити, які найчастіше призводили до краху чи невдачі.

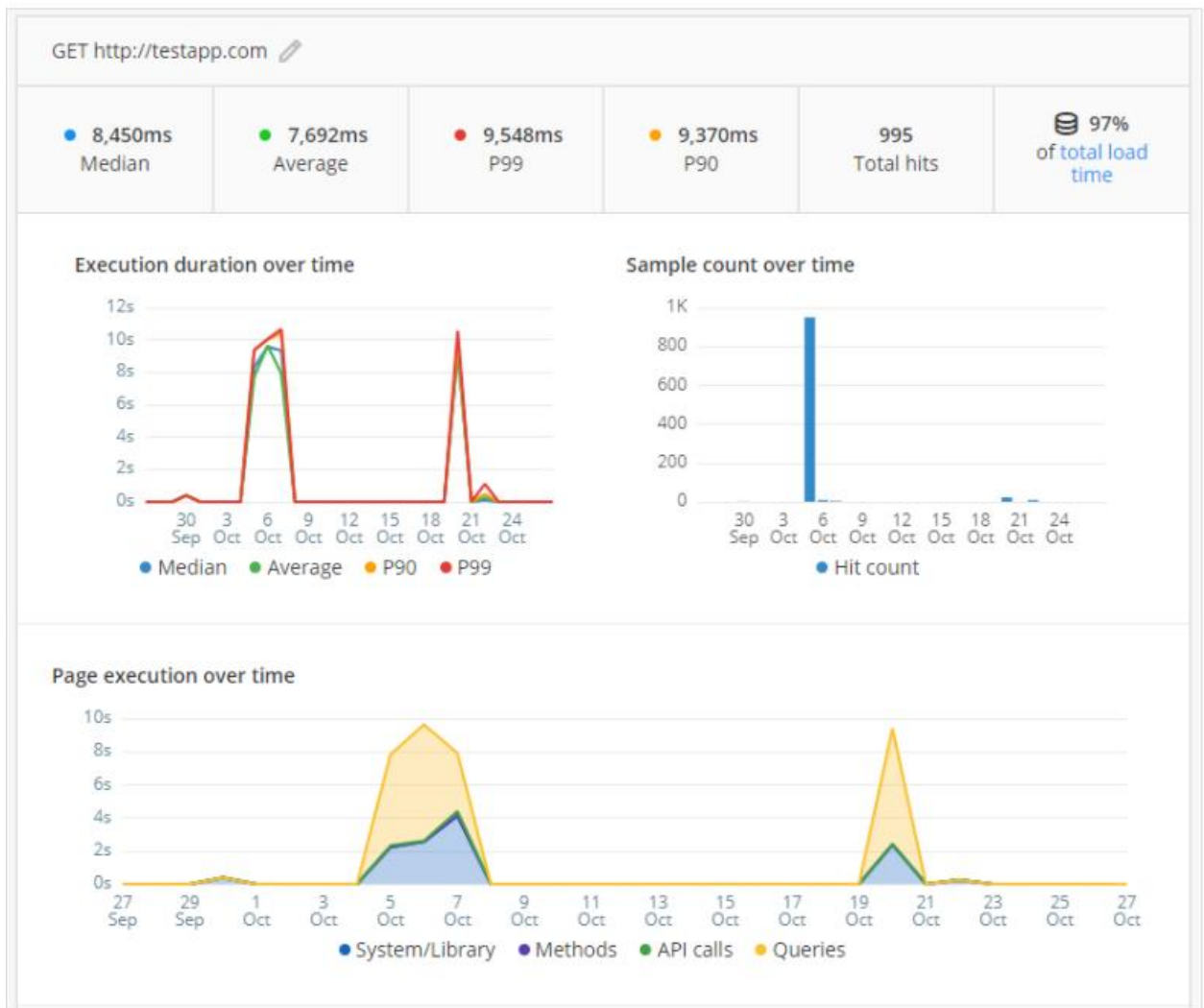


Рисунок 1.3 – Детальна інформація про обраний запит на інформаційній панелі «Discover» Raygun APM [5]

Вкладка «Сліди» (англ. «Traces») дозволяє отримати інформацію про історію (або інакше кажучи, «сліди») внутрішніх та зовнішніх запитів системи, що дає змогу отримати чітку картину поведінки вашого додатку. Приклад одного із віджетів цього вікна приведено на рисунку 1.4.

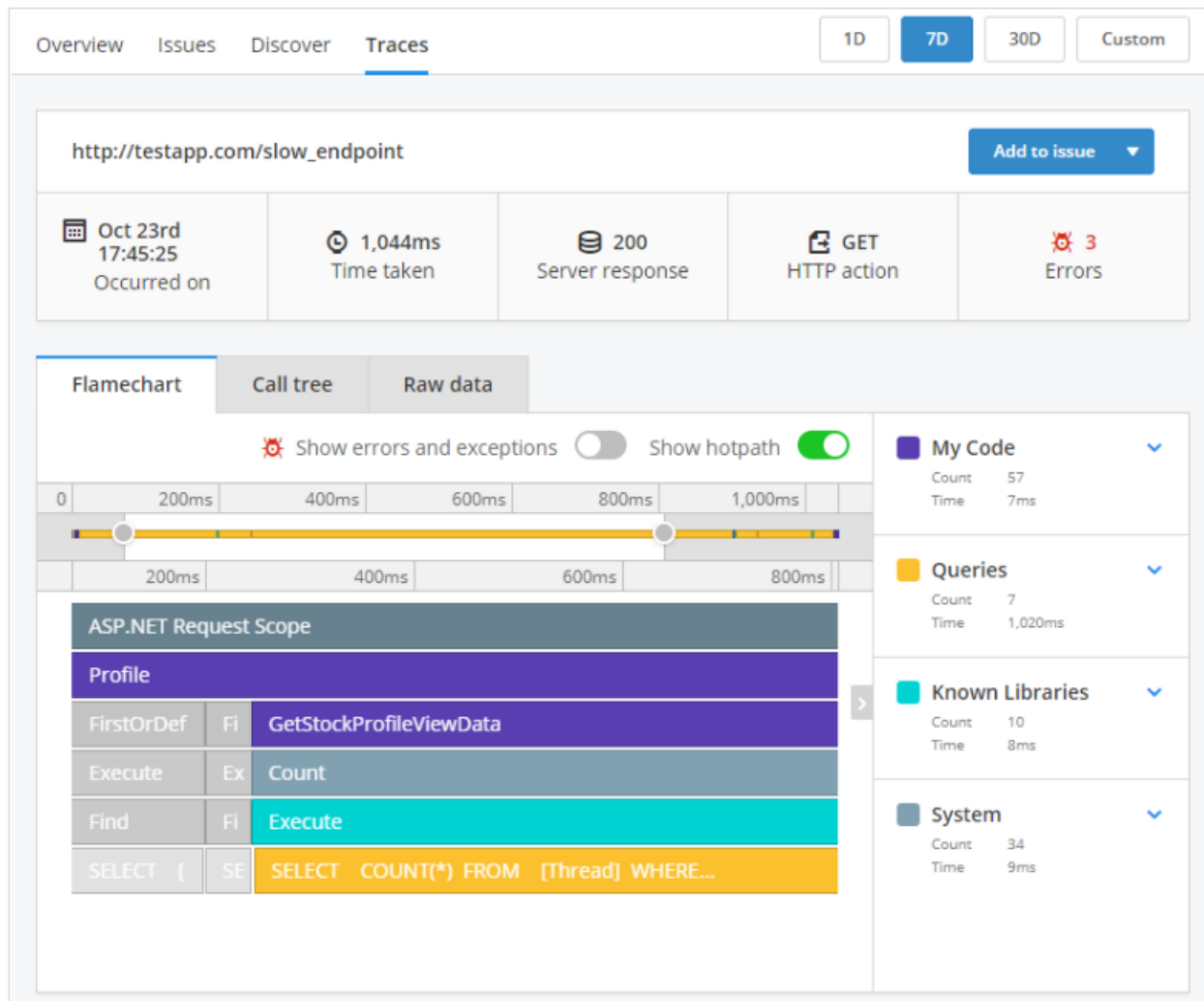


Рисунок 1.4 – Детальна інформація про хід виконання певного запиту на вкладці «Traces» Raygun APM [5]

Із мінусів системи можна виділити відсутність підтримки деяких провідних технологій (наприклад Python, PHP, C++), недостатньо насичену інтеграцію із іншими сервісами (наприклад, Azure DevOps, тощо), відносну дороговизну та дещо незрозумілу в деяких моментах систему оплати.

1.3.2 Stackify Retrace

Stackify Retrace (раніше відомий просто як Stackify) - це надійний інструмент управління та моніторингу ІТ системам, який допомагає як в управлінні інфраструктурою, так і в управлінні продуктивністю додатків

(APM). Даний продукт вирізняється глибоким «розумінням додатків» та винятковим представленням даних. На рисунку 1.5 показано приклад інформаційної панелі системи.

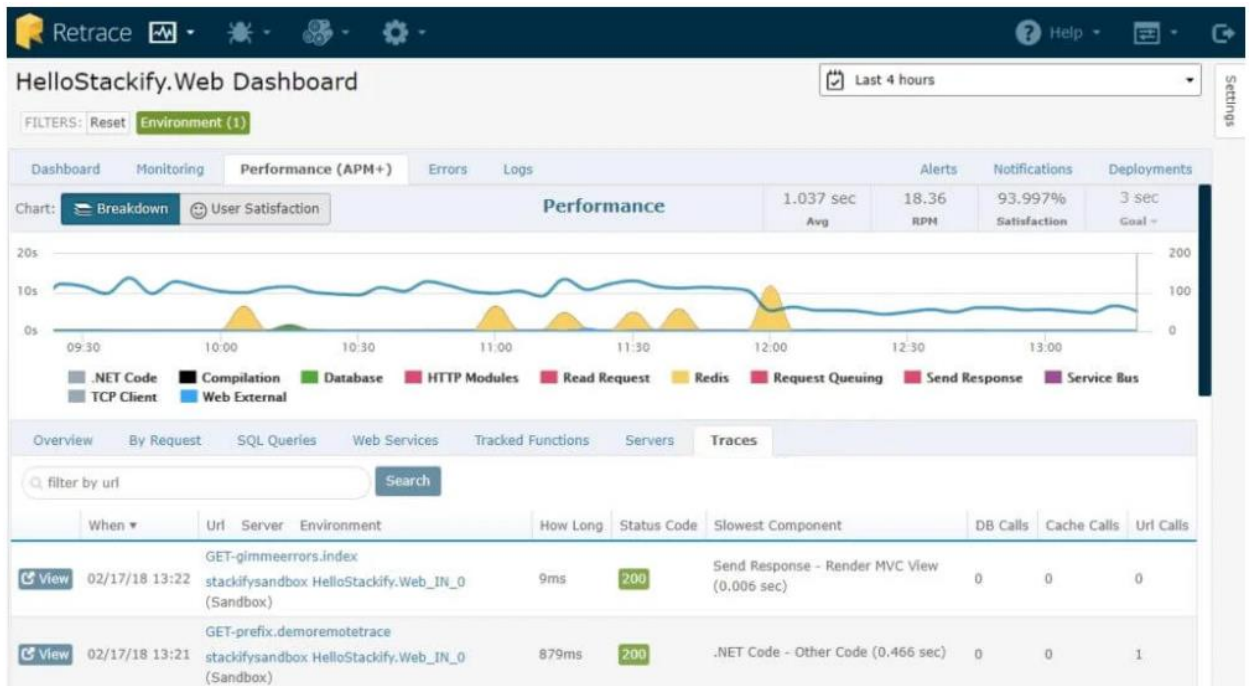


Рисунок 1.5 – Базова інформаційна панель системи Stackify Retrace [6]

Порівнявши із попереднім інструментом можна зробити висновок, що панель не настільки насичена візуалізацією, графіками. Проте в той же час, поточна система пропонує більшу деталізацію – функціонал розбитий на велику кількість допоміжних вікон.

Наприклад, тут також можна отримати детальну інформацію по конкретному запиту (рис.1.6), чи переглянути детальний звіт про певну виявлену в процесі роботи системи помилку (рис.1.7).

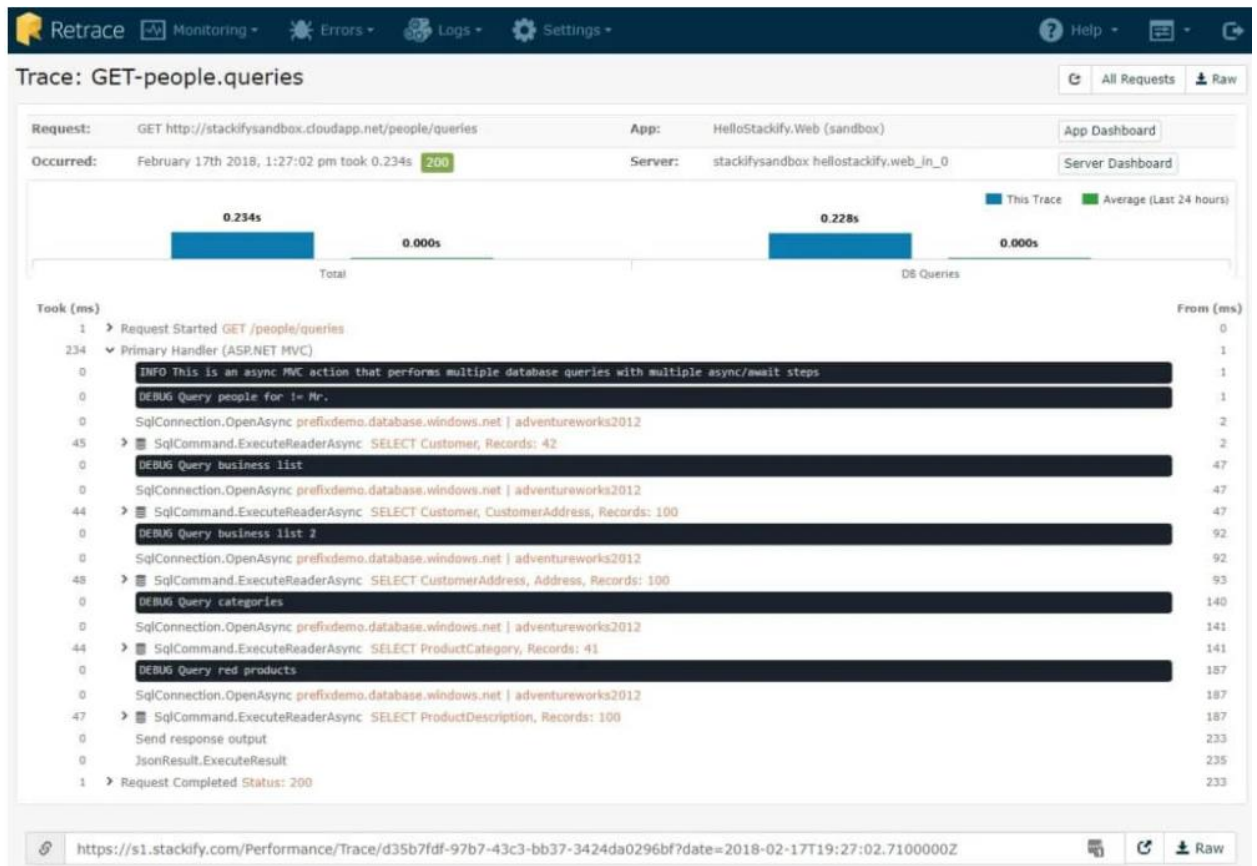


Рисунок 1.6 – Детальна інформація про виконаний запит в Stackify Retrace [6]

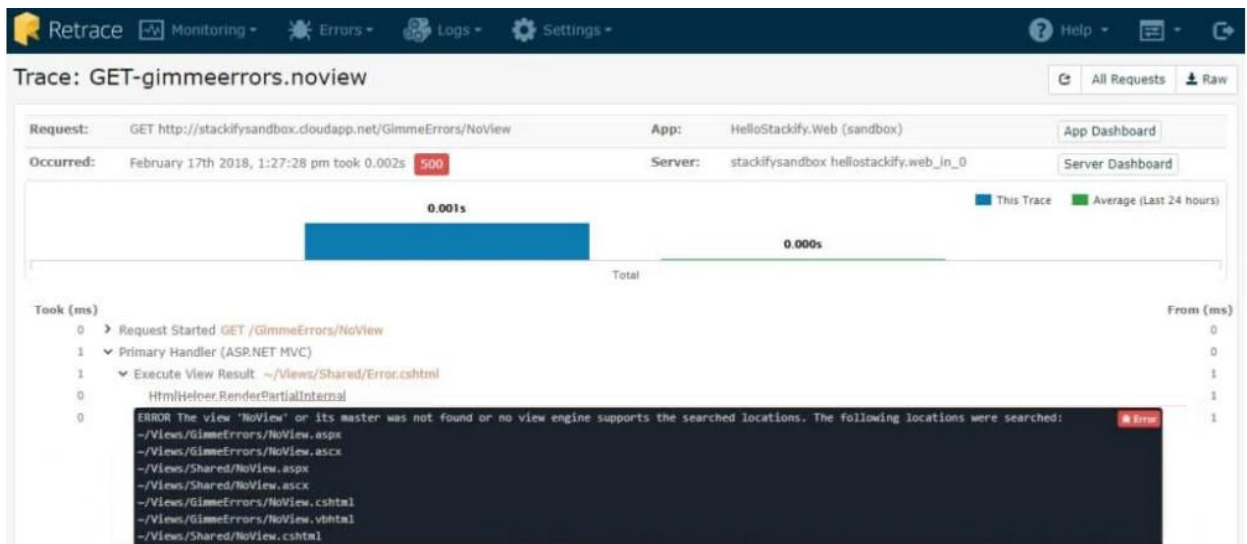


Рисунок 1.7 – Детальна інформація про виявлену помилку в роботі системи на інформаційній панелі Stackify Retrace [6]

В свій час, Stackify широко вдосконалив цей продукт і перейменував його на Retrace. Із визначних змін: він почав підтримувати Java поряд із Microsoft .NET, а також підтримувати моніторинг не веб-програм. Відмінною особливістю Stackify є зосередженість на використанні журналів як ключового джерела інформації про роботу сервера.

Із мінусів можна виділити, звичайно, вузьку спеціалізацію (підтримується тільки .NET та Java) та використання окремого агента під кожен окрему систему.

1.3.3 Prometheus

Prometheus - це класичне рішення для моніторингу систем, спочатку розроблене SoundCloud, що відрізняється від інших систем тим, що його вихідний код відкритий. Він широко використовується для зберігання та обробки „часових рядів” (time series), що являють собою дані, які описують дії з часом. Prometheus часто поєднують з іншими інструментами, особливо з Grafana, для візуалізації даних часових рядів та надання інформаційних панелей. Подальшим його розвитком є проект Prometheus Anomaly Detector (PAD) компанії Red Hat. Розробники відмічають такі особливості Prometheus як багаторозмірну модель даних, гнучку мову запитів для оперування даними (PromQL), підтримку декількох режимів побудови графіків та звітів [7].

В той же час, загальна функціональність інструменту відстає від аналогів, тому Prometheus все таки варто використовувати в першу чергу як доповнення. Як і було сказано раніше, у плані візуалізації даних Prometheus також покладається на інші інструменти. Приклад вигляду інформаційної панелі Prometheus, створеної за допомогою інструменту Grafana приведено на рисунку 1.8.



Рисунок 1.8 – Інформаційна панель Prometheus (на основі інструменту Grafana) [7]

1.3.4 AppDynamics

AppDynamics від Cisco - це потужне рішення APM, яке допомагає організаціям забезпечити ефективну роботу своїх додатків. Платформа підтримує ряд мов, включаючи Java, .NET, PHP, Node.js, C ++, Python та Go.

Хоча це рішення є потужним і може допомогти покращити взаємодію з користувачем, його впровадження може зайняти деякий час. Можливо, також необхідно буде витратити час на навчання працівників тому, як ефективно користуватися цим інструментом, так як поріг входу доволі високий. Більше того, ціна цього інструменту може бути недоступною для багатьох менших незалежних розробників, тому зазвичай цей потужний інструмент використовується у великих компаніях та проектах. Але з іншої сторони,

компанія-виробник Cisco створила собі доволі потужну репутацію, що говорить про високу надійність та продуктивність інструменту за свою ціну. Далі приведено декілька прикладів інформаційної панелі інструменту.



Рисунок 1.9 – Контроль використання пам'яті в AppDynamics [8]

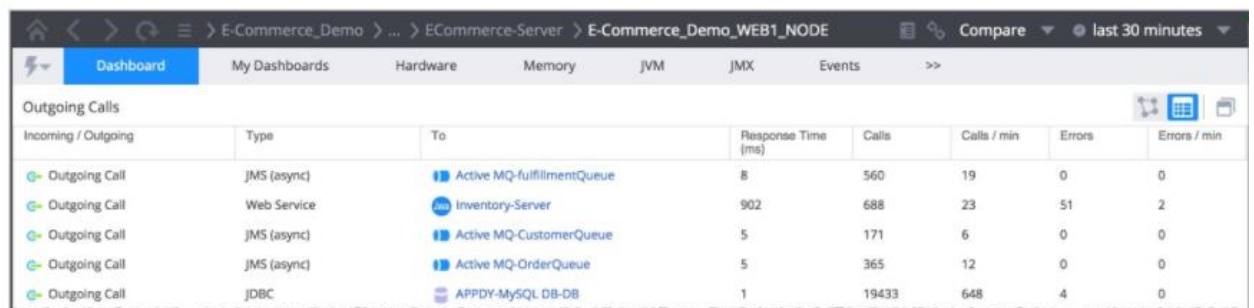


Рисунок 1.10 – Звіт по вузлам додатку в AppDynamics [8]

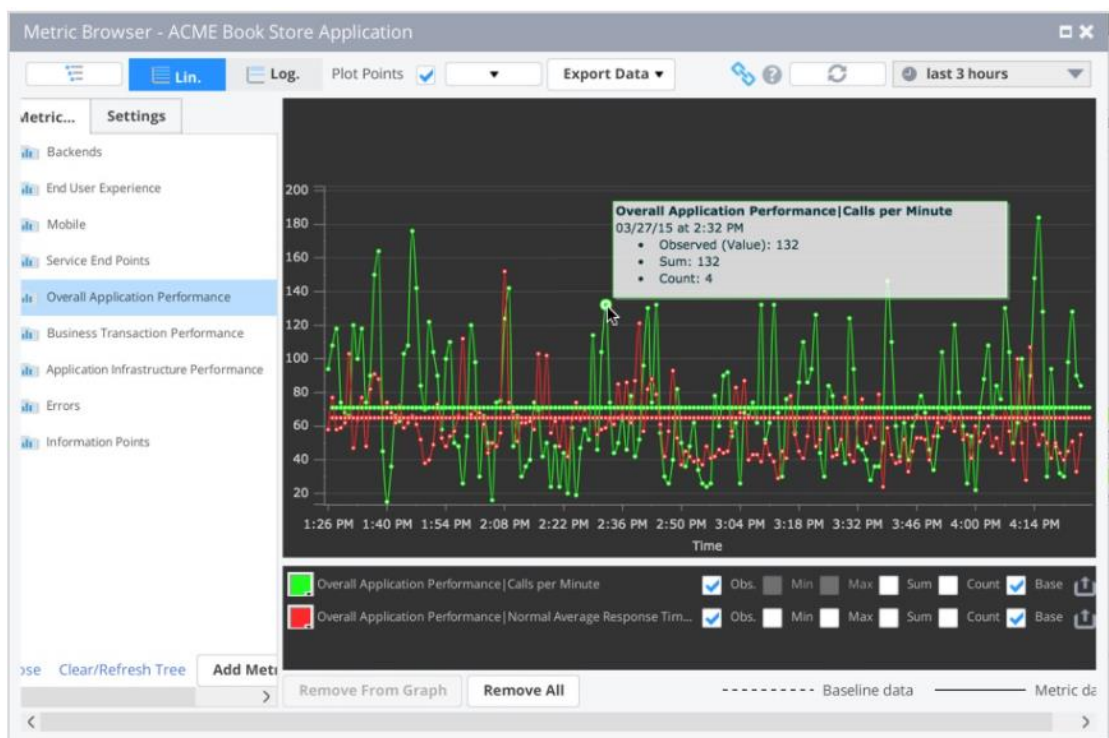


Рисунок 1.11 – Огляд загальної продуктивності додатку в AppDynamics
[8]

1.4 Висновки

На сьогоднішній момент на ринку існує широкий спектр засобів моніторингу різномітних систем, в тому числі й розподілених. Всі вони відрізняються за ціною, потужностями, зручності інтерфейсу, простоті використання, тощо. Однак у більшості своїй вони пропонують однаковий функціонал, зазвичай це вимірювання добре провірених часом метрик, таких як кількість запитів у хвилину, використання ресурсів, відслідковування помилок і так далі. Проте, звичайно, системи можна і треба досліджувати різними способами, використовуючи часом і нестандартні підходи. Оскільки з часом системи все сильніше у складнюються у зв'язку із природнім розростанням та появі нових технологій, інструментарій для дослідження систем повинен також розвиватися. Таким чином варто поступово розширювати технічний кругозір, переходячи від виміру певних числових показників до більш комплексних речей, які даватимуть більше простору для маневру та глибокого аналізу. Одним із потенційних рішень може бути експлуатація теорії графів для опису різного роду систем та їх внутрішніх і зовнішніх зв'язків.

РОЗДІЛ 2. ОГЛЯД ІДЕЇ ЗАСТОСУВАННЯ ТЕОРІЇ ГРАФІВ ДЛЯ ОПИСУ ТА АНАЛІЗУ РОЗПОДІЛЕНИХ ТА МІКРОСЕРВІСНИХ РІШЕНЬ

2.1 Графові бази даних як інструмент для опису та аналізу формалізованих знань

Дуже часто в сфері інформаційних технологій оперують поняттям дані. Однак більш широким та глибоким поняття є саме поняття знань. Фактично, знання – це доволі збірне поняття, що несе в собі якийсь факт, глибоке значення, і це значення в свою чергу вже може базуватися на певних даних, інформації. Однак знання доволі абстрактне поняття, і для оперування знаннями в сфері комп’ютерної техніки необхідна їх попередня формалізація.

Власне формалізація знань - це процес, частина інженерії знань, що являє собою подання попередньо структурованих знань на певній мові представлення знань. Сьогодні існує багато способів представлення знань, як традиційних, так і нещодавно розроблених. Для інтелектуальних комп’ютерних систем, наприклад, актуальними моделями є продукційна із властивим її виведення, семантична із її наочністю представлення понятійних знань (представлення причинно-наслідкові зв’язки між елементами, структури складної системи знань), фреймова із її зручністю в описанні структури і характеристик однотипних об’єктів, логічна, яка дозволяє зручно представляти логічні взаємозв’язки між фактами.

В процесі оперування формалізованими даними безумовно одну з найбільш важливих ролей відіграють бази даних, адже не дивлячись на те, що терміни «база даних» та «база знань» не позначають один і той же термін, ми не можемо обійтись без таких інструментів при будь-яких операцій з даними. Таким чином питання «бути чи не бути» в даному контексті для такого інструменту, як СБД не стоїть, натомість краще звернути увагу на широкий спектр вибору технологій в цій області та, співставивши сильні та слабкі сторони кожного із підходів, визначити, як та чи інша БД може бути

використана із найбільшою ефективністю для вирішення поставленої задачі із обробки формалізованих знань.

Зокрема, одразу на думку спадають класичні рішення, наприклад реляційні бази, та рішення, які вже плавно стають класичними – документні чи NoSQL бази даних. Стосовно класичних БД, то реляційні, як відомо, являють собою прекрасне рішення в задачах класичного зберігання та обробки добре структурованої та схематично визначеної інформації. Іншими словами, вони не дають достатньої гнучкості, яка була б доречною при роботі з формалізованими знаннями, які часто важко строго описати певними схемами. В такому випадку краще глянути в бік тих рішень, які не настільки строго визначені та дають простір для маневру, який дозволить оперувати в тому числі тією інформацією, яку дуже складно або взагалі неможливо точно структурувати.

Якщо ж подібне питання виникає в області мікросервісних архітектур, чи взагалі в області систем із великою кількістю взаємодіючих частин, а завдання, що ставиться, полягає не у підборі інструменту як внутрішньої складової механізмів, а для проведення дослідницької діяльності чи для певного роду моніторингу (тобто коли постає необхідність глянути на всі складові «зверху» та отримати певні знання про них), на думку повинно інтуїтивно спадати рішення, ґрунтоване на теорії графів. Звичайно, мова іде про графові бази даних (ГБД).

ГБД дають необхідну гнучкість при оперуванні даними, адже не накладають строгих обмежень на їх структуру, проте їх головна особливість, яка доречна в цьому контексті – це зручність опису будь-якого роду зв'язків, адже графова модель як найкраще для цього підходить. Поєднання цих двох особливостей відкриває великі можливості для отримання цінних знань про те, що відбувається в тій чи іншій, наприклад, розподіленій системі, де існує маса внутрішніх зв'язків, які можуть таким чином бути детально досліджені (та завдяки відсутності строгих структурних обмежень доволі добре описані).

2.2 Поняття графу знань

Зберігання даних систем та забезпечення доступу до них у класичному їх розумінні, звичайно, не єдина область застосування графових баз даних. Теорія графів разом із ефективними супутніми механізмами обробки дозволяє використовувати ГБД для широкого спектру нетипових завдань, зокрема дуже часто такі технології можуть застосовуватися для дослідницьких цілей та для аналізу систем. В такому випадку часто доводиться мати справу із неструктурованими даними, оперувати скоріше знаннями, ніж даними у звичному розумінні. Однак гнучкість ГБД дозволяє уникнути очікуваних труднощів в такій області. Прикладом такого застосування є використання інструментарію ГБД для створення графу знань.

2.2.1 Огляд ідеї графу знань

Граф знань (ГЗ) – це сукупність взаємопов'язаних описів сутностей - об'єктів, подій або понять [9]. Графи знань створюють контекст даних за допомогою зв'язування та семантичних метаданих, і таким чином забезпечують основу для інтеграції, уніфікації, аналітики та обміну різноманітними даними.

ГЗ представляє сукупність взаємопов'язаних описів сутностей - реальних об'єктів та подій або абстрактних понять (наприклад, документів, веб-сторінок) – в яких:

- Семантика опису сутностей є достатньо формальною, що дозволяє обробляти її ефективно не тільки людям, але й зробити її обробку такою ж ефективною і однозначною в комп'ютеризованих середовищах;
- Сутності посилаються одне на одну, таким чином утворюючи мережу (граф), де кожна сутність представляє додаткову частину опису інших сутностей, пов'язаних з нею, і забезпечує контекст для їх інтерпретації. Тобто взявши групу взаємопов'язаних сутностей, ми отримуємо загальну сукупність пов'язаної інформації, що дає нам певний контекст.

Поодинокі кожна сутність несе мало інформації, своєрідний атомарний факт, проте разом утворюється логічна система.

Для кращого розуміння, приклад графу знань приведено на рисунку 2.1.

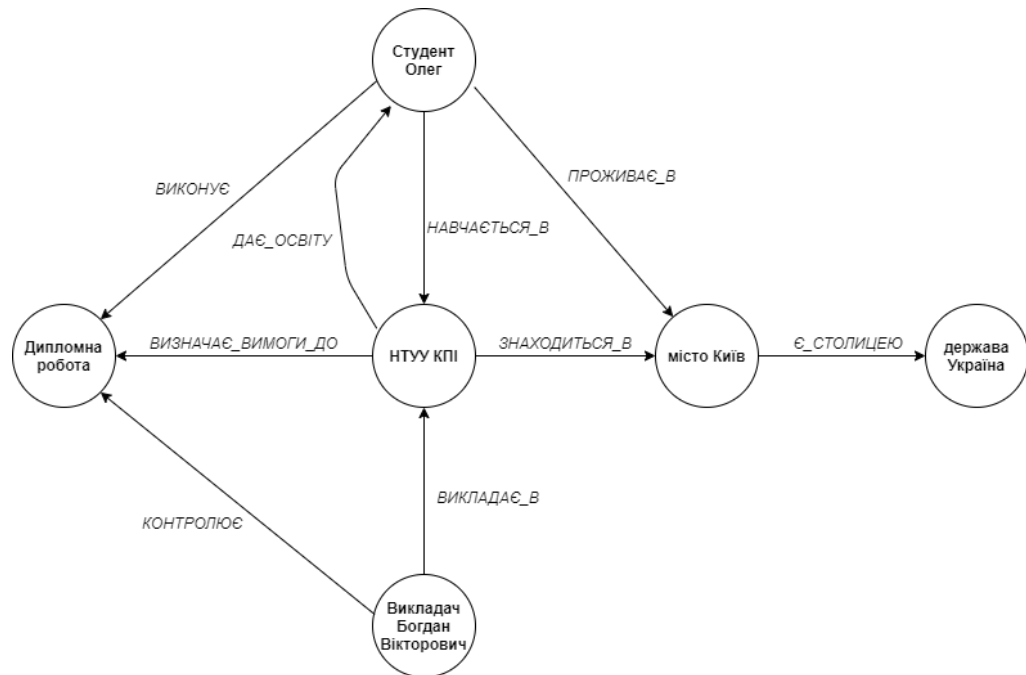


Рисунок 2.1 – Приклад графу знань

Як видно з рисунка, основними елементами графу знань є сутності із відповідними їм короткими описами (наприклад «Студент Олег») та зв'язки між ними (наприклад «НАВЧАЄТЬСЯ_В» між сутностями «Студент Олег» та «НТУУ КПІ»). Поодинокі кожна сутність ідентифікується за допомогою короткого опису, що сам по собі дає, по суті, атомарну операцію. Однак у сукупності всі елементи створюють певний контекст (наприклад, цей невеликий граф знань описує контекст виконання студентом дипломної роботи в університеті).

Можна також помітити, що графи знань поєднують у собі властивості декількох понять, зокрема:

- бази даних, оскільки даними в ГЗ можна маніпулювати за допомогою структурованих запитів

- графів, оскільки ними можна оперувати як будь якою іншою мережевою структурою даних
- бази знань, оскільки вони містять формальну семантику, яка може бути використана для інтерпретації даних та виведення нових фактів

Оскільки для вираження та інтерпретації даних графа знань використовують формальну семантику, існує ряд інструментів подання даних та моделювання [9]:

- **Класи:** це поняття стосується опису сутності та дозволяє утворювати на основі них ієрархію. Наприклад, при роботі з діловою інформацією, граф може містити такі класи, як «Керівник», «Програміст», «Організація», «Проект», тощо. «Керівник» та «Програміст» можуть мати спільний суперклас «Працівник». «Проект», наприклад, може мати численні підкласи, наприклад, «Організаційний проект», «Технічний проект», тощо. Очевидно, ця ідея запозичена із об'єктно-орієнтованої парадигми.
- **Типи відношень:** відношення між сутностями, як можна побачити із рисунку вище, також несуть певну порцію інформації. Зазвичай, вони співвідносяться із типами, на основі яких визначається природа зв'язку між сутностями, наприклад, «Друг», «Роди», «Конкурент», або, як на рисунку вище, «Виконує», «Навчається в» тощо. Типи відношень також можуть мати формальне визначення, наприклад: відношення «є батьком» є зворотним до відношення «є дитиною» та обидва відношення є спеціальними випадками відношення «є родичем», яке в свою чергу є симетричним.
- **Категорії:** сутності може бути додатково об'єднані в певну категорію, яка описує якийсь аспект його семантики, наприклад, "провідні аутсорсингові компанії Києва" або "компанії, що спеціалізуються на розробці програмного забезпечення". Деяка компанія, що займається розробкою ПЗ може належати одночасно до обидвох цих категорій. Категорії описані та упорядковані в систематику.

- **Описи у вільному тексті:** часто елементи графу знань також додатково містять опис у вигляді "людино-дружнього" тексту, щоб уточнити деякі моменти та спростити сприйняття структури людиною. Сам цей опис може не приймати участі у внутрішніх обчислювальних операціях системи.

2.2.2 Приклади застосування графів знань

Безумовно подібна технологія не могла бути залишеною уваги, тому на даний момент в експлуатації знаходиться маса подібних систем, впроваджених передовими технологічними гігантами. Зазвичай, запорукою таких систем є використання потужних систем графових баз даних всередині. Зокрема, можна навести наступні приклади:

- **Граф знань Google.** Власне, термін «Граф знань» якраз і набув широкої популярності, коли компанія Google анонсувала граф знань у 2012 році. Однак велика кількість даних залишається неопублікованою, тому існує дуже мало технічних деталей щодо його організації, охоплення та розміру. Існує також дуже обмежена можливість використання цього графа знань поза власними проектами Google.
- **DBPedia.** Проект, що базується на структурі, притаманній інформаційним скринькам Вікіпедії, забезпечує створення величезного набору даних та онтології, яка охоплює енциклопедичні сутності, такі як люди, місця, фільми, книги, організації, види, хвороби тощо. Цей набір даних є основою руху Open Linked Data.
- **Geonames.** Маштабна система для доступу до 25 мільйонів географічних сутностей та об'єктів.
- **Wordnet.** Приклад використання графових технологій у лексикології. Одна з найбільш відомих лексичних баз даних для англійської мови, що містить визначення термінів та синоніми. Часто використовується

для підвищення продуктивності обробки природніх мов та пошукових програм.

- **FactForge.** Після багатьох років розвитку знань у галузі видавничих новин, Ontotext створив граф своїх знань про зв'язані відкриті дані та статті про людей, організації та місцезнаходження. Він включає дані з описаних вище ГЗ, а також спеціалізованих онтологій, таких як онтологія бізнесу фінансової галузі.

2.3 Сучасні технології графових баз даних

2.3.1 Поняття графових баз даних

Технічно кажучи, графові бази даних - це бази даних, які зберігають дані способом представлення їх у формі вузлів, ребер і зв'язків, які забезпечують суміжність без індексу – можливість пошуку елемента без безпосереднього вказання його індексу.

Таким чином, дані зберігаються у вигляді вузлів, кожен вузол (або блок даних) пов'язаний з іншим, і це з'єднання називається ребром. Ребру також задаються параметри чи властивості, а також напрямок, щоб у такий спосіб визначити зв'язок між одним вузлом і іншим - цей опис називається відношенням. Оскільки можна дуже просто визначити кожен вузол, до якого підключений поточний, (а всі вони пов'язані через ребра), це усуває необхідність пошуку в блоці даних по його «індексу», звідси і термін «суміжність без пошуку». Доволі очевидним застосуванням для такого роду інструментів є соціальні мережі. Тривіальний приклад застосування теорії графів в області соцмереж приведений на рисунку 2.2.

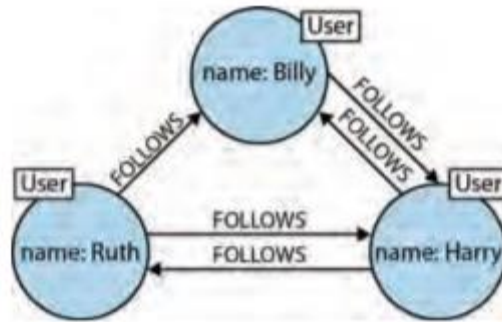


Рисунок 2.2 – Граф, що позначає трьох користувачів та зв'язок «слідкує» між ними

Сьогодні більшість сайтів соціальних мереж, таких як Facebook, використовують графові бази даних для зберігання своєї величезної кількості даних. Помимо цього, безліч рішень із інших областей, таких як наука, державні структури, бізнес, тощо, вже сьогодні або виконують перехід на технології, основні на графах, або вже накопичили досвід їх експлуатації.

2.3.2 Переваги графових баз даних

Не зважаючи на той факт, що практично все можна представити у вигляді графа, повсюдний перехід на графові технології, очевидно, неможливий, адже ми живемо в прагматичному світі бюджетів, проектів з жорсткими графіками, корпоративних стандартів і комерційних правил, які накладають певні обмеження на впровадження нових технологій. Наданий графові базами даних новий потужний метод моделювання даних сам по собі не є достатньою підставою для заміни усталених і зрозумілих платформ обробки даних - від цього повинна бути нехайна і вельми значна практична користь. Для графових баз даних такою мотивацією може послужити застосування її в тих випадках і до таких моделей даних, коли при переході на графову модель буде досягнуто збільшення продуктивності на один і більше порядків. Цього можна добитися, зваживши на наступні переваги графових рішень.

1) *Продуктивність*. Однією з вагомих причин вибору графових баз даних є великий приріст продуктивності при роботі зі взаємопов'язаними даними, в порівнянні з реляційними базами даних і NoSQL-сховищами. На відміну від реляційних баз даних, де облік взаємозв'язків значно погіршує продуктивність запитів на великих наборах даних, продуктивність графових баз даних в цьому випадку залишається незмінною зі збільшенням обсягу збережених даних. Це пов'язано з тим, що запити локалізуються в певній частині графа. В результаті час виконання кожного запиту залежить від розміру частки графа, яку потрібно обійти для задоволення запиту, а не від загального розміру графа

2) *Гнучкість*. Розглянемо наступну ситуацію: розробникам і проектувальникам необхідно організувати взаємозв'язок між даними згідно з вимогами області застосування, водночас структура даних повинна відповідати мінливим потребам, а не нав'язуватися заздалегідь і залишатися незмінною. У графових базах даних це завдання легко вирішується. Графова модель даних відображає і охоплює потреби бізнесу таким чином, що може змінюватися зі швидкістю зміни самого бізнесу.

Власлива графам можливість розширення означає, що можна додавати нові види взаємозв'язків, нові вузли, нові графові мітки і нові підграфи в існуючу структуру, не порушивши при цьому існуючих запитів і функціонал програми. Це позитивно впливає на продуктивність розробки і знижує ризики

для проекту. Завдяки гнучкості графової моделі не є необхідним попередньо моделювати завдання в найдрібніших подробицях, що в реальних проєктах є дуже незручним через швидко мінливі бізнес-вимоги. Здатність графів до розширення також дозволяє зменшити кількість міграцій, що знижує навантаження при обслуговуванні даних і зменшує ризик втрати даних.

3) *Оперативність*. Модель даних повинна не відставати від інших складових частин програми та використовувати технології, які відповідають

сучасним ітераційним методам розгортання програмного забезпечення. Сучасні графові бази даних оснащені всім необхідним для розробки і системного обслуговування. Зокрема, вбудована графова модель даних, позбавлена схем, в поєднанні з вбудованим

програмним інтерфейсом (API) і совою запитів, дозволяє ефективно вести розробку додатків. У той же час завдяки відсутності схеми графові бази даних не передбачають наявності орієнтованих на схеми механізмів контролю даних, які широко застосовуються в реляційному світі - тут вони замінені набагато більш зручними і ефективними видами контролю, зокрема контроль виконується в програмній формі, за допомогою тестів для моделей даних і запитів, а також за допомогою визначення бізнес-правил, заснованих на графі. Зараз така методика вже не викликає сумнівів: розробка за допомогою графових баз даних повністю відповідає сучасним методикам гнучкої і надійної розробки програмного забезпечення, що дозволяє розробці додатків

з використанням графових баз даних не відставати від бізнес-середовища

2.3.3 Відомі приклади застосування

Найкращим місцем для застосування ГБД буде той випадок, де дані, які повинні зберігатися в базі, є асоціативними, тобто коли відношення між двома блоками даних мають велике значення, особливо якщо відношення між елементами має більше значення, ніж параметри самих елементів, чи необхідно зацентувати додаткову увагу на відношенні (додавши йому деякі властивості, наприклад). В цілому, рекомендовано використовувати ГБД якщо ситуації властиві наступні маркери:

- ведеться робота над зв'язаними даними (наявні велика кількість зв'язків)
- обхід даних більш важливий, за індексацію

- наступний шуканий елемент залежить від попереднього
- ведеться робота над різними інтерпретаціями задачі Комівояжера

Із деяких реальних прикладів застосування можна виділити:

- **Pagerank:** Google використовує концепцію графових баз даних при розрахунку порядку відображення результатів пошуку. Для цього, напрумлений граф використовується для з'єднання всесвітніх веб-сторінок разом у вигляді вузлів і гіперпосилань одна з одною у вигляді ребер. Кількість вихідних ребер на графік призначається в якості ваги для ребра. Таким чином, рейтинг сторінки визначається за вагою на одному ребрі в порівнянні з іншими ребрами.
- **Керування даними:** Cisco, одна з провідних світових мережесих організацій, із якої часто беруть приклад інші, прийняла ієрархічну систему управління даними, що базується на графічній утиліті бази даних Neo4j. Це забезпечує їм дуже швидкий доступ до даних в порівнянні з Oracle RAC. Вони також реалізують цю концепцію в ієрархії продуктів, щоб обслуговувати користувачів в режимі реального часу.
- **Соціальний зв'язок:** такі сайти, як Facebook, Twitter, LinkedIn, Viadeo, Glassdoor, зберігають свої зв'язки у вигляді графічних баз даних у вигляді відношень, що є дуже природнім підходом. Рекомендації важливі з точки зору користувачів. Відносини і зв'язки можуть бути дуже добре керовані і доступні в режимі реального часу в порівнянні з реляційними базами даних
- **Керування безпекою та доступом:** у креативній хмарі Adobe використовується структура графової бази даних, яка пов'язує деталі аутентифікації і тим самим надає доступ до контенту як його адміністраторам, так і користувачам.

Це тільки декілька прикладів, натомість в даний момент існує надзвичайно велика кількість рішень, оснований на теорії графів.

2.4 Приклад опису мікросервісної архітектури на основі графової моделі

Приклад графу знань чудово демонструє перспективи опису практично будь чого на основі графа. Звісно тут можна доволі легко прийти до ідеї опису мікросервісних архітектур із використанням графів. Співставивши ідеї розподілених архітектур до ідеї графових баз даних, можна знайти багато спільного. Одразу запрошується співставлення елементів розподіленої системи (наприклад, окремих мікросервісів) до вузлів графа, зв'язки між вузлами комфортно використовувати для опису міжсервісної комунікації (до того ж враховуючи опцію напрямленості зв'язків). Щоб краще зрозуміти, як це може бути використано, необхідно розглянути приклад.

Нехай є невеликий додаток для складання тестів із математики, побудований на ідеї розподіленої архітектури. Він вимагає від користувача зареєструватися та пропонує можливість зареєструватися на здачу тесту, здати тест онлайн, обговорити результати тестів на внутрішньому форумі. Всього система складається із 5-ти модулів та 4 індивідуальних БД. Їх опис приведено в таблиці 2.1.

Таблиця 2.1 - Складові компоненти системи для складання онлайн-тестів з математики

Номер компонента	Назва компонента	Функції	Взаємодія з іншими компонентами	Доступ до бази даних
1	Центральний сервіс	Взаємодія з користувачем; делегатія виконання основних задач спеціалізованим модулям	2, 3, 4, 5, 6	-

Продовження таблиці 2.1

Номер компонента	Назва компонента	Функції	Взаємодія з іншими компонентами	Доступ до бази даних
2	Сервіс реєстрації на здачу онлайн-тесту	Забезпечення механізму реєстрації на здачу тесту	1	База даних запланованих іспитів
3	Сервіс авторизації в системі	Авторизація та реєстрація користувачів	1, 6	База даних персональних даних користувача
4	Сервіс складання іспиту	Забезпечення механізмів онлайн-здачі тесту	1	База даних наявних тестів
5	Сервіс внутрішнього чату	Забезпечення механізму внутрішнього спілкування користувачів	1, 3, 6	База даних повідомлень
6	Сервіс роботи з результатами	Зберігання та обробка результатів тестів, статистика	1, 5	База даних для зберігання результатів

Враховуючи логіку взаємодії компонентів між собою, можна доволі інтуїтивно описати систему у вигляді графа (рис.2.3)

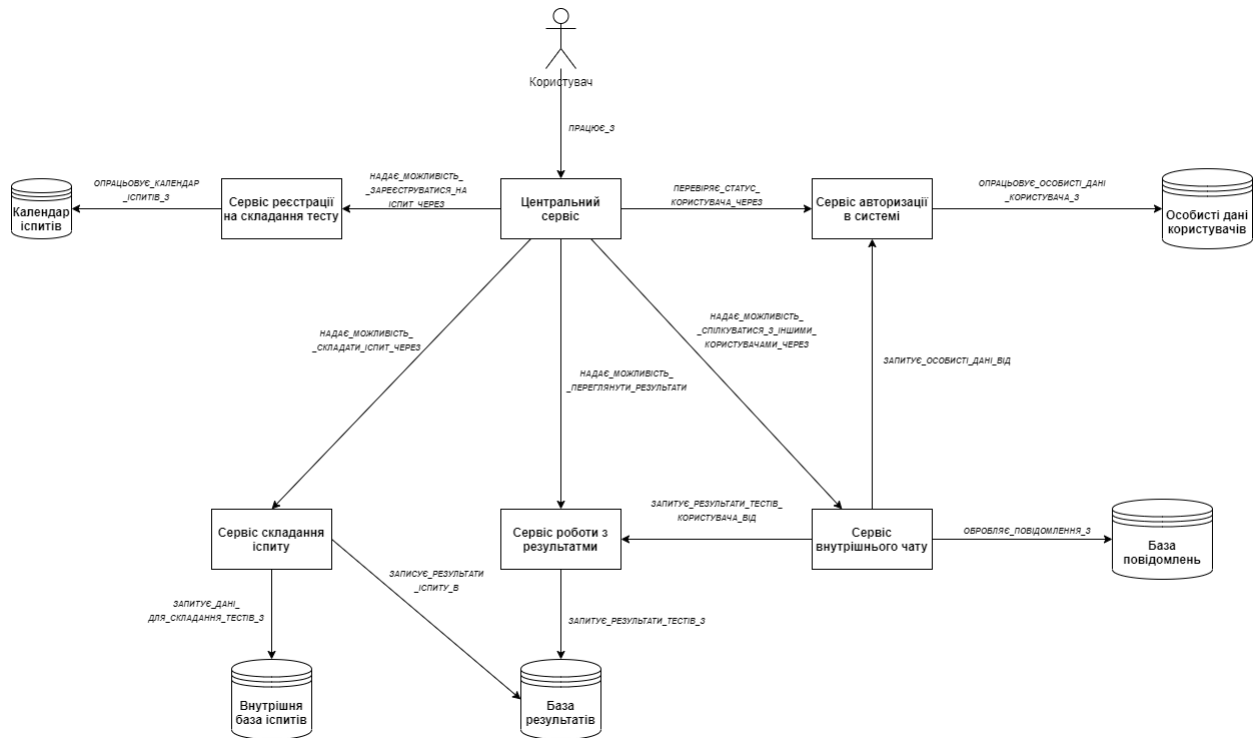


Рисунок 2.3 – Загальний опис мікросервісної архітектури на основі графа

Як видно з рисунка вище, в результаті отримується своєрідний граф знань. Типи зв'язків, разом із коротким описом вузлів, створюють необхідний для розуміння контекст, та дають загальну картину того, що відбувається в системі.

На цьому потенціал використання теорії графів не закінчується. Значно ціннішу інформацію можна отримати, якщо піти далі, і описати систему більш деталізовано, спустившись на рівень нижче, перейшовши із концептуального рівня ближче до технічного (адже на рисунку 2.3 отримано лише загальний граф системи).

Припустимо, кожен сервіс спроектований на основі ООП (тобто його функціонал розбитий на тематичні класи, що виконують певні під функції). В такому випадку можна отримати більше технічної інформації про взаємодію компонентів, відобразивши деталі про наявні класи на графі. На рисунку 2.4 приведено приклад більш детального опису взаємодії компонентів (приклад

обмежується декількома компонентами, щоб уникнути непотрібного перевантаження рисунка).

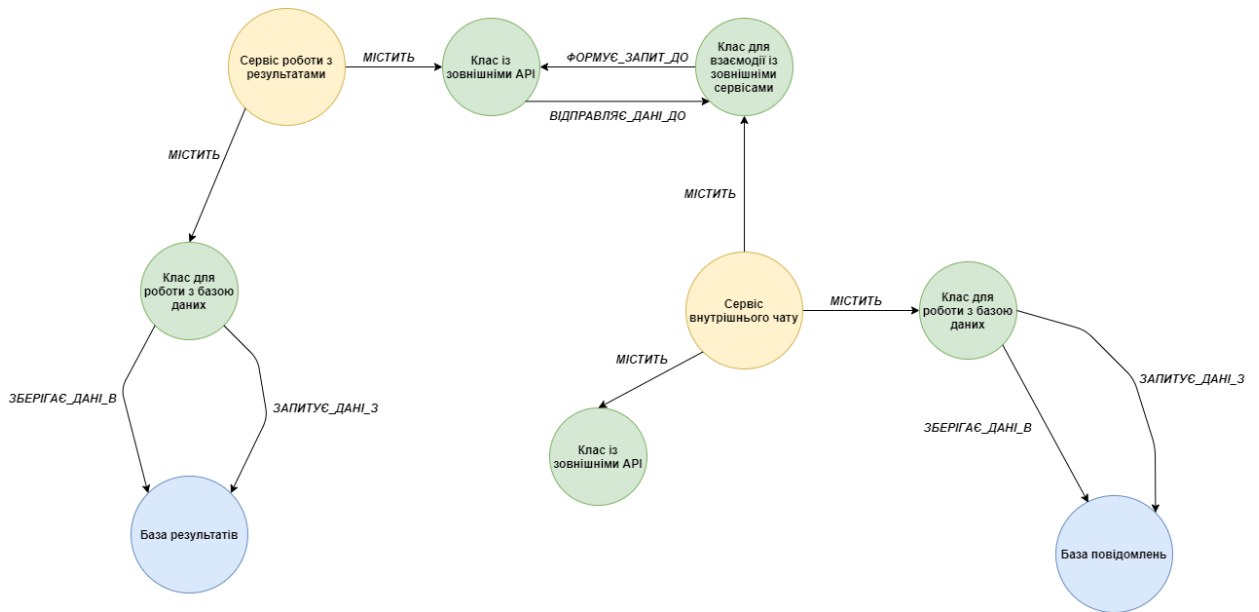


Рисунок 2.4 – Опис системи мікросервісів (частковий), що містить інформацію про класи

Для кращого сприйняття, вузли пофарбовані у кольори згідно за своїм призначенням (жовтий – вузол сервісу, зелений – вузол класу, синій – вузол джерела даних).

Звичайно, можна продовжити деталізацію. Зокрема, код у класах повинен бути розбитий на методи та змінні (згідно ООП парадигми), і ця інформація також може бути відображена у графі. Приклад подальшої деталізації приведено на рисунку 2.5.

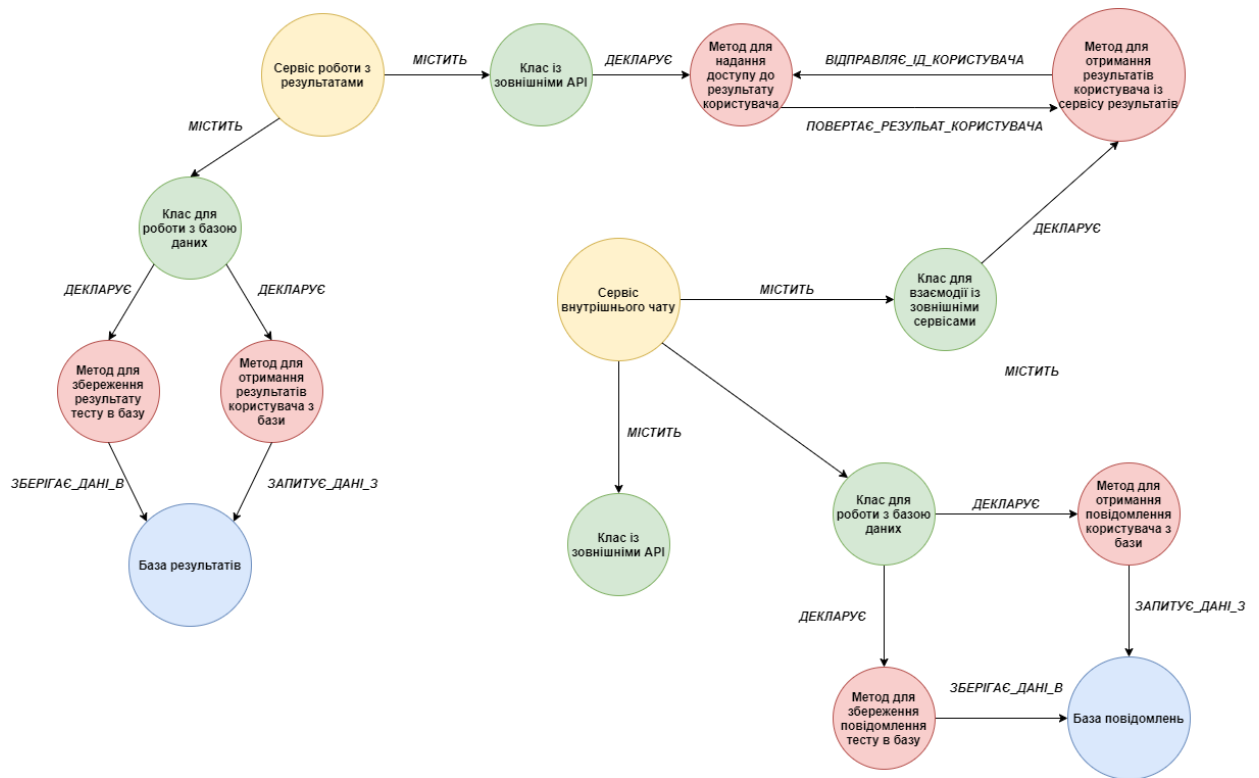


Рисунок 2.5 – Опис системи мікросервісів (частковий), що містить інформацію про методи

Порівняно з попереднім прикладом, тут з'явився ще один тип вузлів – методи. Це дозволяє більш детально розглянути систему, уточнити взаємозв'язки між вузлами. Тут варто додатково уточнити, що позначення вузлів типами є не обов'язковим, адже графова модель не строго типізована. Це робиться просто для логічного об'єднання вузлів. Зрештою варто зауважити, що подібні схеми легко відтворюються практично у будь-якій сучасній графовій базі даних. Таким чином, подібними результатами можна оперувати не тільки в якості схем, проте й у якості даних певної бази, що відкриває нам додаткові можливості із обробки таких даних.

Наприклад, граф із рисунка 2.5 із легкістю може бути відтворений за допомогою серії команд додавання у найпопулярнішій графовій базі сучасності Neo4j. Графічне представлення цих даних у базі Neo4j приведено на рисунку 2.6.

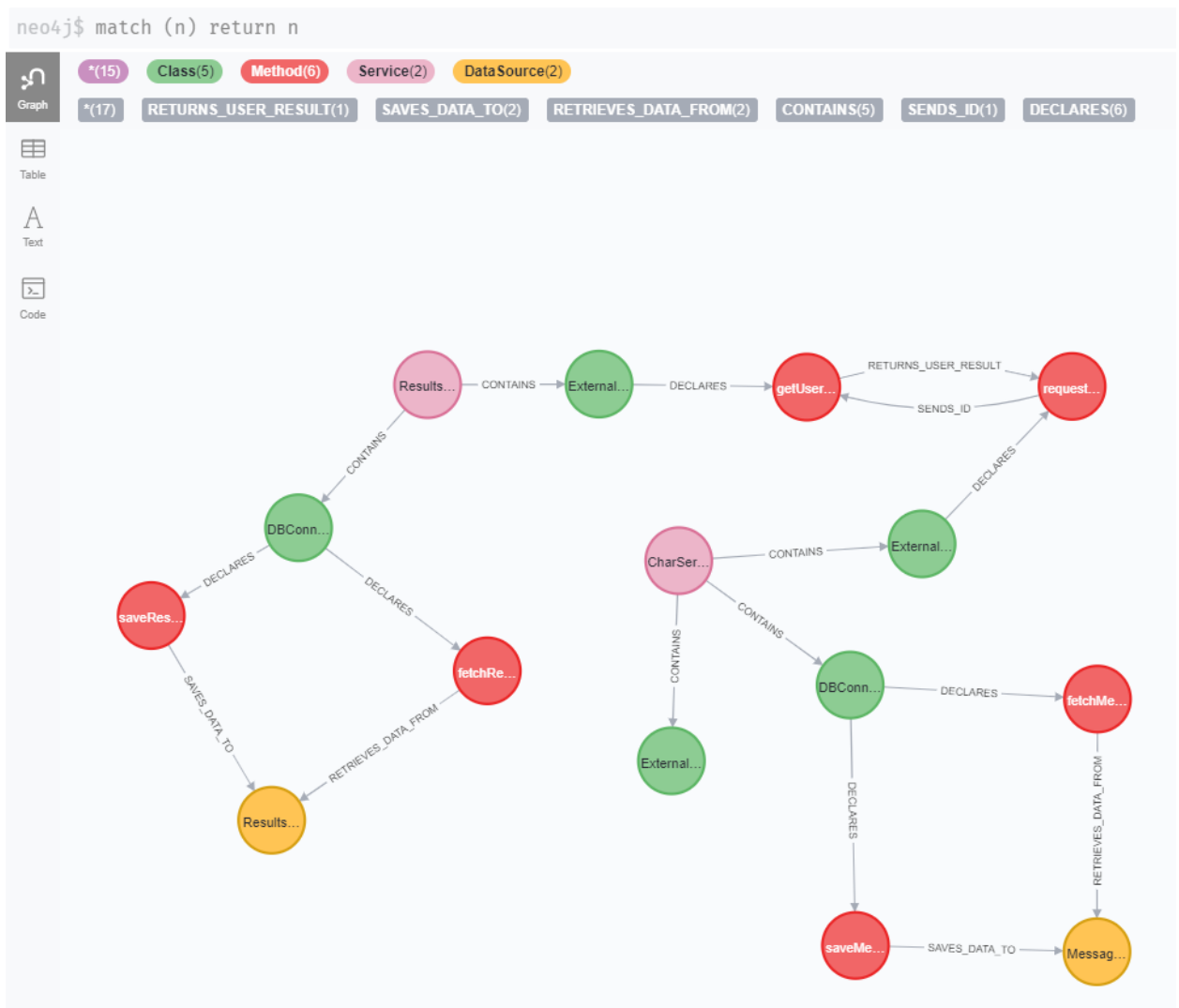


Рисунок 2.6 – Опис системи мікросервісів (частковий) у графовій БД Neo4j

Задля зручності, схожі за роллю вузли позначені відповідними мітками (внутрішній механізм ГБД Neo4j), наприклад «Class», що означає, що такий вузол описує клас. Відповідно до мітки, вузол автоматично зафарбовується певним кольором. Отже, отриманими даними можна ефективно маніпулювати, адже схема, зображена на рисунку 2.6, є всього лиш графічним відображенням даних, що зберігаються в графовій СБД. А раз це система баз даних, то вона надає різноманітні механізми керування даними. Врахувавши той факт, що продовжуючи подальшу деталізацію системи ми отримаємо більше різноманітних вузлів, а система баз даних Neo4j (або,

зрештою, й інші популярні рішення) помимо потужних інструментів обробки даних дає можливість детально описувати окремі сутності чи зв'язки (наприклад, додавати їм майже скільки завгодно властивостей типу «ключ-значення»), можна констатувати, що таке рішення створює широкий простір для дослідницької діяльності, дозволяє детально подивитися на всю систему зовсім під іншим кутом. Звичайно, в першу чергу це дозволить краще зрозуміти систему, так би мовити, побачити її. Однак дане рішення буде особливо цінним у питанні вирішення надзвичайно важливої проблеми у сфері мікросервісних та розподілених архітектур – відслідковування, аналіз та керування залежностями.

Звичайно, схема на рисунку 2.6 - це тільки приклад. Насправді, системи є набагато більшими, містять сотні компонентів і ще більше зв'язків між ними, тому вручну будувати такий опис часто є або зовсім неможливим (адже через людський фактор система не буде точно відтворена), або нераціональним з точки зору витраченого часу. Отже постає наступне завдання: створити додаток, який зможе ефективно аналізувати мікросервісну систему, та на основі цього аналізу автоматично створювати детальну графову схему, яка надасть інформацію про систему в цілому й особливо допоможе у виявленні зв'язків між компонентами, які потребують уваги.

2.5 Висновки

Теорія графів – надзвичайно потужний інструмент, який може бути застосований практично в будь-якій сфері аналізу. Коли ж мова йде про мікросервісні архітектури, різновидності розподілених систем, зв'язок із графами стає інтуїтивним. Опис таких систем на основі графів є природнім та максимально зрозумілим для людини, що спрощує сприйняття та відкриває дорогу для аналізу та спостережень. Наприклад, документні чи реляційні рішення тут підходять не найкращим чином. Враховуючи наявність на ринку доволі розвинутих засобів зберігання та

обробки даних на основі графів, можна встановити, що, по суті, на даний момент існують чудові можливості технічно інтегрувати потужності теорії графів у сучасні системи, що відкриває потенційні можливості для різноманітних маніпуляцій, зокрема не тільки для оперування основними даними додатків, проте й для більш далекоглядних цілей – аналізу, збору метрик та статистики.

3. РОЗДІЛ 3: ПРАКТИЧНА РЕАЛІЗАЦІЯ ДОДАТКУ ДЛЯ ОПИСУ ЕКОСИСТЕМИ МІКРОСЕРВІСІВ ІЗ ВИКОРИСТАННЯМ ГРАФОВОЇ БАЗИ ДАНИХ

3.1 Вимоги до тестового додатку для опису екосистеми мікросервісів

Як і на кожен програмний продукт, на подібний додаток також будуть накладатися певні обмеження у зв'язку із метою його використання. Подібні вимоги стають свого роду критеріями, які повинні забезпечити ефективне виконання безпосереднього завдання такої підсистеми – допомога у запобіганні утворення критичних проблем із залежностями, вчасна діагностика та виявлення проблемних зон. Недостатньо просто виконати аналіз та сформуванню певний текстовий вивід, адже, як було сказано вище, дана характеристики системи є доволі складною, комплексною, тому і результат роботи програми має давати комплексний результат, що відкриє поле для змістовного та глибокого аналізу.

3.1.1 Формування зрозумілої картини системи

Перше завдання, яке ставиться відповідно до узгодженого завдання (аналіз залежностей) – це побачити залежності. Програма повинна виступати скоріше у ролі своєрідного помічника в плані аналізу, а ніж у плані автоматичного вирішення проблеми. Враховуючи той факт, що дана проблема погано представляється у вигляді добре структурованих даних, що по відношенні до неї не надто ефективно використовувати традиційні підходи, як то наприклад вивід даних у вигляді різноманітних таблиць, необхідно більше приділяти уваги засобам представлення знань у вигляді більше зрозумілого в першу чергу для людини. Тобто, користувач повинен отримати зрозумілу картину, в якій легко буде побачити певні аномалії чи проблеми, і отримавши ці цінні дані, він зможе вжити ефективних засобів для ліквідації потенційних або вже наявних критичних моментів.

Враховуючи природу розподілених архітектур, а також очевидну необхідність у візуалізації даних, природньо буде звернути увагу на рішення, основані на теорії графів. Мікросервісна архітектура потенційно добре описується в області вузлів та зв'язків між ними, а оскільки будь-які дані можна представити у вигляді графу, цілком справедливо буде сподіватися, що таке рішення забезпечить необхідний рівень зрозумілості.

3.1.2 Можливість ефективного оперування отриманими даними

Отримавши чітку картину системи, розробнику, тим не менш, необхідно мати можливість ефективно опрацювати отримані дані, адже завжди існує необхідність зацентувати увагу на певному факторі, на певній області системи, чи проекспериментувати із даними щоб отримати більше інформації.

В класичних рішеннях, як то реляційні бази даних, оперування з даними відбувається на основі структурованих запитів, що є доволі ефективним. Однак вище вже зазначалося, що реляційні рішення в даному випадку будуть не найкращим інструментом, так як в контексті поставленого завдання дані не добре структуризовані. Тим не менше, поточне завдання добре узгоджується із запропонованим вище компромісом – графовими базами даних. Зокрема, наприклад рішення Neo4j із її спеціальною декларативною мовою запитів Cypher не поступається в ефективності SQL, тому оперування даними не повинно стати проблемою. Таким чином, отримавши графове представлення системи, яке, ймовірно, буде доволі насиченим великою кількістю вузлів та зв'язків між ними можна доволі просто, виконавши серію запитів, наприклад, виокремити певні сервіси, відкинувши інші, чи зацентувати увагу та тому, де використовується конкретно взятий сервіс. Таким чином, аналіз переходить із області загального погляду до більш концентрованого підходу. Таке рішення дає гнучкість в роботі.

3.1.3 Масштабованість

Безумовно, говорячи про опис мікросервісних систем, не можна пройти мимо такої важливої для розподілених систем теми, як масштабованість. Справа в тому, що взагалі вся мікросервісна ідеологія побудована на меті позбавлення від труднощів при масштабізації, адже, як відомо, сучасні системи активно ростуть, кожна система статистично піддається набагато більшій кількості змін, ніж раніше, тож і кількість сервісів невпинно буде рости, і це треба враховувати. Система росте, а це означає, що невпинно росте і кількість сервісів, з яких вона складається. У багатьох сучасних системах кількість складових мікросервісів йде на сотні. При цьому, системи активно насичені зовнішніми залежностями. Таким чином, варто розуміти, що потенційно опис такої системи буде доволі трудомістким і вимагатиме великої кількості ресурсів. Потрібно передбачити необхідні технічні рішення, що дозволять додатку бути застосовуваним до систем якомога більшого розміру. На щастя, графові бази даних в цьому аспекті дають велику надію, адже їхня природня узгодженість із розподіленими архітектурами спрощує вимоги до ресурсів, які для такого ж завдання в контексті, наприклад, реляційних рішень, можуть бути в рази більшими. Зокрема, варто в першу чергу прийняти до уваги поняття відношень, що йде поряд із поняттям «залежність» в мікросервісній системі. Графові бази даних дозволяють легко і без особливих затрат описувати зв'язки, в той час як інші рішення мають із цим серйозні проблеми.

3.1.4 Простота використання

Зрештою, не варто забувати, що цільовий додаток – всього лиш допоміжний інструмент. Основну складність представляє сама система, та аналіз, який необхідно провести для рішення проблеми. Щоб не ускладнювати загальну картину, такий додаток має доволі просто та зрозуміло виконувати своє завдання та не вимагати в свою сторону занадто

багато уваги, інакше зрештою загальна задача перетвориться в дуже трудомісткий процес, що в багатьох випадках (наприклад, у випадку не сильно насичених функціоналом систем) буде нераціональним, а в інших випадках, у яких було б і корисно застосувати таке рішення, відлякне спеціалістів від ідеї проведення такої кампанії.

3.1.5 Продуктивність

Як вже зазначалося, потенційна програма позиціонується як допоміжний інструмент і в загальному циклі вирішення проблем із залежностями чи неправильною архітектурою системи не повинна привертати до себе занадто багато уваги – основна увага відводиться аналізу результатів.

Відповідно час роботи такого механізму не повинен займати занадто багато часу. Необхідно грамотно підібрати інструменти та стратегію таким чином, щоб ресурси використовувалися максимально ефективно, так як потенційно прийдеться сканувати доволі велику кількість файлів, і це повинно бути прийнято до уваги.

3.2 Підбір інструментів для реалізації

Задля спрощення реалізації ідеї, необхідно сконцентрувати сили на певній області систем, бо цього повинно бути достатньо для дослідження. Зокрема, ціллю дослідження будуть мікросервісні архітектури, створені на основі інструментарію мови Java, що на сьогоднішній день є доволі поширеним рішенням. Таким чином потенційно вдасться створити застосунок, який покриватиме в своїй базовій реалізації доволі великий сегмент області.

3.2.1 Графова база даних neo4j

Основу майбутнього додатку, безумовно, складає база даних – місце, в якому повинна акумулюватися вся зібрана інформація. І як було пояснено раніше, раніше – це повинна бути графова база даних.

Neo4j – графова база даних із відкритим вихідним кодом, яка з’явилася на світ в 2003 році завдяки інвестиційному проекту компанії «Neo Technology». З 2007 року технологія є публічно доступною. Neo4j підтримує всі необхідні характеристики бази даних, зокрема забезпечена технічна підтримка ACID (для забезпечення надійної роботи транзакцій), підтримка розбиття на кластери та відновлення після збоїв. Сьогодні це беззаперечний лідер в області графових баз даних.

Компонентами графової бази даних є вузли та ребра. Обидва ці елементи можуть бути доповнені власним набором характеристик (у вигляді пар «ключ-значення»). Схематично, модель такої БД зображена на рисунку 3.1 і дуже схожа із тією, яку було представлено при розборі вимог до застосунку.

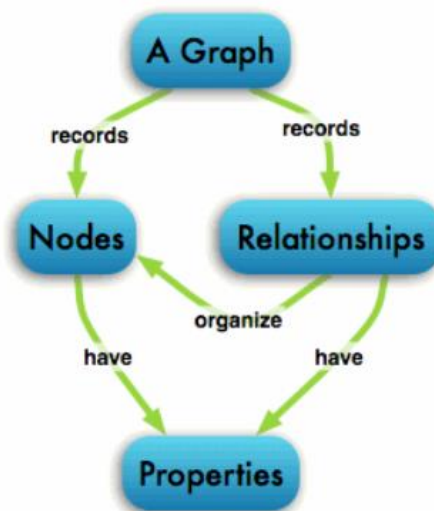


Рисунок 3.1 – Модель графової бази даних Neo4j

3.2.2 Мова запитів Cypher

Cypher - декларативна мова запитів, схожа на SQL, але оптимізована для графів, яка була спочатку розроблена спеціально для Neo4j, однак зараз використовується іншими базами даних, такими як SAP HANA Graph та Redis graph завдяки проекту openCypher.

Деякі основні приклади запитів:

CREATE (n:Service {serviced:5541}) – створити новий вузол типу Service із властивістю serviced, рівною 5541

MATCH (n) RETURN n - *знайти усі вузли*

MATCH (n:Service {serviced:5541}) RETURN n - знайти вузли типу Service із властивістю serviced рівною 5541

CREATE (n{serviceId:5541})-[r:SendsRequest]->(m{serviced:1221}) - *створити на пов'язку два вузли зв'язком типу SendsRequest*

MATCH (a {serviceId: 5541}), (b{serviceId: 1221}) MERGE (a)-[r:SendsRequest]->(b) – *знайти існуючу вузли та пов'язати їх зв'язком*

Синтаксис Cypher забезпечує візуальний та логічний спосіб пошуку патернів вузлів та взаємозв'язків у графі. Вона дозволяє нам визначити, що ми хочемо знайти, вставити, оновити або видалити з наших графічних даних без опису того, як саме це зробити (власне, це і називається декларативною моделлю). За допомогою Cypher користувачі можуть створювати виразні та ефективні запити для обробки необхідних функцій створення, читання, оновлення та видалення.

Cypher - це не тільки найкращий спосіб взаємодії з даними та Neo4j - це ще й відкритий код! Проект openCypher передбачає специфікацію відкритої мови, набір технічної сумісності та довідкову реалізацію аналізатора, планувальника та середовища виконання для Cypher. Він підтримується декількома компаніями у галузі баз даних і дозволяє розробникам баз даних

та клієнтам вільно користуватися, використовувати та сприяти розвитку мови openCypher.

3.2.3 Засоби візуалізації: Neo4j embedded visualizing tool/Neo4j Desktop

Ще однією вагомою перевагою Neo4j у порівнянні із іншими графовими базами даних є потужні вбудовані засоби візуалізації даних. Це дозволяє оперативно, без установки додаткового програмного забезпечення, отримати засіб, що дозволить «побачити» дані. В рамках поточної задачі це, фактично, є необхідною умовою прийому роботи, так як завдання з опису вимагає результат, який дасть чітку картину системи для подальшого аналізу. Не маючи картини системи, буде дуже важко, чи навіть і неможливо отримати деяку інформацію, що цікавить, наприклад не вдасться побачити певні підозрілі зв'язки між компонентами. Засоби візуалізації neo4j дозволяють паралельно із візуалізацією виконувати запити на мові Cypher, що дає прекрасну нагоду динамічно маніпулювати даними для як детального дослідження, динамічно отримувати картину системи з різних ракурсів (а не тільки певне статичне зображення системи). Приклад засобів візуалізації neo4j приведено на рисунку 3.2.

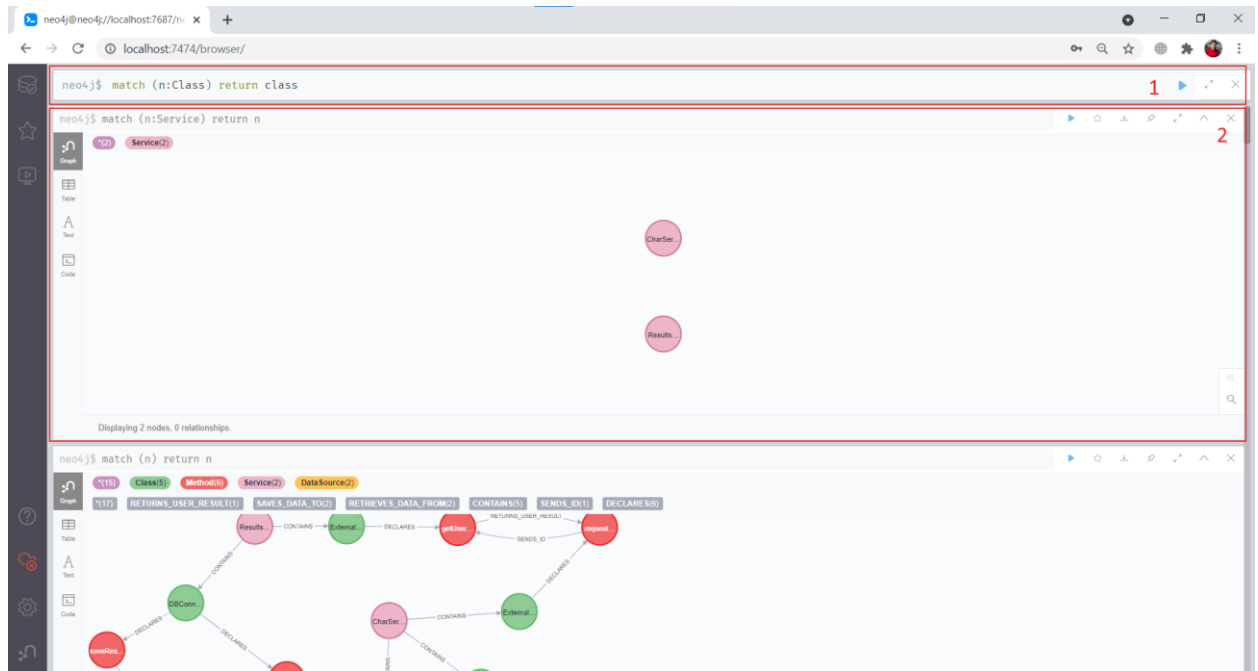


Рисунок 3.2 – Вбудовані засоби візуалізації Neo4j (браузер-версія)

На рисунку 3.1 червоний квадрат із номером «1» позначає область введення команд на мові Cypher для оперування даними, червоний квадрат із номером «2» позначає область виводу, туди виводяться дані про виконання запитів, зокрема інформація про складові графу – кількість та типи вузлів, зв'язків, і звичайно візуальне представлення, із яким можна інтерактивно взаємодіяти.

Версія neo4j Desktop пропонує практично то й же самий інтерфейс.

3.2.4 Мова програмування Java

У якості мови програмування було обрано мову програмування високого рівня Java. Фактично поставлене завдання можна було б виконати і на інших мовах програмування, однак це не було б ефективніше. Перш за все, цей вибір матиме добру сумісність із загальним напрямком роботи (адже мета роботи – дослідити системи, побудовані саме на мові програмування Java), до того ж графова база даних Neo4j сама написана на Java та має зрозумілу документацію саме для інтеграції із цією мовою програмування.

Із загальних переваг Java, які стануть у пригоді під час виконання поставленого завдання, можна виділити потужну базу для роботи із файлами, добре налаштовані механізми багатопотоковості (що, безумовно, є плюсом з точки зору масштабованості), загальну широку функціональність та надійність, надзвичайно широку спільноту, що в результаті дає дуже велику кількість додаткових бібліотек та механізмів, зокрема й бібліотеку Object ASM.

3.2.5 Бібліотека Object ASM

Ядром роботи програми має стати обробка коду системи мікросервісів (а точніше, байт коду), на основі аналізу якого, власне, і буде будуватися графове представлення всієї системи. Маніпуляції із байт-кодом – доволі складний тип операцій, однак завдяки широкій спільноті екосистеми Java, у вільному доступі вже зараз є готові інструменти для таких завдань. Найпопулярнішим на даний момент рішенням є бібліотека Object ASM. Вона пропонує механізми для маніпулювання та аналізу скомпільованого Java-коду, що дозволяє модифікувати вже існуючі класи, або динамічно створювати нові [12]. Ключовою відмінністю даного фреймворка від схожих є акцент на продуктивності, що особливо доречно в контексті даної роботи: для аналізу великих мікросервісних систем прийдеться сканувати сотні класів. Якщо ця операція буде дорогою з точки зору часових ресурсів – одна із вимог до додатку (продуктивність) точно не буде виконана.

Підключити бібліотеку до проекту абсолютно не складно (завдяки інструментарію Java), а сама ідея логіки роботи фреймворку є доволі природньою та базується на моделі «Відвідувача» (англ. «Visitor»). В рамках роботи важливим буде механізм аналізу скомпільованих класів, що дозволить отримувати інформацію про оголошені методи, змінні, використання тих чи інших типів. Бібліотека дозволяє зробити це доволі

легко, а така інформація дозволить отримати чітку картину взаємозв'язків у системі – саме те, що потрібно за умовою задачі.

3.3 Реалізація програми

3.3.1 Модель програми

Перш за все варто визначити, які дані подаватимуться на вхід програми, на основі чого буде йти побудова. Врахувавши той факт, що робота концентрується над розподіленими архітектурами, написаними на Java, найбільш доцільним видається дослідження так званих JAR-пакетів. Саме в такі пакети збираються класи програми та інші необхідні файли для подальшого розгортання на серверах. Тобто фактично, JAR-пакет представляє собою мікросервіс. Аналіз JAR-файлу фактично представляє собою аналіз конкретного мікросервісу. Зваживши на це, доцільно буде подавати на вхід програми набір JAR файлів, кожен із яких представлятиме мікросервіс. Так вдасться проаналізувати всю систему.

За допомогою бібліотеки Object ASM вдасться просканувати файл по всіх його ключовим показникам: сам клас, його методи, поля, анотації, атрибути. Така інформація у сукупності про кожен клас буде поступово створювати загальну картину системи.

Життєвий цикл програми приведено на рисунку 3.3.

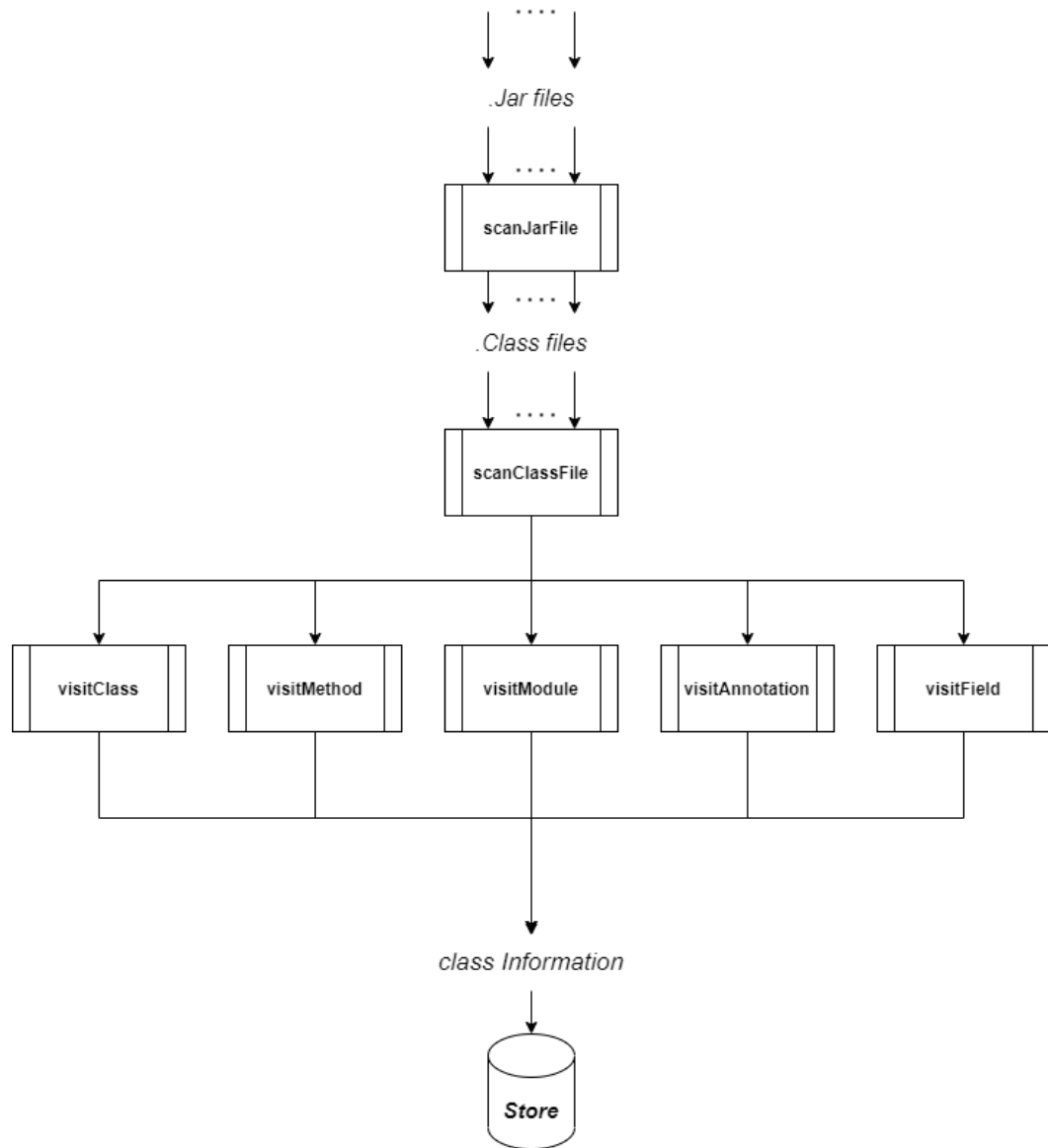


Рисунок 3.3 – Схематичне зображення системи аналізу

3.3.2 Ключові фрагменти коду програми

Виділимо деякі значущі елементи кодової бази, на основі яких буде вибудовано систему аналізу. Весь код програми можна знайти в додатку 1.

3.3.2.1 Доменна модель

Перш за все необхідно відштовхуватись від того, які дані хочеться отримати. Для цього варто описати класи доменної моделі, які будуть відповідати сутностям фінальної моделі, та виступатимуть своєрідними

будівельними блоками для поступової побудови загальної картини.

В лістингу 3.1 приведено код, що описує сутність «Клас».

```

@Data
@NoArgsConstructor
@Entity(label = "Class")
public class ClassFileNode implements Node {
    @Id
    @GeneratedValue
    private Long id;

    @Property
    private String simpleName;
    @Property
    private String fullName;

    @Relationship(type = "USES")
    private Set<ClassFileNode> usedClasses = new HashSet<>();

    @Relationship(type = "USES")
    private Set<PublicMethodNode> usedPublicFields = new HashSet<>();

    @Relationship(type = "EXTENDS")
    private ClassFileNode extendedClass;

    @Relationship(type = "IMPLEMENTS")
    private Set<InterfaceNode> implementedInterfaces = new HashSet<>();

    @Relationship(type = "DECLARES")
    private Set<PublicMethodNode> declaredPublicMethods = new
HashSet<>();

    @Relationship(type = "DECLARES")
    private Set<PublicFieldNode> declaredPublicFields = new
HashSet<>();

    @Relationship(type = "DECLARES")
    private Set<EndpointNode> declaredEndpoints = new HashSet<>();

    @Override
    public String getUniqueProperty() {
        return fullName;
    }
}

```

Лістинг 3.1 – Доменна модель, що описує клас

В даному випадку використовується підхід об'єктно-реляційного мапінгу, тобто фінальний граф будується не шляхом атомарних запитів до БД, а шляхом внутрішньопрограмного створення та заповнення моделей, і вже їх подальшого зберігання, що робить програму чистішою, більш сприйнятливою та природньою.

В лістингу 3.1 можна побачити зокрема, що клас помічено анотацією `NodeEntity`, що вказує на те, що клас представляє об'єкт доменної моделі. Тут же йому присвоюється мітка (в даному випадку – мітка «Class»). Це необхідно для логічного групування об'єктів за спільним функціоналом. В самому класі анотацією `Property` позначено властивості сутності (набори типу «ключ-значення», які несуть додаткову інформацію про сутність; в даному випадку це коротке та повне ім'я класу), анотація `Relationship` позначає відношення сутності, причому тут же вказується його тип (фактично назва), а також вказується набір сутностей, до яких застосовується це відношення. Також обов'язковим полем кожної сутності є ідентифікатор (`Id`). Це значення автоматично генерується системою. Інші сутності програми описані аналогічно (з поправкою на їх індивідуальні властивості, назви, та зв'язки). Ознайомитися із кодом всіх сутностей можна в додатку А. Тут варто відмітити, що всього програма оперує такими сутностями:

- .jar файл (фактично, сервіс)
- Пакет класів
- Клас
- Інтерфейс
- Публічний метод
- Публічна константа/змінна
- Кінцева точка

Між цими сутностями встановлені інтуїтивно-зрозумілі зв'язки, як наприклад «ДЕКЛАРУЄ» стосовно класу, та його методу, «ВИКЛИКАЄ»

стосовно одного методу, який викликає інший, «МІСТИТЬ» стосовно пакету та класів, що до нього входять, тощо.

3.3.2.2 З'єднання із базою даних

Для успішної роботи програми необхідно забезпечити з'єднання із базою даних Neo4j, та налаштувати методи для зберігання та отримання даних.

Перш за все, необхідно створити файл конфігурації, що міститиме необхідні дані для підключення до попередньо запущеного сервера. Такими даними є адреса хоста, логін та пароль користувача. При використанні даних за замовчуванням, у випадку локальної роботи, достатньо буде додати у файл конфігурації (neo4j.properties) рядок наступного вигляду:

URI=bolt://neo4j:neo4j@localhost:7687

Він визначає єдиний параметр, який, тим не менш, є комплексним та вміщає в собі одразу декілька складових: neo4j (перше входження, до символу двокрапки)– логін, neo4j (друге входження, після двокрапки) – пароль, localhost:7687 —хост бази даних. При підключенні до інших серверів, ці дані необхідно відповідно замінити. Далі, використовуючи цей файл конфігурації, можна проініціалізувати з'єднання. Відповідний код приведено в лістингу 3.2

```
private void init() {
    ConfigurationSource props = new
    ClasspathConfigurationSource(PROPERTIES_FILE_PATH);
    Configuration config = new Configuration.Builder(props).build();

    sessionFactory = new SessionFactory(config, DOMAIN_PACKAGE_NAME);
    currentSession = sessionFactory.openSession();
}
```

Лістинг 3.2 – Стартова ініціалізація з'єднання із базою даних Neo4j

Тут *PROPERTIES_FILE_PATH* – вказує повний шлях до файлу конфігурації (описаний вище), *DOMAIN_PACKAGE_NAME* – повна назва пакету, який містить опис доменних моделей (описані в попередньому розділі).

Головне завдання – створити на базі отриманих даних користувача БД фабрику сесій, через яку можна буде отримувати об’єкти сесії, які, в свою чергу, дозволять виконувати операції збереження\звчитування\видалення.

Власне приклад методу, що зберігає об’єкт сутності приведено в лістингу 3.3.

```
public boolean saveJarFileNode(JarFileNode jarNodeToSave) {
    Collection<JarFileNode> existingNodes =
currentSession.loadAll(JarFileNode.class,
                        new Filter(JAR_FILE_NODE_PROPERTY_NAME,
ComparisonOperator.EQUALS, jarNodeToSave.getName()));

    if (existingNodes != null && !existingNodes.isEmpty()) {
        System.out.println("There are such jars nodes already! (" +
existingNodes.size() + ")");
        existingNodes.forEach(node -> System.out.println("id" +
node.getId() + ":" + node.getName()));
        return false;
    }

    currentSession.save(jarNodeToSave);
    System.out.println("Jar Node (" + jarNodeToSave.getName() + ")
saved!");

    return true;
}
```

Лістинг 3.4 – Метод для збереження вузла, що описує JAR файл

Враховуючи той факт, що тут відбувається глибоке збереження (тобто вузол зберігається разом із його внутрішніми під-вузлами), в роботі достатньо буде проводити тільки операції збереження JAR-файлів, попередньо додавши до них внутрішні вузли в процесі обробки файлів.

Варто також здійснювати додаткову перевірку, чи не знайдено в базі вже вузол із таким іменем, оскільки механізми вставки не гарантують нам відсутність дублікатів – це зворотня сторона широкої гнучкості, яку надає нам графова база даних.

В лістингу 3.5 приведено метод для взяття об'єкта із бази даних на основі певного його параметра (властивості), та його значення.

```
public <T> T findNodeByUniquePropertyValue(Class<T> nodeClass, String
propertyName, String propertyValue) {
    Collection<T> existingNodes = currentSession.loadAll(nodeClass,
        new Filter(propertyName, ComparisonOperator.EQUALS,
propertyValue));

    if (existingNodes == null || existingNodes.isEmpty()) {
        return null;
    } else if (existingNodes.size() > 1) {
        ensureClosing();
        throw new DuplicateFoundException("Duplicate found for Class
Node!", existingNodes.toArray()[0]);
    }

    return ((T) existingNodes.toArray()[0]);
}
```

Лістинг 3.5 – Отримання вузла із бази даних за значенням однієї із його властивостей

Метод «узагальнений», тобто повертає різні типи об'єктів (тип бажаного вузла вказується в якості параметра). Це зроблено для уникнення дублювання коду. Також відбувається перевірка на дублікат, адже пошук йде за унікальним значенням властивості.

3.3.2.3 Сканування JAR-файлів

На найвищому рівні все починається зі сканування списку JAR-файлів, що передаються через параметри на вході. Далі відбувається поступове проходження по всіх рівнях (класи, поля, методи, анотації), що дозволяє отримати необхідну інформацію.

Процес сканування JAR-файлу приведено в лістингу 3.6. В ньому використовуються стандартні засоби Java для роботи з JAR-пакетами

```
private void scanJar(JarFile jarFile) {
    System.err.println("Scanning jar file [" + jarFile.getName() +
        "]);

    JarFileNode jarNode = new JarFileNode();
    jarNode.setName(jarFile.getName());

    Map<String, PackageNode> packageMap = new HashMap<>();
    while (jarFile.entries().hasMoreElements()) {
        JarEntry nextEntry = jarFile.entries().nextElement();
        if (nextEntry.getName().endsWith(".class")) {
            String classPackageName = nextEntry.getName().substring(0,
                nextEntry.getName().lastIndexOf("/") + 1);
            if (!packageMap.containsKey(classPackageName)) {
                PackageNode packageNode = new PackageNode();
                packageNode.setName(classPackageName);
                packageMap.put(classPackageName, new PackageNode());
            }
            try {
                InputStream classFileIS =
                    jarFile.getInputStream(nextEntry);
                ClassFileNode classNode = new ClassFileNode();

                ClassReader reader = new ClassReader(classFileIS);
                reader.accept(new ClassVisitor(Opcodes.ASM4,
                    classNode), 0);

                packageMap.get(classPackageName).getClasses().add(classNode);
            } catch (IOException ex) {
                System.out.println("Cannot read class file [" +
                    nextEntry.getName() + "]);
                ex.printStackTrace();
            }
        }
    }

    packageMap.values().forEach(p -> jarNode.getPackages().add(p));
    Neo4jManager.getInstance().saveJarFileNode(jarNode);
}
```

Лістинг 3.6 – Процес сканування JAR-файлу

Структура JarFile містить в собі елементи, що описують внутрішні пакети та класи. Використовуючи їх, можна відкривати дескриптори файлів та зчитувати із них необхідну інформацію. З лістингу 3.6 також видно, що результат формується «дерезовидним» шляхом (а дерево – це всього лиш варіант графа), що є доволі природнім: створюється кореневий вузол jarFile, проходом по інших сутностям цього файлу створюються необхідні вузли пакетів, що в свою чергу містять вузли класів, які генеруються за допомогою спеціального класу, що буде описаний далі. Зрештою, в базу зберігається кореневий вузол, однак разом із ним зберігаються і його внутрішні сутності, що й забезпечує фінальний результат.

3.3.2.4 Сканування класів. Патерн «Відвідувач»

Основна робота виконується саме в класі для сканування класів, що являє собою цільовий об'єкт з точки зору важливості інформації. Зокрема, в лістингу 3.7 приведено приклад частини методу, що збирає поверхнісну інформацію про клас (зокрема, його коротке та повне ім'я, списки інтерфейсів, які той реалізує, ім'я суперкласу), створюючи таким чином доменний об'єкт класу, який надалі зберігатиметься всередині доменного об'єкту пакету.

```

@Override
public void visit(int version, int access, String name, String
signature, String superName, String[] interfaces) {
    classFileNode.setFullName(name);
    classFileNode.setSimpleName(getSimpleName(name));

    if (!superName.equals(JAVA_LANG_OBJECT)) {
        ClassFileNode parentClassNode =
storeManager.findNodeByUniquePropertyValue(
        ClassFileNode.class, NODE_PROPERTY_FULL_NAME, superName
    );
        classFileNode.setExtendedClass(parentClassNode);
    }

    Arrays.stream(interfaces)
        .map(fullName -> {
            InterfaceNode interfaceNode =
storeManager.findNodeByUniquePropertyValue(
            InterfaceNode.class, NODE_PROPERTY_FULL_NAME,
fullName
            );
            return interfaceNode;
        })
        .forEach(node ->
classFileNode.getImplementedInterfaces().add(node));

    super.visit(version, access, name, signature, superName,
interfaces);
}

```

Лістинг 3.7 – Збір поверхневої інформації про клас

Інша інформація збирається за допомогою делегування роботи підлеглим класам. Тут варто відмітити, що в загальному Object ASM реалізовує патерн «Відвідувач». Це означає, що для обробки певного об'єкту (в нашому випадку це файл класу) створюється певний клас обробника, який визначає різноманітні методи обробки, або, інкше кажучи, методи, які будуть застосовані над об'єктом і теоретично могли б бути визначені в ньому самому.

Так, в класі відвідувача файлів в поточній програмі реалізовано декілька методів для обробки окремих частин файлу: метод для обробки

(відвідування) полей, методів, модулів, тощо. Приклад такого методу приведено в лістингу 3.8.

```

@Override
public FieldVisitor visitField(int access, String name, String
descriptor, String signature, Object value) {
    if ((access & Opcodes.ACC_PUBLIC) > 0) {
        PublicFieldNode publicFieldNode = new PublicFieldNode();
        publicFieldNode.setName(name);

        return new
com.graph.analysis.scan.visitor.FieldVisitor(Opcodes.ASM4, publicFieldNode,
classFileNode);
    }

    return super.visitField(access, name, descriptor, signature,
value);
}

```

Лістинг 3.8 – «Відвідування» полей класу

Тут зокрема відбувається перевірка, чи поле є публічним (адже в контексті аналізу залежностей нам важливі саме публічні поля, які можуть бути використані зовні і таким чином створити залежність між компонентами). Якщо так – операція по збору інформації передається допоміжному класу. Із іншими об'єктами проводяться аналогічні операції. Більше прикладів коду можна знайти в додатку А, тут тільки варто відмітити, що подібні «Відвідувачі» створені для всіх об'єктів доменної моделі окрім JAR файлу та пакету (необхідна інформація про них, як було показано вище, збирається на верхньому рівні).

3.4 Приклади роботи програми

Дослідимо роботу програми, для цього знадобиться взяти за приклад певний додаток, побудований на мікросервісній архітектурі. Для прикладу, із відкритих джерел (публічного репозиторію на платформі

GitHub), оберемо невеликий такий додаток [13] – додаток для контролю метрик особистих накопичень.

На рисунку 3.4 приведено схему системи цього додатку.

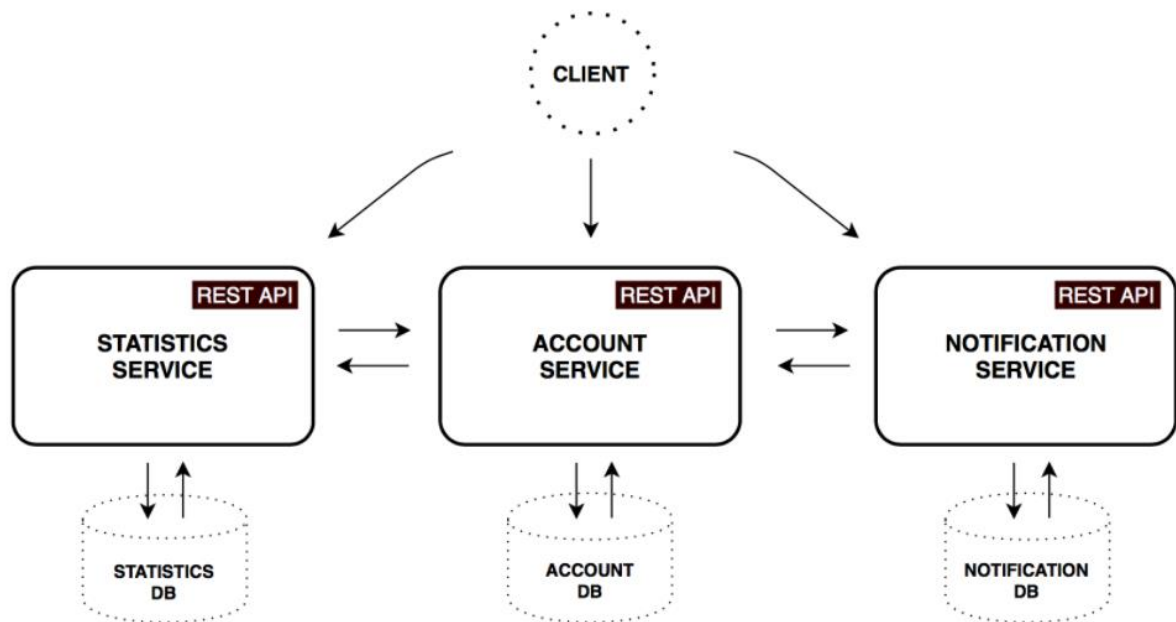


Рисунок 3.4 – Схема мікросервісного додатку для контролю метрик витрат [13]

Як видно з рисунка, в системі є три сервіси, що кумінують між собою, кожен сервіс має своє джерело даних.

Для початку роботи необхідно запуснути локальний сервер бази даних. Для цього необхідно завантажити та установити базу Neo4j до себе на комп'ютер [14], після чого в місці установки послідовно виконати наступні команди:

```
bin\neo4j install-service
```

```
bin\neo4j start
```

Далі, тестовий додаток для аналізу цієї системи, для цього у директорії з проектом виконаємо наступну команду:

```
mvn clean install.
```

Далі запусимо програму на виконання наступною командою:

```
mvn exec:java -Dexec.mainClass="com.graph.analysis.Main" -Dexec.args="--files jars/account-service.jar jars/notification-service.jar jars/statistics-service.jar"
```

Тут в якості параметрів вказуємо прапорець `--files`, що вказує, що за ним йдуть файли, далі послідовно перераховуємо ці файли. Необхідно відмітити, що в даному випадку JAR-файли, що тестувалися, були поміщені у внутрішню директорію проекту із назвою `jars`. Інакше необхідно було б вказувати повний шлях до файлів.

Консоль після цього матиме наступний вигляд (рис.3.5)

```
D:\Proga\diploma_project>mvn exec:java -Dexec.mainClass="com.graph.analysis.Main" -Dexec.args="--files jars/notification-service.jar jars/account-service.jar jars/statistics-service.jar"
[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.kpi.diploma:microservice-graph-inspector >-----
[INFO] Building microservice-graph-inspector 1.0-SNAPSHOT
[INFO] -----[ jar ]-----
[INFO]
[INFO] --- exec-maven-plugin:3.0.0:java (default-cli) @ microservice-graph-inspector ---
Program has run!
-----Scanning jar file [jars\notification-service.jar]-----
Finished scanning the file: com/piggymetrics/notification/domain/Frequency.class
Finished scanning the file: com/piggymetrics/notification/domain/NotificationSettings.class
Finished scanning the file: com/piggymetrics/notification/domain/NotificationType.class
Finished scanning the file: com/piggymetrics/notification/domain/Recipient.class
Finished scanning the file: com/piggymetrics/notification/NotificationServiceApplication$CustomConversionsConfig.class
Finished scanning the file: com/piggymetrics/notification/NotificationServiceApplication.class
Finished scanning the file: com/piggymetrics/notification/client/AccountServiceClient.class
Finished scanning the file: com/piggymetrics/notification/repository/converter/FrequencyWriterConverter.class
Finished scanning the file: com/piggymetrics/notification/controller/RecipientController.class
Finished scanning the file: com/piggymetrics/notification/repository/converter/FrequencyReaderConverter.class
Finished scanning the file: com/piggymetrics/notification/repository/RecipientRepository.class
Finished scanning the file: com/piggymetrics/notification/service/EmailService.class
Finished scanning the file: com/piggymetrics/notification/service/EmailServiceImpl.class
Finished scanning the file: com/piggymetrics/notification/service/NotificationService.class
Finished scanning the file: com/piggymetrics/notification/service/NotificationServiceImpl.class
Finished scanning the file: com/piggymetrics/notification/service/RecipientService.class
Finished scanning the file: com/piggymetrics/notification/service/RecipientServiceImpl$1.class
Finished scanning the file: com/piggymetrics/notification/service/RecipientServiceImpl.class
Finished scanning the file: com/piggymetrics/notification/config/ResourceServerConfig.class
-----Scanning jar file [jars\account-service.jar]-----
Finished scanning the file: com/piggymetrics/account/client/StatisticsServiceClientFallback.class
Finished scanning the file: com/piggymetrics/account/AccountApplication.class
Finished scanning the file: com/piggymetrics/account/client/AuthServiceClient.class
Finished scanning the file: com/piggymetrics/account/config/ResourceServerConfig.class
Finished scanning the file: com/piggymetrics/account/domain/Account.class
Finished scanning the file: com/piggymetrics/account/domain/Currency.class
Finished scanning the file: com/piggymetrics/account/domain/Item.class
Finished scanning the file: com/piggymetrics/account/domain/Saving.class
Finished scanning the file: com/piggymetrics/account/domain/TimePeriod.class
Finished scanning the file: com/piggymetrics/account/domain/User.class
Finished scanning the file: com/piggymetrics/account/repository/AccountRepository.class
Finished scanning the file: com/piggymetrics/account/service/AccountService.class
Finished scanning the file: com/piggymetrics/account/service/AccountServiceImpl.class
```

Рисунок 3.5 – Процес сканування мікросервісів

Як видно, програма поступово виводить всі архіви та відповідні їм класи, що пройшли сканування. Вкінці виводу з'являється повідомлення про

завершення процесу та пропозиція перейти за посиланням, щоб працювати із побудованим графом (рис.3.6).

```

Finished scanning the file: com/piggymetrics/statistics/client/ExchangeRatesClient.class
Finished scanning the file: com/piggymetrics/statistics/repository/converter/DataPointIdWriterConverter.class
Finished scanning the file: com/piggymetrics/statistics/client/ExchangeRatesClientFallback.class
Finished scanning the file: com/piggymetrics/statistics/repository/DataPointRepository.class
Finished scanning the file: com/piggymetrics/statistics/service/ExchangeRatesService.class
Finished scanning the file: com/piggymetrics/statistics/service/ExchangeRatesServiceImpl.class
Finished scanning the file: com/piggymetrics/statistics/service/security/CustomUserInfoTokenServices.class
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Program finished scanning!
Go to: http://localhost:7474/
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 2.265 s
[INFO] Finished at: 2021-06-08T22:05:32+03:00
[INFO] -----
D:\Proga\diploma_project>

```

Рисунок 3.6 – Закінчення роботи програми

Перейшовши за посиланням, можна побачити вікно із вимогою ввести дані (рис.3.7). Тут необхідно ввести дані для входу, які були вказані у файлі конфігурації.

\$:server connect

Connect to Neo4j

Database access might require an authenticated connection

Connect URL

neo4j:// localhost:7687

Database - leave empty for default

Authentication type

Username / Password

Username

neo4j

Password

Connect

Рисунок 3.7 – Авторизація користувача

Ввівши логін та пароль користувача, можна нарешті побачити справжні результати роботи. Для цього необхідно вибрати всі створені елементи, ввівши наступну

команду

в

полі

команд:

MATCH (N) RETURN n;

Результати виконання команди приведено на рисунку 3.8.

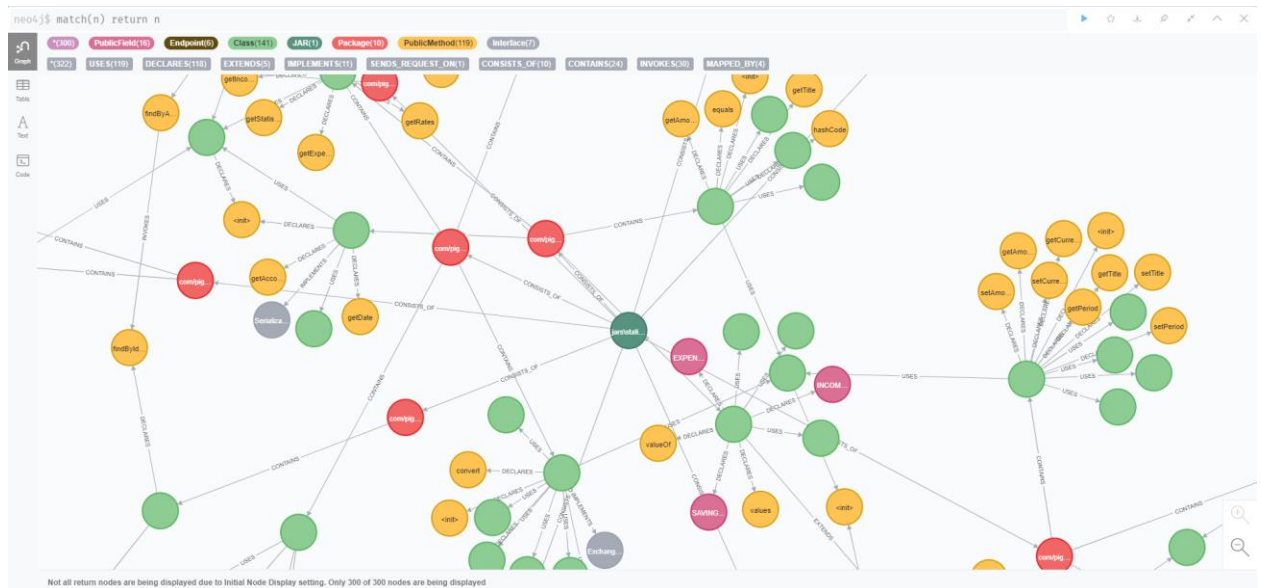


Рисунок 3.8 – Побудований граф системи

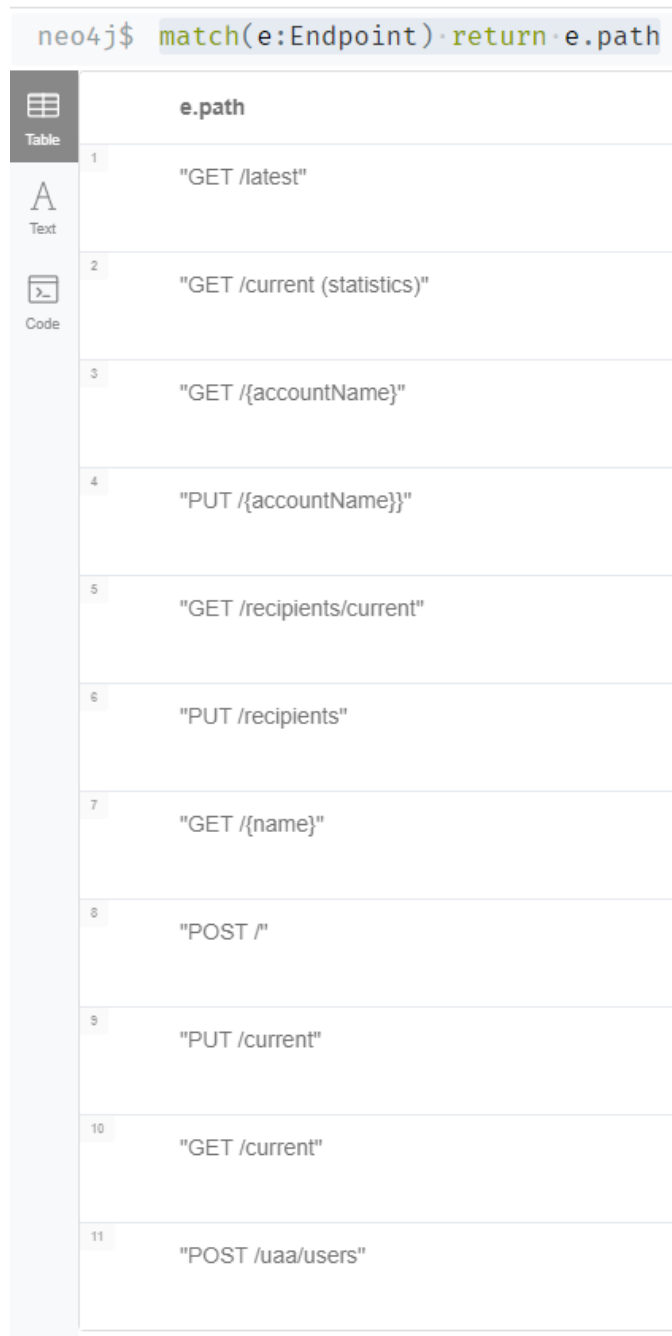
Тут можна побачити кількість вузлів та зв'язків, що відображаються, їх типи (зверху, в лівому куті). В основній області вікна можна побачити, власне, побудований граф – те, чого і хотілось добитися за допомогою створеної програми. Оглянувши граф, можна побачити певні тенденції, наприклад відслідкувати які пакети містить архів, які той визначає класи. Однак значно цікавіше поглянути на взаємодію між класами, зокрема на взаємні виклики методів, залежності між класами. Варто відзначити, засоби візуалізації Neo4j дозволяють одночасно виводити тільки 300 вузлів, тому в основній області відображені не всі. Зрештою, занадто велика кількість вузлів створює занадто велике когнітивне навантаження. Тут найкращим рішенням буде скористатися засобами мови Cytoscape для детального дослідження даних, що нас цікавлять.

3.6 Оперування отриманим графом

Отже, найбільшу вигоду можна отримати, оперуючи графом за допомогою вбудованої мови структурованих запитів. Нище приведено деякі цікаві команди.

3.6.1 Пошук всіх кінцевих точок системи

Для початку можна вибрати список всіх кінцевих точок, наявних у системі, адже це, по суті, головні канали міжсервісної взаємодії. Для цього необхідно виконати наступну команду: `MATCH (n:Endpoint) RETURN n.path`. Результати приведені на рисунку 3.9.



```
neo4j$ match(e:Endpoint)·return·e.path
```

	e.path
1	"GET /latest"
2	"GET /current (statistics)"
3	"GET /{accountName}"
4	"PUT /{accountName}"
5	"GET /recipients/current"
6	"PUT /recipients"
7	"GET /{name}"
8	"POST /"
9	"PUT /current"
10	"GET /current"
11	"POST /uaa/users"

Рисунок 3.9 – Всі знайдені кінцеві точки в системі

Отже всього в системі виявлено 11 кінцевих точок. Підемо далі.

3.6.2 Виявлення місця декларації кінцевих точок

Можна отримати дещо більше інформації встановивши, де саме визначені знайдені кінцеві точки (рис. 3.10).

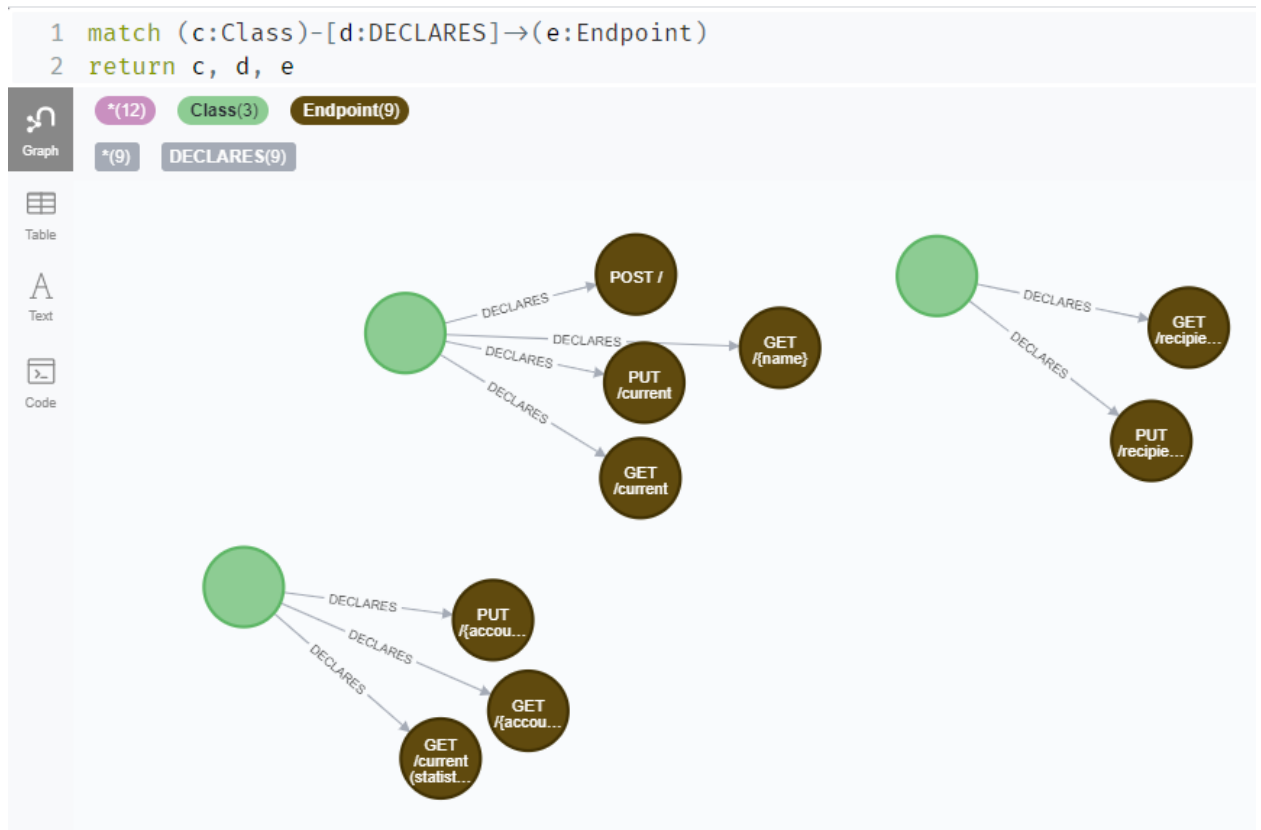


Рисунок 3.10 – Визначення місць (класів) декларування кінцевих точок

Навівши курсор на певний вузол, можна побачити додаткову інформацію про нього, зокрема його атрибути. Так, навівши курсор на вузол, що декларує 4-ри кінцеві точки, можна визначити, що це за клас (рис.3.11)

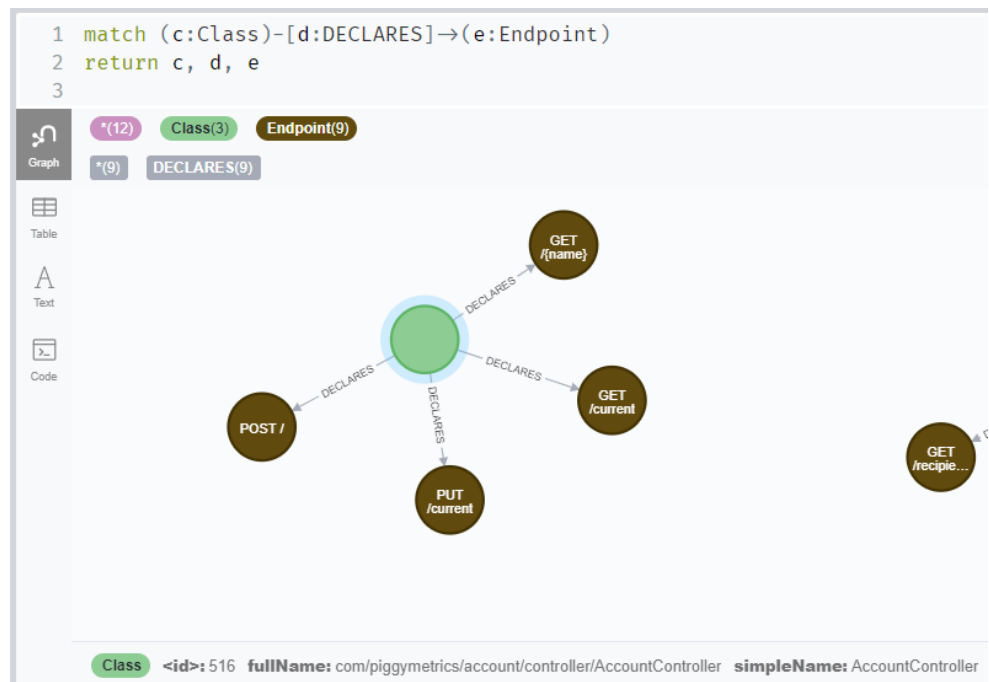


Рисунок 3.11 – Додаткові атрибути вузла

Оскільки тестування проводилося на скомпільованих пакетах проекту із відкритим вихідним кодом, є можливість заглянути в сам код та переконатися, що там справді заделкаровано 4-ри кінцеві точки. Відкривши відповідний клас в редакторі, можна побачити наступний код (рис.3.12):

```

1 package com.piggymetrics.account.controller;
2
3 import ...
4
5 @RestController
6 public class AccountController {
7
8     @Autowired
9     private AccountService accountService;
10
11     @PreAuthorize("#oauth2.hasScope('server') or #name.equals('demo')")
12     @RequestMapping(path = "/{name}", method = RequestMethod.GET)
13     public Account getAccountByName(@PathVariable String name) { return accountService.findByName(name); }
14
15     @RequestMapping(path = "/current", method = RequestMethod.GET)
16     public Account getCurrentAccount(Principal principal) {
17         return accountService.findByName(principal.getName());
18     }
19
20     @RequestMapping(path = "/current", method = RequestMethod.PUT)
21     public void saveCurrentAccount(Principal principal, @Valid @RequestBody Account account) {
22         accountService.saveChanges(principal.getName(), account);
23     }
24
25     @RequestMapping(path = "/", method = RequestMethod.POST)
26     public Account createNewAccount(@Valid @RequestBody User user) { return accountService.create(user); }
27 }

```

Рисунок 3.12 – Код класу контролера

Справді, в знайденому класі 4-ри точки доступу.

3.6.3 Виявлення залежностей між мікросервісами

Мова Cypher запитів дозволяє легко знаходити залежності (ц це звичайний тип зв'язків) між вузлами у графі. Так, виконавши нескладну для розуміння команду, можна виявити залежність між мікросервісами (рис.3.13).



Рисунок 3.13 – Залежні мікросервіси

Для кращого розуміння, можна вивести всі вузли (рис.3.14)

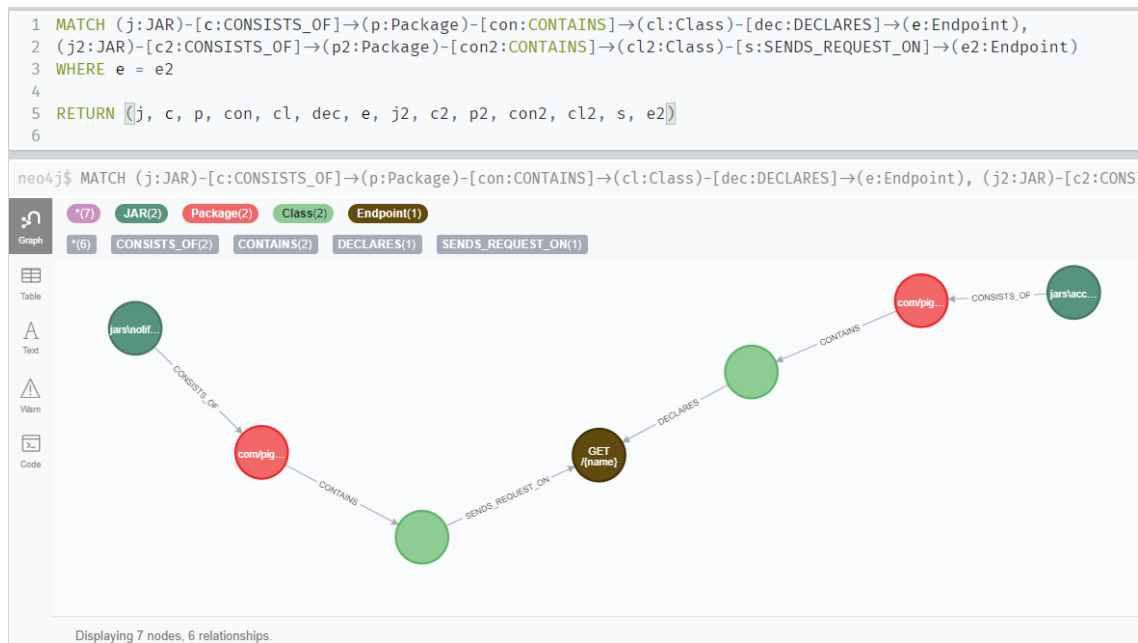
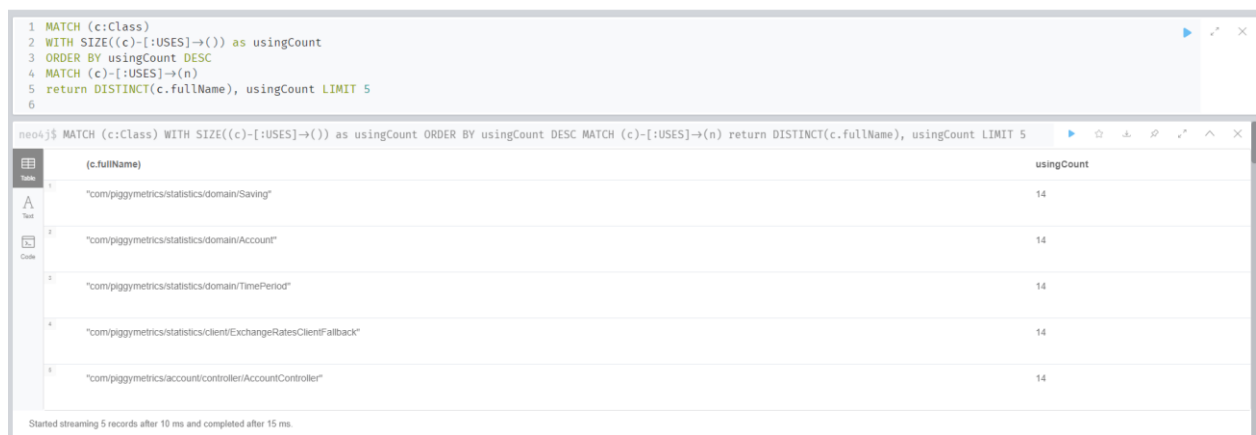


Рисунок 3.15 – Візуальне відображення залежностей між мікросервісами

Схожий патерн можна самостійно відшукати в масиві вузлів графа, а можна вибирати вручну.

3.6.4 Виявлення завантажених точок

Зрештою, над масивом вузлів можна проводити дуже велику кількість операцій, поле для досліджень широке. Зокрема, на графі можна візуально відстежувати вузли, насичені зв'язками. Проте оскільки область візуального виводу обмежена, краще проводити точні розрахунки. Наприклад, можна знайти вузли, що найбільше від інших використовують інші класи (рис.3.16)



```

1 MATCH (c:Class)
2 WITH SIZE((c)-[:USES]->()) as usingCount
3 ORDER BY usingCount DESC
4 MATCH (c)-[:USES]->(n)
5 return DISTINCT(c.fullName), usingCount LIMIT 5
6

```

	(c.fullName)	usingCount
1	"com.piggymetrics/statistics/domain/Saving"	14
2	"com.piggymetrics/statistics/domain/Account"	14
3	"com.piggymetrics/statistics/domain/TimePeriod"	14
4	"com.piggymetrics/statistics/client/ExchangeRatesClientFallback"	14
5	"com.piggymetrics/account/controller/AccountController"	14

Started streaming 5 records after 10 ms and completed after 15 ms.

Рисунок 3.16 – Вузли із найбільшою кількістю використань зовнішніх класів

В даному випадку, всі п'ять вузлів із найбільшою кількістю зовнішніх використань класів/змінних мають по 14 таких залежностей.

3.7 Висновки

Створений додаток – перший крок в інтеграції теорії графів та інструментів моніторингу систем. Вже на прикладі цього тестового додатку можна переконатися у великому потенціалі такого рішення, адже простота сприйняття та цінність отриманої інформації одразу створюють місце у сучасних системах для подібних рішень. Мікросервісні архітектури дуже добре описуються графами, це інтуїтивно зрозумілий підхід для людини який в той же час відкриває великі можливості і з точки зору комп'ютерних обчислень. Тож найближчі подальші кроки в цьому напрямку - розширення

спектру технологій, які можуть бути піддані подібному аналізу, забезпечення ефективного збору ще більшої потенційно корисної інформації.

4. РОЗДІЛ: ФУНКЦІОНАЛЬНО-ВАРТІСНИЙ АНАЛІЗ ПРОГРАМНОГО ПРОДУКТУ

З метою оцінки реалістичності впровадження розробленого продукту в реальні системи виникає необхідність у проведенні функціонально-вартісного аналізу (ФВА). Необхідно оцінити потенційні витрати, варіанти виробництва щоб по перше – визначити, чи реальними є витрачені ресурси та зусилля на реалізацію продукту (і відповідно чи буде він рентабельним), і по друге – обрати найбільш вигідні стратегії реалізації.

Головними кроками в цьому завданні є визначення етапів розробки, підрахунок потенційних витрат, затраченого часу, підбір профілю та кількості спеціалістів щоб у кінцевому етапі визначити фінальну вартість проекту і на основі отриманого результату оцінювати можливості його впровадження.

4.1 Постановка задачі

Функціонально-вартісний аналіз в даній роботі використовується з метою проведення техніко-економічного аналізу процесу розробки програмного продукту для автоматичного опису мікросервісних архітектур на основі екосистеми технологій мови програмування Java.

Перш за все, на самому початку повинні бути виставлені певні вимоги, котрі розроблена система повинна задовольняти. Це відразу встановить пені рамки та спрямує майбутній додаток по правильному шляху. Отже, до даного продукту ставляться наступні вимоги:

- Побудова детальної та зрозумілої картини (зокрема, представлення у вигляді графа) досліджуваної додатком системи
- Забезпечення можливості ефективного оперування отриманими даними з метою подальшого аналізу
- Масштабованість (можливість обробки великих систем, простота додавання нового функціоналу за необхідності)

- Простота застосування
- Продуктивність (виконання поставленої задачі за прийнятні проміжки часу)

4.2 Визначення функцій потенційного продукту

Отже, головна мета (тобто функція) – створення програмного продукту (додатку) для автоматичного опису мікросервісних систем на основі графової бази даних. Позначимо її як Φ_0 . Тоді можна визначити такі підфункції: Φ_1 – вибір базової технології програмування (мова програмування)

Φ_2 – вибір графової бази даних

Φ_3 – вибір допоміжних фреймворків

Φ_4 – вибір середовища розробки

Кожну із функцій можна виконувати (реалізовувати) по-різному, адже існують щонайменше декілька адекватних варіантів. Зокрема можна виділити наступні:

Функція Φ_1

- 1) Java
- 2) C#
- 3) Python

Функція Φ_2

- 1) Neo4j
- 2) OrientDB
- 3) InfoGrid

Функція Φ_3

- 1) Object ASM (Java)
- 2) Jni4net (C#)
- 3) Python-javatools (Python)

Функція Φ_4

- 1) IntelliJ Idea (Java)
- 2) Eclipse (Java)
- 3) Microsoft Visual Studio (C#)

- 4) Visual Studio Code (C#)
- 5) PyCharm (Python)
- 6) Jupiter Notebook (Python)

Морфологічна карта, приведена на рисунку 4.1, відображає всі можливі варіанти реалізації описаних вище функцій

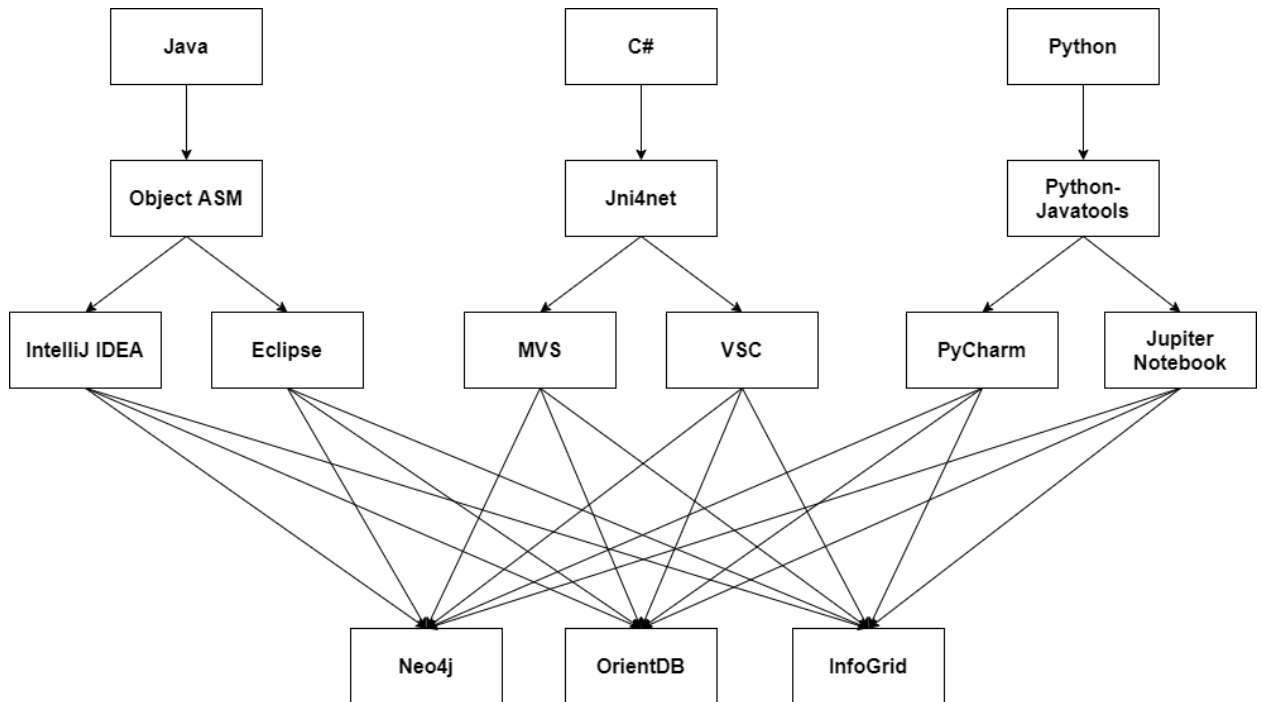


Рисунок 4.1 – Морфологічна карта множин варіантів основних функцій

Розглянемо переваги та недоліки по кожному із варіантів кожної функції. Результати зведено до таблиці 4.1.

Таблиця 4.1 – Переваги та недоліки варіантів реалізації функцій

Позначення функції	Варіант реалізації	Переваги	Недоліки
Φ_1	1	Повна сумісність із технологією дослідження, широкий вибір інструментів, кросплатформеність	Дещо складніша система порівняно із іншими

Продовження таблиці 4.1

Позначення функції	Варіант реалізації	Переваги	Недоліки
Φ ₁	2	Широкий набір інструментів,	Відсутність повної сумісності
	3	Швидкість роботи, простіша технологія	Слабка сумісність, недостатня кількість необхідних бібліотек
Φ ₂	1	Найбільш розповсюджена графова БД, потужний інструментарій, засоби візуалізації	Трохи повільніша за інші варіанти
	2	Потужний інструментарій, підтримка SQL	Менша розповсюдженість
	3	Швидкість роботи	Відносно слабший функціонал
Φ ₃	1	Широкий функціонал для обробки цільових файлів	Доступна тільки для Java
	2	Простота використання	Доступна тільки для C#
	3	Швидкість роботи	Доступна тільки для Python
Φ ₄	1	Потужний інструментарій	Повільніша за інші варіанти
	2	Відкритий доступ	Менший функціонал
	3	Потужний інструментарі	Повільна швидкість роботи

Продовження таблиці 4.1

Позначення функції	Варіант реалізації	Переваги	Недоліки
Φ ₄	4	Висока швидкість, відкритий доступ	Слабкий функціонал, необхідність попереднього налаштування
	5	Потужний функціонал, зручність	Повільна робота
	6	Швидкість, доступність	Слабший функціонал

Аналізуючи таблицю вище, можемо зробити наступні припущення щодо кожної із функцій:

Φ₁: перевага віддається повній сумісності із системою, широкому та потужному функціоналу та бажано, що програма була крос-платформеною. Найкращим варіантом буде варіант 1 (Java)

Φ₂: перевага віддається перевіреним рішенням, із широким інструментарієм та хорошими засобами візуалізації результатів – найкращим є варіант 1 (Neo4J).

Φ₃: у зв'язку із попередніми варіантами коло вибору звужене до того, що підтримує Java. Тоді вибір – варіант 1 (Object ASM).

Φ₄: у зв'язку із попередніми варіантами коло вибору звужене до того, що підтримує Java. Обидва такі варіанти підходять.

Зрештою отримано два вірогідні варіанти:

$$\Phi_{1-1} + \Phi_{2-1} + \Phi_{3-1} + \Phi_{4-1}$$

$$\Phi_{1-1} + \Phi_{2-1} + \Phi_{3-1} + \Phi_{4-2}$$

Для того, щоб мати змогу якісно оцінити визначені функції, необхідно сформулювати систему параметрів.

4.3 Система параметрів програмного продукту

Щоб точно оцінити параметри вибору, необхідно визначити певні технічні параметри (які забезпечать однозначність). Зокрема, можна використати наступні технічні показники:

П1 – швидкодія обраної МП

П2 – використання ресурсів (зокрема використання пам'яті для обчислень)

П3 – час аналізу досліджуваної системи

П4 – потенційний об'єм результуючої кодової бази

Щоб мати розуміння, які значення параметрів вважати задовільними, необхідно описати певні рамки. Опис параметрів, визначених вище, зведено в таблицю 4.2.

Таблиця 4.2 - Основні технічні параметри програмного продукту

Параметр	Позначення	Одиниці виміру	Оцінка показника		
			Незадовільно	Задовільно	Дуже добре
Швидкодія мови програмування (МП)	П1	оп/мс	11000	15000	20000
Обсяг використовуваної пам'яті	П2	Мб	>1000	500	<300
Час аналізу досліджуваної системи	П3	с	>300	120	<60
Потенційний об'єм програмного коду	П4	КБ	>1000	500	<300

Враховавши дані із таблиці 4.2, можна побудувати графічні характеристики кожного з параметрів.

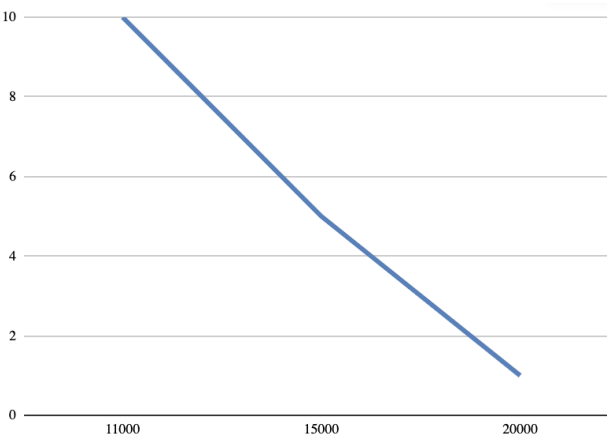


Рисунок 4.2 – Грфічна характеристика параметра П1 (Швидкодія МП)

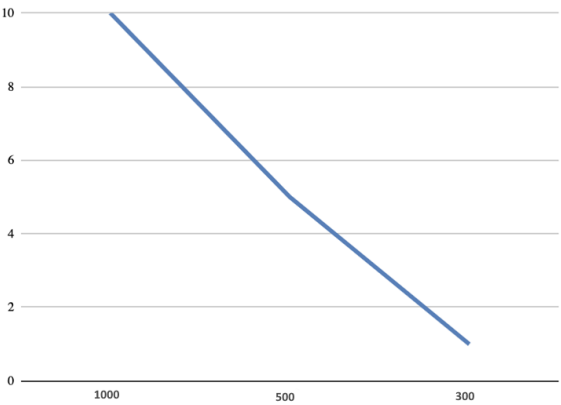


Рисунок 4.3 – Графічна характеристика параметра П2 (обсяг використовуваної пам'яті)

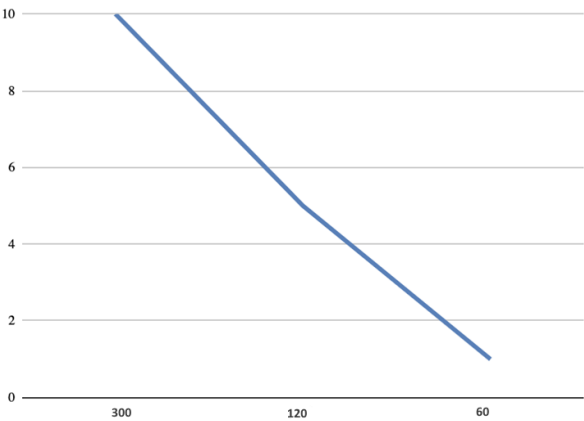


Рисунок 4.4 – Графічна характеристика параметра П3 (час аналізу досліджуваної системи)

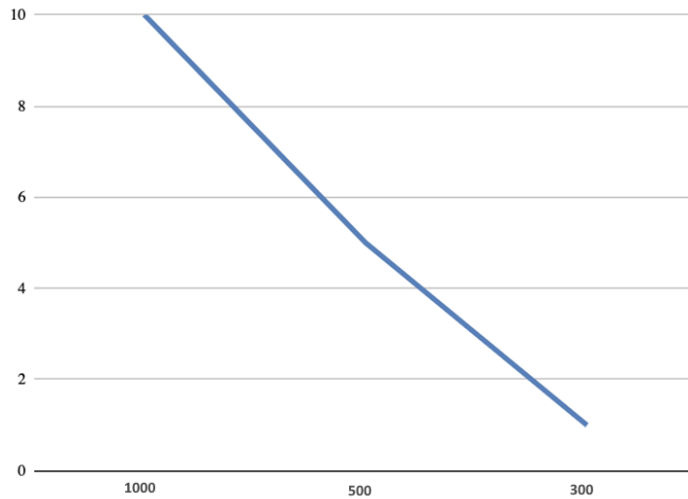


Рисунок 4.5 – Графічна характеристика параметра П4 (потенційний об'єм програмного коду)

4.4 Експертна оцінка параметрів

Для найбільш точної оцінки необхідно залучити експертів. Вони аналізують важливість кожного із визначених параметрів в контексті ефективного досягнення поставленої мети, чим в даному випадку є створення програмного додатку для опису системи мікросервісів у графовій БД.

Чи більша кількість експертів, тим кращий середній результат можна отримати, однак і в цьому питанні є певні логічні межі. Оцінку проводять 5 експертів. Вони роблять свої прогнози на основі попарного порівняння кожного із параметрів між собою. Зрештою вираховується коефіцієнт значимості кожного із запропонованих параметрів. При цьому алгоритм включає наступні кроки:

- 1) Присвоєння кожному із параметрів рангу, на основі якого визначається рівень значимості
- 2) Перевірка експертних оцінок на предмет раціональності
- 3) Визначення оцінки попарного пріоритету
- 4) Обробка отриманої на попередніх кроках інформації та фінальне визначення коефіцієнту значимості

Результати такого ранжування зведено до таблиці 4.3.

Таблиця 4.3 – Експертне ранжування запропонованих параметрів

Параметр	Назва	Одиниці виміру	Визначений експертом ранг параметра					Сума рангів, R_i	Відхилення, Δ_i	Δ_i^2
П1	Швидкодія МП	оп/мс	4	3	5	2	5	19	-4	16
П2	Обсяг використовуваної пам'яті	Мб	1	4	2	5	2	12	3	9
П3	Час аналізу досліджуваної системи	с	5	5	3	3	4	20	-5	25
П4	Потенційний об'єм програмного коду	КБ	2	3	2	1	1	9	6	36
	Разом		12	15	12	11	12	60	0	86

Необхідно провести перевірку степені достовірності оцінок, наданих експертами. Для цього проводяться наступні операції:

- 1) обчислення загальної суми рангів

$$R_{\text{заг}} = \sum_{i=1}^n R_i = 60 \quad (4.1)$$

n – кількість оцінюваних параметрів

- 2) обчислення середньої суми рангів

$$T = \frac{1}{n} R_{\text{заг}} = 15 \quad (4.2)$$

3) обчислення відхилення суми рангів кожного параметра від середньої суми рангів:

$$\Delta_i = R_i - T \quad (4.3)$$

При цьому, сума відхилень по всіх параметрах має дорівнювати нулю.

4) Загальна сума квадратів відхилень

$$\Delta_{\text{заг}} = \sum_{i=1}^n \Delta_i^2 = 86 \quad (4.4)$$

Тепер можна вирахувати коефіцієнт узгодженості

$$W = \frac{12\Delta_{\text{заг}}}{N^2(n^3 - n)} = \frac{12 \cdot 86}{5^2(4^2 - 4)} = 0.68 \quad (4.5)$$

Нормативний коефіцієнт узгодженості $W_H = 0.67$. В даному випадку, $W > W_H$, отже можна зробити висновок, що експертне ранжування – достовірне. Тепер необхідно звести в таблицю попарне порівняння всіх параметрів (табл. 4.4)

Таблиця 4.4 – Попарне порівняння параметрів

Пара параметрів	Експерти					Фінальна оцінка	Числове значення
	1	2	3	4	5		
П1 і П2	>	<	>	<	>	>	1,5
П1 і П3	<	<	>	<	>	<	0.5
П1 і П4	>	=	>	>	>	>	1,5
П2 і П3	<	<	<	>	<	<	0.5
П2 і П4	<	>	=	>	>	>	1,5

ПЗ і П4	>	>	>	>	>	<	0.5
---------	---	---	---	---	---	---	-----

Числове значення переваги параметра i над параметром j (Π_{ij}) вираховується за наступним правилом:

$$\Pi_{ij} = \{1.5 \text{ якщо } i > j, 0.5 \text{ якщо } i < j, 1.0 \text{ якщо } i = j\}$$

Числові значення використовуються при побудові матриці переваги $A = \|\Pi_{ij}\|$.

Також, для кожного із параметрів, необхідно вирахувати коефіцієнт вагомості $K_{\text{ваг}, i}$:

$$K_{\text{ваг}, i} = \frac{b_i}{\sum_{i=1}^n \Pi_{ij}} \quad (4.6)$$

$$b_i = \sum_{i=1}^N \Pi_{ij} b_j \quad (4.7)$$

Процес розрахунку відносних оцінок продовжується до тих пір, поки наступні значення не будуть незначно відрізнятися від попередніх (наприклад, різниця менше 2%). На другому і наступних кроках відносні оцінки розраховуються за наступними формулами:

$$K_{\text{ваг}, i} = \frac{b_i'}{\sum_{i=1}^n \Pi_{ij}} \quad (4.8)$$

$$b_i' = \sum_{i=1}^N \Pi_{ij} b_j \quad (4.9)$$

Таблиця 4.5 – Результати розрахунку вагомості параметрів

Параметр і	Параметр j				Перша іт.		Друга іт.		Третя іт.	
	П1	П2	П3	П4	b_i	$K_{\text{ваг}, i}$	b_i^2	$K_{\text{ваг}, i}^2$	b_i^3	$K_{\text{ваг}, i}^3$
П1	1.0	1.5	0.5	1.5	4.5	0.28	12.22	0.27	62	0.26
П2	1.5	1.0	0.5	1.5	4.5	0.28	12.22	0.27	62	0.26
П3	0.5	0.5	1.0	0.5	2.5	0.16	15.15	0.19	58.25	0.22
П4	1.5	1.5	0.5	1.0	4.5	0.28	12.22	0.27	62	0.26
Всього:					16	1	51.81	1	244.12	1

Отже, за результатами отриманих значень можна сказати, що параметри П1, П2 та П4 мають однакову вагомість, параметр П3 має менший пріоритет.

4.5 Рівень якості варіантів реалізації функцій

Необхідно визначити рівень якості для двох варіантів виконання основних функцій, які були визначені раніше.

Для початку, треба вирахувати коефіцієнти технічного рівня для кожного із варіантів реалізації ПП за формулою:

$$K_k(j) = \sum_{i=1}^n K_{B_{i,j}} B_{i,j} \quad (4.10)$$

де n – кількість визначених параметрів,

$K_{B_{i,j}}$ – визначений раніше коефіцієнт вагомості i -го параметра,

B_i – оцінка i -го параметра в балах.

Результати обчислень зведені до таблиці 4.6.

Таблиця 4.6 – Розрахунок показників рівня якості варіантів реалізації функцій програмного продукту

Функція	Варіант реалізації функції	Параметр	Абсолютне значення параметра	Бальна оцінка параметра	Коефіцієнт вагомості параметра	Коефіцієнт рівня якості
Φ_1	1	П1	10000	7	0.26	1.82
Φ_2	1	П2	300	6	0.26	1.56
Φ_3	1	П3	60	6	0.22	1.32
Φ_4	1	П4	1000	4	0.26	1.04
	2	П4	500	5	0.26	1.3

Взявши дані з таблиці, можна обчислити рівень якості кожного із варіантів:

$$K_1 = 1.82 + 1.56 + 1.32 + 1.04 = 5.74$$

$$K_2 = 1.82 + 1.56 + 1.32 + 1.3 = 6.04$$

Більше значення коефіцієнту технічного рівня виявлено у другого варіанту, отже саме він є кращим.

4.6 Економічний аналіз варіантів розробки ПП

Для розрахунку загальної вартості проекту варто почати із розрахунку трудомісткості.

Обидва варіанти в цілому складаються із двох основних завдань:

- 1) Розробка проекту програмного продукту
- 2) Безпосередня розробка оболонки

Перше завдання за ступенем новизни відноситься до групи А, завдання 2 – до групи Б. Б. Алгоритми, задіяні в процесі виконання завдання 1, за складністю належать до групи 1, а в завданні 2 – до групи 3.

Для реалізації завдання 1 використовується довідкова загальна інформація, а в завданні 2 використовується інформація у вигляді даних.

Проведемо розрахунок норм часу на розробку та програмування для кожного з завдань.

Загальна трудомісткість обчислюється як

$$T_0 = T_p \cdot K_{\Pi} \cdot K_{СК} \cdot K_M \cdot K_{СТ} \cdot K_{СТ.М} \quad (4.11)$$

тут T_p – трудомісткість розробки програмного продукту,

K_{Π} – коефіцієнт поправки,

$K_{СК}$ – коефіцієнт складності вхідної інформації,

K_M – коефіцієнт рівня мови програмування,

$K_{СТ}$ – коефіцієнт використання стандартних модулів і прикладних програм,

$K_{СТ.М}$ – коефіцієнт стандартного математичного забезпечення.

Для першого завдання, взявши до уваги норми часу для завдань розрахункового характеру степеню новизни А та групи складності алгоритму 1, трудомісткість дорівнює: $T_p = 90$ людино-днів. Коефіцієнт поправки, який враховує вид нормативно-довідкової інформації для першого завдання: $K_{\Pi} = 1.7$. Поправочний складності, який враховує складність контролю вхідної та вихідної інформації для всіх семи завдань рівний 1, тобто $K_{СК} = 1$. Оскільки при розробці першого завдання задіяні стандартні модулі, це позначається за допомогою коефіцієнта $K_{СТ} = 0.8$. Зрештою, загальна трудомісткість програмування першого завдання дорівнює:

$$T_1 = 90 \cdot 1.7 \cdot 0.8 = 122.4 \text{ людино-дні}$$

Аналогічні обчислення застосовуються і для другого завдання.

Проте варто зауважити, що для другого завдання використовується алгоритм третьої групи складності, степінь новизни якого Б, тобто $T_p = 27$ людино-днів, $K_{\Pi} = 0.9$, $K_{СК} = 1$, $K_{СТ} = 0.8$. Зрештою:

$$T_2 = 27 \cdot 0.9 \cdot 0.8 = 19.44 \text{ людино-дні}$$

Отримаємо загальну трудомісткість завдань:

$$T_I = (122.4 + 19.44 + 4.8 + 19.44) \cdot 8 = 1328,64 \text{ людино-годин.}$$

$$T_{II} = (122.4 + 19.44 + 6.91 + 19.44) \cdot 8 = 1345.52 \text{ людино-годин.}$$

Найбільш високу трудомісткість має варіант II.

Виходячи з того, що у розробці задіяні 3 програмісти з окладом 18000 грн., один аналітик в області мікросервісних архітектур з окладом 14000, середня зарплата за годину становитиме:

$$Z_{с.г.} = \frac{O_m}{N_p \cdot T} \text{ грн} \quad (4.12)$$

де O_m – загальний місячний оклад працівників,

N_p – кількість робочих днів у тижні,

T - кількість робочих годин в день.

$$Z_c = \frac{68000}{5 \cdot 21 \cdot 8} = 80.95 \text{ грн}$$

Тепер можна вирахувати заробітну плату:

$$Z_c = Z_{с.г.} \cdot T_i \cdot КД \quad (4.13)$$

де Z_c – величина погодинної оплати праці,

T_i – трудомісткість відповідного завдання,

КД – норматив, який враховує додаткову заробітну плату.

Таким чином, зарплата розробників за варіантами становить:

$$I. C_{зп} = 80.95 \cdot 1328.64 \cdot 1.2 = 129701,83 \text{ грн.}$$

$$II. C_{зп} = 80.95 \cdot 1345.52 \cdot 1.2 = 131349,66 \text{ грн.}$$

Необхідно врахувати також відрахування на єдиний соціальний внесок, який становить 22%:

$$\text{I. } C_{\text{ВІД}} = C_{\text{ЗП}} \cdot 0.22 = 129701,83 \cdot 0.22 = 28534,4 \text{ грн.}$$

$$\text{II. } C_{\text{ВІД}} = C_{\text{ЗП}} \cdot 0.22 = 104117,62 \cdot 0.22 = 28896,92 \text{ грн.}$$

Після цього необхідно обрахувати витрати на оплату однієї машино-години. (C_M)

Так як на кожного програміста приходить одна ЕОМ, (а оклад програміста становить 18000 грн., з коефіцієнтом зайнятості 0,2), то для однієї машини отримаємо:

$$C_G = 12 \cdot M \cdot K_3 = 12 \cdot 18000 \cdot 0,2 = 43200 \text{ грн.}$$

З урахуванням додаткової заробітної плати:

$$C_{\text{ЗП}} = C_G \cdot (1 + K_3) = 43200 \cdot (1 + 0.2) = 51840 \text{ грн.}$$

Відрахування на соціальний внесок:

$$C_{\text{ВІД}} = C_{\text{ЗП}} \cdot 0.22 = 51840 \cdot 0,22 = 11404.8 \text{ грн.}$$

Амортизаційні відрахування (при амортизації 25%) та вартості ЕОМ – 14000 грн.

$$C_A = K_{\text{ТМ}} \cdot K_A \cdot \text{Ц}_{\text{ПР}} = 1.15 \cdot 0.25 \cdot 14000 = 4025 \text{ грн.,}$$

тут K_{TM} — коефіцієнт, який враховує витрати на транспортування та налаштування;

K_A — річна норма амортизації;

C_{PP} — договірна ціна приладу.

Витрати на ремонт та профілактику техніки розраховуються наступним чином:

$$C_P = K_{TM} \cdot C_{PP} \cdot K_P = 1.15 \cdot 14000 \cdot 0.05 = 805 \text{ грн.},$$

де K_P — відсоток витрат на поточні ремонти.

Ефективний годинний фонд часу ПК за рік розраховуємо наступним чином:

$$\begin{aligned} T_{EF} &= (D_K - D_B - D_C - D_P) \cdot t_3 \cdot K_B = (365 - 104 - 12 - 16) \cdot 8 \cdot 0.9 = \\ &= 1677.6 \text{ годин,} \end{aligned}$$

де D_K — календарна кількість днів у році;

D_B, D_C — відповідно кількість вихідних та святкових днів;

D_P — кількість днів планових ремонтів устаткування;

t —кількість робочих годин в день;

K_B — коефіцієнт використання приладу у часі протягом зміни.

Витрати на оплату електроенергії:

$$C_{EL} = T_{EF} \cdot N_C \cdot K_3 \cdot C_{EH} = 1677.6 \cdot 0.9 \cdot 3.52 \cdot 0.8 = 4251,7 \text{ грн.},$$

де N_C — середньо-споживча потужність приладу;

K_3 – коефіцієнтом зайнятості приладу;

$\text{Ц}_{\text{ЕН}}$ – тариф за 1 КВт-годин електроенергії.

Накладні витрати:

$$C_H = \text{Ц}_{\text{ПР}} \cdot 0.67 = 20000 \cdot 0,67 = 13400 \text{ грн.}$$

Тоді, річні експлуатаційні витрати будуть:

$$C_{\text{ЕКС}} = 51840 + 11404.8 + 10589,76 + 4025 + 850 + 4251,7 + 13400 = 96361.26 \text{ грн.}$$

Собівартість однієї машино-години ЕОМ дорівнюватиме:

$$C_{\text{М-Г}} = C_{\text{ЕКС}} / T_{\text{ЕФ}} = 96361.26 / 1677,6 = 57.43 \text{ грн/год.}$$

Врахувавши, що в даному проекті всі роботи ведуться на ЕОМ, витрати на оплату машинного часу, в залежності від обраного варіанта реалізації, складатимуть:

$$C_M = C_{\text{М-Г}} \cdot T \quad (4.14)$$

$$\text{I. } C_M = 48.17 \cdot 1328,64 = 64006,5 \text{ грн.}$$

$$\text{II. } C_M = 48.17 \cdot 1345.52 = 64813,69 \text{ грн.}$$

Накладні витрати складають 67% від заробітної плати:

$$C_H = C_{3П} \cdot 0.67 \quad (4.15)$$

$$\text{I. } C_H = 129701,83 \cdot 0,67 = 86900,22 \text{ грн.}$$

$$\text{II. } C_H = 131349,6 \cdot 0,67 = 88004.23 \text{ грн.}$$

Зрештою, загальна вартість розробки програмного продукту за варіантом дорівнює:

$$C_{ПП} = C_{3П} + C_{ВІД} + C_M + C_H \quad (5.20)$$

$$\text{I. } C_{ПП} = 86900,22 + 8236 + 64006,5 + 86900,22 = 246042,94 \text{ грн.}$$

$$\text{II. } C_{ПП} = 88004.23 + 8236 + 64813,69 + 64921,31 + 88004.23 = 253121,58 \text{ грн.}$$

4.7 Вибір кращого варіанту ПП техніко-економічного рівня

Розрахуємо коефіцієнт техніко-економічного рівня за формулою:

$$K_{\text{ТЕР}j} = K_{Kj} / C_{\Phi j} \quad (5.21)$$

$$K_{\text{ТЕР}1} = 4,25 / 246143,32 = 1,72 \cdot 10^{-5},$$

$$K_{\text{ТЕР}2} = 3,74 / 249165,7 = 1,5 \cdot 10^{-5}.$$

Таким чином, саме перший варіант є кращим. Він відповідає наступним параметрам:

- мова програмування – Java;
- допоміжний фреймворк – Object ASM

- графова БД – Neo4j
- середовище розробки – IntelliJ IDEA

4.8 Висновки

Отримані цифри дають приблизну фінансову картину майбутнього продукту. Зваживши на ці показники, можна приймати рішення щодо доцільності старту такого проекту. Подібні рішення, скоріше за все, повинні прийматися людьми, зацікавленими у впровадженні подібних технологій у свої системи в якості моніторингового додатку. Відповідно в такому випадку є можливість порівняти витрати із потенційно збереженими коштами у зв'язку із раннім виявленням підозрілої поведінки в системі. На даний момент витрати видаються оптимальними, проект має шанс на існування.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. The Hows, Whys and Whats of Monitoring Microservices [Електронний ресурс] / [Dave Swersky](#) – 2018 – Режим доступу до ресурсу: <https://thenewstack.io/the-hows-whys-and-whats-of-monitoring-microservices/>
Дата доступу - 16.04.2021
2. Оценка производительности системы по методике APDEX [Електронний ресурс] / Виталий Онянов – 2015 – Режим доступу до ресурсу : <https://tavalik.ru/metodika-apdex/#p3> Дата доступу – 18.04.201
3. The Five Principles of Monitoring Microservices [Електронний ресурс] / [Apurva Dave](#), [Loris Degioanni](#) – 2016 – Режим доступу до ресурсу: <https://thenewstack.io/five-principles-monitoring-microservices/> Дата доступу - 18.04.2021
4. THE BASICS: The Role of Containers in Your Microservice Architecture [Електронний ресурс] / Edward Kisler – 2021 – Режим доступу до ресурсу: <https://jfrog.com/knowledge-base/the-basics-the-role-of-containers-in-your-microservice-architecture/> Дата доступу - 20.04.2021
5. Rygan. Application Performance Monitoring [Електронний ресурс] / 2021 – Режим доступу до ресурсу: <https://raygun.com/platform/apm> Дата доступу: 25.04.2021
6. Trace View Capabilities within Retrace: Determine Where Problems Are and How to Fix Them [Електронний ресурс] / Megan Amos – 2018 – Режим доступу до ресурсу: <https://stackify.com/trace-capabilities-retrace/> Дата доступу: 26.04.2021
7. Prometheus documentation [Електронний ресурс] / Режим доступу до ресурсу: <https://prometheus.io/docs/> Дата доступу – 27.04.2021
8. AppDynamics APM Platform Documentation 21.6 [Електронний ресурс] / Режим доступу до ресурсу: docs.appdynamics.com Дата доступу – 28.04.2021
9. What is a Knowledge Graph? [Електронний ресурс] / Ontotext– 2020 –
Режим доступу до ресурсу:

<https://www.ontotext.com/knowledgehub/fundamentals/what-is-a-knowledge-graph/> Дата доступа - 03.05.2021

10. Neo4j Documentation [Электронный ресурс] / Режим доступа до ресурсу: <https://neo4j.com/docs/> Дата доступа – 05.05.2021

11. 2. Cypher Query Language [Электронный ресурс] / Neo4j Developer – 2021 – Режим доступа до ресурсу: <https://neo4j.com/developer/cypher/> Дата доступа - 13.05.2021

12. ASM Documentation [Электронный ресурс] / Режим доступа до ресурсу: <https://asm.ow2.io/> Дата доступа: 18.05.2021

13. Microservice Architecture with Spring Boot, Spring Cloud and Docker. Application example and source code [Электронный ресурс] / Режим доступа до ресурсу: <https://github.com/sqshq/piggymetrics> Дата доступа: 24.05.2021

14. Download Neo4j [Электронный ресурс] / Режим доступа до ресурсу: <https://neo4j.com/download/> Дата доступа: 29.05.2021

15. Эрик Редмон, Джим Р. Уилсон. Семь баз данных за семь недель. Введение в современные базы данных и идеологию NoSQL. Под редакцией Жаклин Картер / Пер. с. англ. Слинкин А.А. – М.: ДМК Пресс, 2013. – 384с.: ил.

16. Овчинников В.А. Графы в задачах анализа и синтеза структур сложных систем / В.А. Овчинников. – М.: Изд-во МГТУ им. Н. Э. Баумана, 2014. – 423с.: ил.

ДОДАТОК А: КОД ПРОГРАМИ

Main.java

```
public class Main {
    private final static String URL_PROPERTY = "URL";
    private static String host;
    private static String userName;
    private static String pass;

    public static void main(String[] args) {
        new Main().run(args);
    }

    private void run(String[] args) {
        retrieveDBData();

        System.out.println("Program has run!");
        if (args[0].length() < 2 ||
!args[0].equals(CMDCommands.ACCEPT_FILES.getName()) &&
!args[0].equals(CMDCommands.ACCEPT_FILES_SHORT.getName())) {
            System.out.println("Error! Bad input!");
            System.out.println("Usage: --files [file1.jar]
[file2.jar] ...");
            return;
        }

        new Scanner().scan(prepareJarFiles(args));
        Neo4jManager.getInstance().ensureClosing();
        System.out.println("Program finished scanning!");
        System.err.println("Go to: " + host + " to see the
graph! Use credentials: [" + userName + ", " + pass + "]);
    }

    private List<JarFile> prepareJarFiles(String[] args) {
        return Arrays.stream(args)
            .filter(a
!a.equals(CMDCommands.ACCEPT_FILES.getName()))
            .filter(a -> a.endsWith(".jar"))
    }
}
```

```

        .map(this::createJarFile)
        .collect(Collectors.toList());
    }

    private JarFile createJarFile(String fullJarName) {
        try {
            return new JarFile(fullJarName);
        } catch (IOException ex) {
            System.out.println("Error while creating a jar
file of " + fullJarName);
        }

        return null;
    }

    private void retrieveDBData() {
        try
            (FileReader reader = new
FileReader(Neo4jManager.PROPERTIES_FILE_PATH)) {
            Properties props = new Properties();
            props.load(reader);

            String URL = props.getProperty(URL_PROPERTY);

            host = URL.substring(URL.lastIndexOf("@") + 1);
            userName = URL.substring(URL.lastIndexOf("/"),
URL.lastIndexOf(":"));
            pass = URL.substring(URL.lastIndexOf(":") + 1,
URL.lastIndexOf("@"));
        } catch (IOException ex) {
            ex.printStackTrace();
            throw new
PropertiesFileNotFoundException("Couldn't open properties file
[" + Neo4jManager.PROPERTIES_FILE_PATH + "]");
        }
    }
}

```

EndpointNode.class

```
package com.graph.analysis.neo4j.domain;

import lombok.Data;
import lombok.NoArgsConstructor;
import org.neo4j.ogm.annotation.GeneratedValue;
import org.neo4j.ogm.annotation.Id;
import org.neo4j.ogm.annotation.NodeEntity;
import org.neo4j.ogm.annotation.Property;

@Data
@NoArgsConstructor
@NodeEntity(label = "Endpoint")
public class EndpointNode implements Node {
    @Id
    @GeneratedValue
    private Long id;

    @Property
    private String path;

    @Override
    public String getUniqueProperty() {
        return path;
    }
}
```

InterfaceNode.java

```
package com.graph.analysis.neo4j.domain;

import lombok.Data;
import lombok.NoArgsConstructor;
import org.neo4j.ogm.annotation.GeneratedValue;
import org.neo4j.ogm.annotation.Id;
import org.neo4j.ogm.annotation.NodeEntity;
import org.neo4j.ogm.annotation.Property;
```

```

@Data
@NoArgsConstructor
@NodeEntity(label = "Interface")
public class InterfaceNode implements Node {
    @Id
    @GeneratedValue
    private Long id;

    @Property
    private String simpleName;
    @Property
    private String fullName;

    @Override
    public String getUniqueProperty() {
        return fullName;
    }
}

```

JarFileNode.java

```

package com.graph.analysis.neo4j.domain;

import lombok.Data;
import lombok.NoArgsConstructor;
import org.neo4j.ogm.annotation.Id;
import org.neo4j.ogm.annotation.NodeEntity;
import org.neo4j.ogm.annotation.Property;
import org.neo4j.ogm.annotation.Relationship;
import org.neo4j.ogm.annotation.GeneratedValue;

import java.util.HashSet;
import java.util.Set;

@Data
@NoArgsConstructor
@NodeEntity(label = "JAR")
public class JarFileNode implements Node {
    @Id
    @GeneratedValue

```

```

    private Long id;

    @Property
    private String name;

    @Relationship(type = "CONSISTS_OF")
    private Set<PackageNode> packages = new HashSet<>();

    @Override
    public String getUniqueProperty() {
        return name;
    }
}

```

Node.java

```

package com.graph.analysis.neo4j.domain;

public interface Node {
    String getUniqueProperty();
}

```

PackageNode.java

```

package com.graph.analysis.neo4j.domain;

import lombok.Data;
import lombok.NoArgsConstructor;
import org.neo4j.ogm.annotation.Id;
import org.neo4j.ogm.annotation.NodeEntity;
import org.neo4j.ogm.annotation.Property;
import org.neo4j.ogm.annotation.Relationship;
import org.neo4j.ogm.annotation.GeneratedValue;

import java.util.HashSet;
import java.util.Set;

@Data
@NoArgsConstructor
@NodeEntity(label = "Package")
public class PackageNode implements Node {

```

```

    @Id
    @GeneratedValue
    private Long id;

    @Property
    private String name;

    @Relationship(type = "CONTAINS")
    private Set<ClassFileNode> classes = new HashSet<>();

    @Override
    public String getUniqueProperty() {
        return name;
    }
}

```

PublicFieldNode.java

```

package com.graph.analysis.neo4j.domain;

import lombok.Data;
import lombok.NoArgsConstructor;
import org.neo4j.ogm.annotation.GeneratedValue;
import org.neo4j.ogm.annotation.Id;
import org.neo4j.ogm.annotation.NodeEntity;
import org.neo4j.ogm.annotation.Property;

@Data
@NoArgsConstructor
@NodeEntity(label = "PublicField")
public class PublicFieldNode implements Node {
    @Id
    @GeneratedValue
    private Long id;

    @Property
    private String name;
    @Property
    private String type;
    @Property

```

```

        private String uniqueName;

        @Override
        public String getUniqueProperty() {
            return uniqueName;
        }
    }
}

```

PublicMethodNode.java

```

package com.graph.analysis.neo4j.domain;

import lombok.Data;
import lombok.NoArgsConstructor;
import org.neo4j.ogm.annotation.Id;
import org.neo4j.ogm.annotation.NodeEntity;
import org.neo4j.ogm.annotation.Property;
import org.neo4j.ogm.annotation.Relationship;
import org.neo4j.ogm.annotation.GeneratedValue;

import java.util.HashSet;
import java.util.Set;

@Data
@NoArgsConstructor
@NodeEntity(label = "PublicMethod")
public class PublicMethodNode implements Node {
    @Id
    @GeneratedValue
    private Long id;

    @Property
    private String name;
    @Property
    private String uniqueName;

    @Relationship(type = "MAPPED_BY")
    private EndpointNode mappedEndpoint;

    @Relationship(type = "INVOKES")

```

```
        private Set<PublicMethodNode> usedPublicMethods = new
HashSet<>();
```

```
        @Relationship(type = "MAKES_REQUEST_ON")
        private Set<EndpointNode> usedEndpoints = new
HashSet<>();
```

```
        @Override
        public String getUniqueProperty() {
            return uniqueName;
        }
    }
}
```

Neo4jManager.java

```
package com.graph.analysis.neo4j;
```

```
import com.graph.analysis.neo4j.domain.JarFileNode;
import org.neo4j.ogm.config.ClasspathConfigurationSource;
import org.neo4j.ogm.config.Configuration;
import org.neo4j.ogm.config.ConfigurationSource;
import org.neo4j.ogm.cypher.ComparisonOperator;
import org.neo4j.ogm.cypher.Filter;
import org.neo4j.ogm.session.Session;
import org.neo4j.ogm.session.SessionFactory;
```

```
import java.util.Collection;
```

```
public class Neo4jManager {
    public final static String PROPERTIES_FILE_PATH =
"neo4j/neo4j.properties";
    private final static String DOMAIN_PACKAGE_NAME =
"com.graph.analysis.neo4j.domain";
    private final static String JAR_FILE_NODE_PROPERTY_NAME
= "name";
    private static Neo4jManager neo4jManagerInstance;
    private SessionFactory sessionFactory;
    private Session currentSession;

    public static Neo4jManager getInstance() {
```

```

        if (neo4jManagerInstance == null) {
            neo4jManagerInstance = new Neo4jManager();
        }

        return neo4jManagerInstance;
    }

    public boolean saveJarFileNode(JarFileNode
jarNodeToSave) {
        Collection<JarFileNode> existingNodes =
currentSession.loadAll(JarFileNode.class,
            new Filter(JAR_FILE_NODE_PROPERTY_NAME,
ComparisonOperator.EQUALS, jarNodeToSave.getName()));

        if (existingNodes != null &&
!existingNodes.isEmpty()) {
            System.out.println("There are such jars nodes
already! (" + existingNodes.size() + ")");
            existingNodes.forEach(node ->
System.out.println("id" + node.getId() + ":" + node.getName()));
            return false;
        }

        currentSession.save(jarNodeToSave);
        System.out.println("Jar Node (" +
jarNodeToSave.getName() + ") saved!");

        return true;
    }

    public void ensureClosing() {
        currentSession = null;
        sessionFactory.close();
    }

    private Neo4jManager() {
        init();
    }

    private void init() {

```

```

        ConfigurationSource props = new
ClasspathConfigurationSource(PROPERTIES_FILE_PATH);
        Configuration config = new
Configuration.Builder(props).build();

        sessionFactory = new SessionFactory(config,
DOMAIN_PACKAGE_NAME);
        currentSession = sessionFactory.openSession();
    }
}

```

Scanner.java

```
package com.graph.analysis.scan;
```

```

import com.graph.analysis.NodesCache;
import com.graph.analysis.neo4j.Neo4jManager;
import com.graph.analysis.neo4j.domain.ClassFileNode;
import com.graph.analysis.neo4j.domain.JarFileNode;
import com.graph.analysis.neo4j.domain.Node;
import com.graph.analysis.neo4j.domain.PackageNode;
import com.graph.analysis.scan.visitor.ClassVisitor;
import org.objectweb.asm.ClassReader;
import org.objectweb.asm.Opcodes;

```

```

import java.io.IOException;
import java.io.InputStream;
import java.util.Enumeration;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.jar.JarEntry;
import java.util.jar.JarFile;

```

```

import static
com.graph.analysis.scan.visitor.VisitorUtil.isValidClassPath;

```

```

public class Scanner {
    private final NodesCache cache =
NodesCache.getInstance();

```

```

public void scan(List<JarFile> jarsToScan) {
    jarsToScan.forEach(this::scanJar);
}

private void scanJar(JarFile jarFile) {
    System.out.println("-----Scanning jar file ["
+ jarFile.getName() + "]-----");

    JarFileNode jarNode = new JarFileNode();
    jarNode.setName(jarFile.getName());

    Map<String, PackageNode> packageMap = new
HashMap<>();
    Enumeration<JarEntry> jarEntries =
jarFile.entries();
    while (jarEntries.hasMoreElements()) {
        JarEntry nextEntry = jarEntries.nextElement();
        String entryName = nextEntry.getName();
        if (isValidClassPath(entryName)) {
            String classPackageName =
entryName.substring(0, nextEntry.getName().lastIndexOf("/") +
1);

            if
(!packageMap.containsKey(classPackageName)) {
                PackageNode packageNode = new
PackageNode();

                packageNode.setName(classPackageName);
                packageMap.put(classPackageName,
packageNode);
            }
            try (InputStream classFileIS =
jarFile.getInputStream(nextEntry)) {
                Node classNode =
cache.getNodeIfPresent(ClassFileNode.class.getName(),
nextEntry.getName());

                if (classNode == null) {
                    classNode = new ClassFileNode();
                }
            }
        }
    }
}

```

```

        ClassReader      reader      =      new
ClassReader(classFileIS);
        reader.accept(new
ClassVisitor(Opcodes.ASM5, (ClassFileNode) classNode), 0);

packageMap.get(classPackageName).getClasses().add((ClassFileNode
) classNode);

        cache.putNodeIfAbsent(classNode);
        System.out.println("Finished      scanning
the file: " + nextEntry.getName());
        } catch (IOException ex) {
            System.out.println("Cannot      read      class
file [" + nextEntry.getName() + "]");
            ex.printStackTrace();
        }
    }
}

        packageMap.values().forEach(p
jarNode.getPackages().add(p));
->

Neo4jManager.getInstance().saveJarFileNode(jarNode);
    }
}

```

ClassVisitor.java

```

package com.graph.analysis.scan.visitor;

import com.graph.analysis.NodesCache;
import com.graph.analysis.neo4j.domain.*;
import org.objectweb.asm.AnnotationVisitor;
import org.objectweb.asm.FieldVisitor;
import org.objectweb.asm.MethodVisitor;
import org.objectweb.asm.ModuleVisitor;
import org.objectweb.asm.*;

import java.util.Arrays;
import java.util.Set;

```

```

import static
com.graph.analysis.scan.visitor.VisitorUtil.*;

public class ClassVisitor extends
org.objectweb.asm.ClassVisitor {
    private final static NodesCache CACHE =
NodesCache.getInstance();
    private final static String JAVA_LANG_OBJECT =
"java/lang/Object";
    private final static String REQUEST_MAPPING_ANN_NAME =
"RequestMapping";
    private ClassFileNode classFileNode;

    public ClassVisitor(int api, ClassFileNode nodeHolder)
{
    super(api);
    this.classFileNode = nodeHolder;
}

    @Override
    public void visit(int version, int access, String name,
String signature, String superName, String[] interfaces) {
        classFileNode.setFullName(name);
        classFileNode.setSimpleName(getSimpleName(name));

        if (!superName.equals(JAVA_LANG_OBJECT)) {
            Node parentClassNode =
CACHE.getNodeIfPresent(ClassFileNode.class.getName(),
superName);

            if (parentClassNode == null) {
                parentClassNode = new ClassFileNode();
                ((ClassFileNode)
parentClassNode).setFullName(superName);
                ((ClassFileNode)
parentClassNode).setSimpleName(getSimpleName(superName));
                CACHE.putNodeIfAbsent(parentClassNode);
            }

```

```

        classFileNode.setExtendedClass((ClassFileNode)
parentClassNode);
    }

    Arrays.stream(interfaces)
        .map(fullName -> {
            Node interfaceNode =
CACHE.getNodeIfPresent(InterfaceNode.class.getName(), fullName);

            if (interfaceNode == null) {
                interfaceNode = new
InterfaceNode();
                ((InterfaceNode)
interfaceNode).setFullName(fullName);
                ((InterfaceNode)
interfaceNode).setSimpleName(getSimpleName(fullName));

CACHE.putNodeIfAbsent(interfaceNode);
            }

classFileNode.getImplementedInterfaces().add((InterfaceNode)
interfaceNode);

            return interfaceNode;
        })
        .forEach(node ->
classFileNode.getImplementedInterfaces().add((InterfaceNode)
node));

    super.visit(version, access, name, signature,
superName, interfaces);
}

@Override
    public FieldVisitor visitField(int access, String name,
String descriptor, String signature, Object value) {
        if ((access & Opcodes.ACC_PUBLIC) > 0) {

```

```

        String                fieldUniqueName                =
classFileNode.getFullName() + ":" + name;
        Node                publicFieldNode                =
CACHE.getNodeIfPresent(PublicFieldNode.class.getName(),
fieldUniqueName);

        if (publicFieldNode == null) {
            publicFieldNode = new PublicFieldNode();
            ((PublicFieldNode)
publicFieldNode).setName(name);
            ((PublicFieldNode)
publicFieldNode).setUniqueName(fieldUniqueName);
            ((PublicFieldNode)
publicFieldNode).setType(retrieveType(descriptor));
            CACHE.putNodeIfAbsent(publicFieldNode);
        }

classFileNode.getDeclaredPublicFields().add((PublicFieldNode)
publicFieldNode);

        return new
com.graph.analysis.scan.visitor.FieldVisitor(Opcodes.ASM5,
(PublicFieldNode) publicFieldNode, classFileNode);
    }

    return super.visitField(access, name, descriptor,
signature, value);
}

@Override
    public MethodVisitor visitMethod(int access, String
name, String descriptor, String signature, String[] exceptions)
    {
        Set<String>                dependentTypes                =
retrieveDependentTypes(descriptor);
        dependentTypes.forEach(t -> {
            Node                dependentTypeNode                =
CACHE.getNodeIfPresent(ClassFileNode.class.toString(), t);
            if (dependentTypeNode == null) {

```

```

        dependentTypeNode = new ClassFileNode();
        ((ClassFileNode)
dependentTypeNode).setFullName(t);
        ((ClassFileNode)
dependentTypeNode).setSimpleName(getSimpleName(t));
        CACHE.putNodeIfAbsent(dependentTypeNode);
    }

classFileNode.getUsedClasses().add((ClassFileNode)
dependentTypeNode);
    });

    if ((access & Opcodes.ACC_PUBLIC) > 0 &&
isMethodValuableToBeScanned(name)) {
        String methodUniqueName =
classFileNode.getFullName() + ":" + name;
        Node publicMethodNode =
CACHE.getNodeIfPresent(PublicMethodNode.class.getName(),
methodUniqueName);

        if (publicMethodNode == null) {
            publicMethodNode = new PublicMethodNode();
            ((PublicMethodNode)
publicMethodNode).setName(name);
            ((PublicMethodNode)
publicMethodNode).setUniqueName(methodUniqueName);
            CACHE.putNodeIfAbsent(publicMethodNode);
        }

classFileNode.getDeclaredPublicMethods().add((PublicMethodNode)
publicMethodNode);

        return new
com.graph.analysis.scan.visitor.MethodVisitor(Opcodes.ASM5,
(PublicMethodNode) publicMethodNode,
classFileNode);
    }

```

```

        return super.visitMethod(access, name, descriptor,
signature, exceptions);
    }

    @Override
    public AnnotationVisitor visitAnnotation(String
descriptor, boolean visible) {
        if (descriptor.contains(REQUEST_MAPPING_ANN_NAME))
{
            return new
com.graph.analysis.scan.visitor.AnnotationVisitor(Opcodes.ASM5,
classFileNode);
        }
        return super.visitAnnotation(descriptor, visible);
    }

    @Override
    public ModuleVisitor visitModule(String name, int
access, String version) {
        return new
com.graph.analysis.scan.visitor.ModuleVisitor(Opcodes.ASM5,
classFileNode);
    }
}

```