Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук
(повна назва)

Кафедра програмної інженерії
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

Дослідження методів автоматизації API Governance: керування інтерфейсами у мікросервісній системі

(тема)

Виконав:
студент 2 курсу, групи ІПЗм-22-3

Подорожний М.П.
(прізвище, ініціали)

Спеціальність 121 – Інженерія програмного забезпечення
(код і повна назва спеціальності)

Тип програми освітньо-наукова

Керівник доц. Ревенчук І.А.
(посада, прізвище, ініціали)

Допускається до захисту
Зав. кафедри _____ Дудар З.В.
(підпис) (прізвище, ініціали)

2024 р.

Харківський національний університет радіоелектроніки

Факультет комп’ютерних наук
Кафедра програмної інженерії
Рівень вищої освіти другий (магістерський)
Спеціальність 121 – Інженерія програмного забезпечення
Тип програми освітньо-наукова програма
Освітня програма Інженерія програмного забезпечення
(шифр і назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«___» _____ 2024 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Подорожному Михайлу Павловичу
(прізвище, ім’я, по батькові)

1. Тема роботи «Дослідження методів автоматизації API Governance: керування інтерфейсами у мікросервісній системі».

Затверджена наказом по університету від 29 березня 2024 р. № 250 Ст

2. Термін подання студентом роботи до екзаменаційної комісії 20 червня 2024 р.

3. Вихідні дані до роботи: дослідження методів автоматизації API Governance: керування інтерфейсами у мікросервісній системі.

4. Перелік питань, що потрібно опрацювати в роботі: дослідження існуючих api governance систем, які взаємодіють із мікросервісами, аналіз та порівняння існуючих варіантів реалізації api governance, доскональне вивчення принципів роботи кожної з обраних систем, розробка рекомендацій щодо впровадження подібного функціоналу в інші api governance системи, експериментальне власне розширення та проведено розробку персонального розширення для API.

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1	Видача теми, узгодження і затвердження	02.04.2024	*виконано*
2	Аналіз предметної галузі	02.04.2024	*виконано*
3	Огляд існуючих методів	09.04.2024	*виконано*
4	Оформлення пояснювальної записки	05.05.2024	*виконано*
5	Здача готового проекту	20.06.2024	*виконано*

Дата видачі завдання: 20 січня 2024 р.

Студент _____ Подорожний М.П.
(підпис)

Керівник роботи _____ доц. Ревенчук І.А.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ / ABSTRACT

Пояснювальна записка: 61 с., 7 рис., 1 табл, 13 джерел, 5 додатків.

API GOVERNANCE, АВТОМАТИЗАЦІЯ, МІКРОСЕРВІСНА ІНТЕРФЕЙСИ, СИСТЕМА, УПРАВЛІННЯ.

Об'єктом глибокого аналізу та дослідження є методи автоматизації API Governance для ефективного керування інтерфейсами в мікросервісній системі. Задачею дослідження є аналіз, розробка та впровадження передових автоматизованих стратегій для управління API в умовах розгалужених мереж мікросервісів.

У процесі наукового дослідження застосовуються передові підходи сучасних технологій розробки програмного забезпечення, зокрема ті, які пов'язані з автоматизацією управління інтерфейсами у мікросервісах. Великий акцент приділяється розгляду різноманітних варіантів методів управління API та їх адаптації до потреб розподіленої архітектури.

Основні напрямки дослідження включають аналіз існуючих підходів до управління API, визначення ключових вимог до автоматизованих рішень та розробку механізмів інтеграції з існуючими мікросервісами.

В результаті дослідження розробляються та впроваджуються комплексні інструменти для автоматизації API Governance. Це сприяє не лише ефективному керуванню інтерфейсами в мікросервісній архітектурі, але й підвищує загальну робочу ефективність та безпеку управління API в комплексних та розгалужених середовищах.

API GOVERNANCE, AUTOMATION, MICROSERVICE INTERFACES, SYSTEM, MANAGEMENT.

API Governance automation methods for effective management of interfaces in a microservice system are the object of in-depth analysis and research. The goal of the research is the analysis, development and implementation of advanced automated strategies for API management in the context of distributed networks of microservices.

In the process of scientific research, advanced approaches of modern software development technologies are used, in particular those related to the automation of interface management in microservices. Great emphasis is placed on consideration of various options for API management methods and their adaptation to the needs of a distributed architecture.

The main areas of research include analysis of existing approaches to API management, identification of key requirements for automated solutions, and development of integration mechanisms with existing microservices.

As a result of the research, comprehensive tools for automating API Governance are developed and implemented. This not only contributes to the efficient management of interfaces in a microservice architecture, but also improves the overall operational efficiency and security of API management in complex and distributed environments.

Я, Подорожний Михайло Павлович, студент гр. ІПЗм-22-6, здобувач вищої освіти на другому (магістерському) рівні кафедри «Програмна інженерія», заявляю: моя робота на тему «Дослідження методів автоматизації API Governance: керування інтерфейсами у мікросервісній системі», що буде представлена в екзаменаційну комісію для публічного захисту, виконана самостійно, в ній не містяться елементи плагіату і вона може бути опублікована в електронному архіві відкритого доступу ElArKhNURE. Всі запозичення з друкованих та електронних джерел мають відповідні посилання.

Я ознайомлений(на) з діючим положенням «Про протидію академічному плагіату в ХНУРЕ», згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування дисциплінарних заходів.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

JS – JavaScript

NPM – Node Package Manager

SSG – Static Site Generation

SSR – Server-Side Rendering

TS – TypeScript

CSR – Client-Side Rendering

ВСТУП

У невпинному впливі стрімкого росту мікросервісної архітектури важливим елементом стає управління API, яке визначає гнучкість та ефективність взаємодії в складних мережах. З урахуванням стрімких змін у сучасних технологіях, необхідно постійно вдосконалювати методи управління інтерфейсами для надійної та безпечної взаємодії між сервісами. Дане дослідження приділяє особливу увагу розробці та впровадженні автоматизованих стратегій для ефективного керування API в системах мікросервісів.

В умовах стрімкого зростання кількості мікросервісів та їхнього розгалуження перед інженерами постають великі завдання щодо забезпечення консистентності та легкості управління інтерфейсами. Це особливо актуально у великих корпоративних системах, де виникає необхідність у створенні єдино центрального та ефективного механізму управління API.

Головною метою цього дослідження є дослідження та розробка інноваційних рішень для автоматизації API Governance в умовах мікросервісної архітектури. Створення інструментів, які не лише гарантують ефективність та надійність, але також легко інтегруються з різноманітними технологічними стеками, є великим викликом, що характерний для сучасних підходів до розробки.

Проведення дослідження передбачає аналіз існуючих методів управління API, визначення їхніх переваг та обмежень, а також розробку та тестування ефективних автоматизованих стратегій. Важливим етапом роботи є розгляд можливостей інтеграції цих стратегій у вже діючі мікросервісні системи.

Дослідження буде здійснюватися шляхом аналізу актуальних технологій, специфікацій та нормативів, пов'язаних із управлінням API в мікросервісах. Особлива увага буде приділена розробці інтелектуальних механізмів, які дозволять системам адаптуватися до змін у конфігурації та ефективно взаємодіяти з різними типами сервісів.

Заключні висновки роботи будуть не лише включати оцінку розроблених стратегій, але й намагатимуться визначити перспективи їхнього впровадження у практиці. Розгляд можливостей впровадження результатів у реальних проектах та

подальшого розвитку цього напрямку стане важливою частиною завершального етапу дослідження.

Однаково важливим є і розгляд можливостей впровадження отриманих знань у великих технологічних екосистемах, враховуючи змінність та вимоги до стандартів у майбутньому. Впровадження результатів дослідження у широкому масштабі та їхній вплив на розвиток інформаційних технологій стануть ключовими аспектами довгострокового успіху даного проекту.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Аналіз предметної області систем управління інтерфейсами та запитами

Мікросервісна архітектура, що утверджується як один із важливих стандартів у розробці програмного забезпечення, висуває високі вимоги до систематизації та ефективного керування інтерфейсами. Саме тут відіграє ключову роль автоматизоване керування інтерфейсами, що стає невід'ємною частиною забезпечення консистентності та безпеки мікросервісів.

Важливість автоматизованого керування інтерфейсами проявляється у вирішенні складних завдань, пов'язаних із ручним керуванням великою кількістю сервісів. Зі зростанням якісного та кількісного показника мікросервісів наголошується на необхідності швидких та ефективних методів управління їхніми інтерфейсами для забезпечення їхньої безпеки та функціональності.

Автоматизоване керування інтерфейсами активно сприяє підвищенню ефективності розробки та впровадження нових сервісів. Цей підхід значно зменшує час, необхідний для внесення змін у існуючі сервіси та розгортання нового функціоналу, забезпечуючи тим самим швидкі та гнучкі вирішення завдань.

Забезпечення консистентності та безпеки взаємодії мікросервісів є ще однією важливою перевагою автоматизованого керування інтерфейсами. Гарантована стабільність роботи і усунення можливих помилок, пов'язаних із невірним визначенням чи зміною інтерфейсів, забезпечує високий рівень надійності системи в цілому.

Моніторинг та аналіз інтерфейсів в реальному часі не лише оперативно виявляє та вирішує проблеми, але й відкриває широкі можливості для подальшої оптимізації та вдосконалення системи. Це необхідне знаряддя для підтримання високого рівня ефективності та відповідності вимогам користувачів.

Використання автоматизованих методів керування інтерфейсами є критично важливим для забезпечення ефективності та стабільності у мікросервісній архітектурі. Це стає ключовим фактором для успішного функціонування сервісів у сучасному інформаційному середовищі, де вимоги до швидкості та надійності взаємодії стають визначальними в реалізації комплексних програмних рішень [1].

У світі швидкого розвитку інформаційних технологій мікросервісна архітектура набуває все більшої популярності як ефективний підхід до розробки програмного забезпечення. З введенням цього підходу, виникає необхідність в систематичному та ефективному управлінні інтерфейсами, яке відіграє ключову роль у забезпеченні гнучкості та надійності мікросервісних систем.

Серед переваг використання автоматизованих методів керування інтерфейсами у мікросервісах важливо виділити швидкість та ефективність впровадження нового функціоналу чи змін. Завдяки автоматизованим стратегіям, розгортання нових сервісів та внесення змін у вже існуючі може відбуватися значно швидше, що є критичним аспектом у конкурентному середовищі.

Другим аспектом, який слід враховувати, є підвищення стійкості системи та зниження ймовірності виникнення помилок. Основні завдання автоматизації API Governance включають у себе контроль за правильністю визначення інтерфейсів та їхньої відповідності функціональним вимогам. Це забезпечує стабільну роботу сервісів та знижує ризик неправильного їх використання.

Важливим фактором у контексті мікросервісної архітектури є також можливість моніторингу та аналізу інтерфейсів в реальному часі. Це дозволяє оперативно виявляти та вирішувати проблеми, підвищуючи тим самим ефективність системи. Збір та аналіз даних з інтерфейсів стає важливою складовою для подальшої оптимізації та вдосконалення системи.

У підсумку, важливість автоматизації API Governance у мікросервісах визначається потребою у невідкладній реакції на зміни, а також забезпеченням стабільності та надійності в умовах динамічного інформаційного середовища. Впровадження автоматизованих методів керування інтерфейсами стає вирішальним елементом для успішного функціонування мікросервісних систем у сучасному програмному ландшафті.

У глибокій мікросервісній архітектурі, де складність та розгалуженість системи наростають експоненційно, впровадження автоматизованих методів керування інтерфейсами вносить свої виклики та розгортає перед інженерами нові горизонти [2].

Серед ключових труднощів можна виділити проблему стандартизації. У світі мікросервісів, де кожен сервіс може використовувати власний технологічний стек та протоколи взаємодії, навести єдиний стандарт для керування інтерфейсами є завданням не зовсім тривіальним. Велика кількість сервісів вимагає гнучкості та адаптабельності в стратегіях автоматизації.

З іншого боку, надмірна динаміка мікросервісів створює виклик у відстеженні та адаптації до змін. Оновлення та модифікації інтерфейсів вимагають невпинного удосконалення автоматизованих стратегій, щоб забезпечити їхню актуальність та ефективність.

Важливою аспектом є також взаємодія з різноманіттю технологій. Різні сервіси можуть використовувати різні мови програмування, бази даних та протоколи, що ставить під сумнів можливість створення універсального рішення для всіх випадків.

Додатковою трудністю може стати нестача стандартизованого підходу до моніторингу та аналізу взаємодії API. Враховуючи різноманітність інструментів та методів аналізу, обрання ефективних засобів може виявитися викликом.

Слід зазначити, що недоліки та виклики автоматизації API Governance у мікросервісах визначаються потребою універсалізації стратегій, постійною адаптацією до змін, різноманіттям технологічних стеків та необхідністю створення гнучких рішень для керування інтерфейсами у високодинамічному середовищі мікросервісів.

Зі зростанням кількості мікросервісів виникають виклики щодо забезпечення консистентності та зменшення складності управління інтерфейсами. Розробники стикаються з завданням синхронізації та забезпечення відповідності інтерфейсів до змін в системі. Автоматизоване керування інтерфейсами вирішує ці виклики, роблячи процеси більш прозорими та ефективними.

Інструментарій автоматизованого керування інтерфейсами включає в себе різноманітні технології та бібліотеки, спрямовані на полегшення роботи розробників. Деякі з них, такі як Swagger, GraphQL та Postman, дозволяють описувати, тестувати та моніторити API в автоматичному режимі. Інші

інструменти забезпечують централізоване управління конфігурацією та моніторингом інтерфейсів.

Здійснення автоматизованого керування інтерфейсами призводить до ряду переваг. Однією з них є покращення ефективності розробки та впровадження нових сервісів. Зменшення часу, необхідного для внесення змін в існуючі сервіси та впровадження нового функціоналу, сприяє більш швидкому реагуванню на вимоги бізнесу.

Ще однією важливою перевагою автоматизованого керування інтерфейсами є гарантія стабільної роботи та усунення можливості помилок, пов'язаних з невірним визначенням чи зміною інтерфейсів. Це стає вирішальною умовою для забезпечення надійності та неперервності роботи системи в цілому [3].

Важливою складовою автоматизованого керування інтерфейсами є моніторинг та аналіз інтерфейсів в реальному часі. Це дозволяє оперативно виявляти та вирішувати проблеми, що виникають при взаємодії сервісів. Такий підхід сприяє не лише підвищенню ефективності системи, але й надає можливість аналізу для подальшої оптимізації та вдосконалення.

Тож можна сказати, що використання автоматизованих методів керування інтерфейсами визначається необхідністю забезпечення ефективності та стабільності у мікросервісній архітектурі. Це стає ключовим фактором для успішного функціонування сервісів у сучасному інформаційному середовищі, де вимоги до швидкості та надійності взаємодії є суттєвими. Автоматизація керування інтерфейсами не лише полегшує роботу розробників, але й гарантує високий рівень якості та безпеки в мікросервісній архітектурі.

Враховуючи вищезазначені аспекти, можна визначити, що автоматизація інтерфейсів в мікросервісах відіграє критичну роль у забезпеченні ефективності та надійності системи. Цей підхід не лише полегшує роботу розробників, але й гарантує високий рівень керованості, безпеки та гнучкості мікросервісної архітектури.

Автоматизація API Governance включає в себе використання різних інструментів та систем для забезпечення відповідності, безпеки та ефективності

інтерфейсів у мікросервісах. Роль цих інструментів полягає у виявленні, вирішенні та запобіганні проблемам, пов'язаним з інтерфейсами сервісів.

Інструменти автоматизації дозволяють проводити валідацію API, перевіряти їхню відповідність заданим стандартам та автоматично генерувати документацію. Наприклад, використання Swagger чи OpenAPI дозволяє автоматично створювати та оновлювати документацію API, що полегшує взаємодію з ними для розробників та користувачів.

Інструменти моніторингу та логування надають можливість в режимі реального часу слідкувати за станом роботи API, виявляти аномалії та забезпечувати оперативну реакцію на можливі проблеми. Це є критичним для забезпечення неперервності роботи мікросервісів.

Інструменти автоматизації API Governance повинні інтегруватися з процесами Continuous Integration та Continuous Delivery (CI/CD). Це дозволяє автоматизовано виконувати тести, перевіряти безпеку та валідацію при внесенні змін в інтерфейси, забезпечуючи безперервний процес вдосконалення.

Інструменти автоматизації API Governance грають ключову роль у забезпеченні безпеки взаємодії мікросервісів. Вони включають засоби автоматизованого керування доступом, шифрування та інші механізми, що гарантують захищеність обміну даними між сервісами.

Автоматизація API Governance також має спрямовуватися на масштабованість та гнучкість системи. Інструменти повинні забезпечувати ефективну роботу у великих та динамічних мікросервісних архітектурах.

Розглянуті інструменти та підходи автоматизації API Governance в мікросервісній системі визначають ефективний шлях для забезпечення високої якості, безпеки та надійності взаємодії сервісів. Їхнє впровадження дозволяє оптимізувати розробку, поліпшує безпеку та сприяє стабільності мікросервісної архітектури [4].

В контексті дослідження методів автоматизації API Governance у мікросервісній системі, важливо визначити переваги, які ці інструменти можуть

надати для ефективного керування інтерфейсами та забезпечення їхньої стабільності та безпеки.

Автоматизовані інструменти API Governance грають важливу роль у визначенні та забезпеченні дотримання стандартів розробки та взаємодії сервісів. Це включає в себе автоматичну перевірку відповідності коду до встановлених норм, що забезпечує єдність інтерфейсів та запобігає виникненню помилок.

Шодо тестування, то інструменти API Governance автоматизують процеси тестування та валідації інтерфейсів. Це не тільки спрощує виявлення помилок та неполадок, але і дозволяє швидко реагувати на зміни в коді та впроваджувати новий функціонал без порушення стабільності системи.

За допомогою автоматизованих інструментів можна вести моніторинг використання API, аналізувати швидкість відповідей та виявляти можливі точки витоку продуктивності. Це сприяє оперативній реакції на проблеми та удосконаленню роботи сервісів.

Інструменти автоматизації API Governance активно залучені до забезпечення безпеки взаємодії мікросервісів. Це включає в себе автоматичну перевірку на вразливості, шифрування даних та ефективне керування доступом до ресурсів.

Автоматизовані засоби дозволяють налагоджувати процеси Continuous Integration та Continuous Delivery (CI/CD). Це допомагає підтримувати неперервний цикл вдосконалення, забезпечуючи швидке впровадження змін та нового функціоналу.

Автоматизація API Governance повинна легко інтегруватися з іншими інструментами розробки, такими як системи контролю версій, інструменти тестування, різноманітні IDE тощо. Це дозволяє забезпечити єдність процесів розробки та зробити їх більш ефективними.

Тож, автоматизація API Governance в мікросервісній архітектурі виступає як ключовий елемент для забезпечення ефективності, стабільності та безпеки інтерфейсів сервісів. Її впровадження приводить до поліпшення якості розробки, надійності системи та забезпечує шлях до подальшого розвитку мікросервісної архітектури.

В рамках дослідження методів автоматизації API Governance у мікросервісній системі, важливо розглянути автоматизований процес керування інтерфейсами, який визначає ефективність, стабільність та безпеку взаємодії сервісів.

Один з ключових етапів в автоматизованому процесі – це генерація документації для API. Використання відповідних інструментів дозволяє автоматично створювати чітку та оновлювану документацію, яка відображає всі можливості сервісів та визначає способи їх використання.

Автоматизований процес включає в себе систему моніторингу та логування інтерфейсів. За допомогою відповідних інструментів можна в режимі реального часу слідкувати за використанням API, виявляти аномалії та швидко реагувати на можливі проблеми [4].

Автоматизовані тести дозволяють перевірити функціональність API в автоматичному режимі. Це включає тестування різних сценаріїв взаємодії, валідацію даних та перевірку стабільності системи під час навантаження.

Процес автоматизації включає в себе інструменти для перевірки доступності та відмовостійкості API. Це дозволяє забезпечити неперервний доступ до сервісів та забезпечити їхню надійність.

Автоматизований процес включає ефективний механізм керування версіями API та забезпечення сумісності між різними версіями. Це гарантує безперервний розвиток системи без порушення вже існуючого функціоналу.

Автоматизований моніторинг перформансу і оптимізація інтерфейсів є ключовою складовою. Засоби аналізу та оптимізації дозволяють визначити та усунути можливі проблеми, що можуть виникнути під час роботи системи в реальному часі [5].

В рамках автоматизованого керування інтерфейсами важливо мати систему автоматичних попереджень та резервного запасу. Це дозволяє оперативно реагувати на будь-які проблеми та забезпечує неперервну роботу системи.

Тож, Автоматизований процес керування інтерфейсами в мікросервісній системі є важливою складовою для забезпечення ефективності, стабільності та

безпеки. Впровадження відповідних інструментів дозволяє забезпечити надійну роботу системи та полегшити роботу розробників.

1.2 Автоматизоване керування клієнтським рендерінгом у мікросервісній архітектурі

Автоматизоване керування клієнтським рендерінгом в мікросервісній архітектурі визначає ефективний спосіб оптимізації та керування відображенням інтерфейсів для кінцевого користувача. Розглянемо ключові аспекти цього процесу.

Автоматизований вибір оптимального методу рендерінгу на клієнті є важливою частиною API Governance. Враховуючи особливості різних методів, таких як SPA (Single Page Application) чи SSR (Server-Side Rendering), можна забезпечити оптимальні умови для відображення контенту.

Застосування автоматизованого підходу дозволяє оптимізувати інтерфейси для різних платформ та пристроїв. Наприклад, використання responsive design або інших технік автоматично адаптує інтерфейс для різних умов відображення.

Керування інтерфейсами включає в себе автоматизований контроль за візуальною спрямованістю та користувацьким досвідом. Інструменти для тестування та аналізу UX дозволяють виявляти можливі проблеми та покращити загальний користувацький досвід.

Автоматизований моніторинг інтерфейсів на клієнті в реальному часі гарантує швидке виявлення та усунення можливих проблем, таких як затримки завантаження чи помилки відображення.

Автоматизоване забезпечення сумісності інтерфейсів з різними браузерами є критичним етапом. Застосування інструментів для автоматичного тестування та перевірки сумісності гарантує коректну роботу в різних середовищах.

Аналіз використання ресурсів на клієнті є важливим кроком у забезпеченні ефективності відображення інтерфейсів. Автоматизовані інструменти визначають та оптимізують завантаження ресурсів для поліпшення продуктивності.

Автоматизоване керування клієнтським рендерінгом у мікросервісній системі допомагає забезпечити ефективність, надійність та зручність відображення інтерфейсів для користувачів. Інтеграція відповідних інструментів у процес розробки дозволяє автоматизувати багато етапів від створення до вдосконалення інтерфейсів.

Автоматизоване керування Static Site Generation (SSG) в мікросервісній архітектурі є ключовим елементом забезпечення ефективності та керування інтерфейсами відображення. Розглянемо основні аспекти використання SSG в контексті API Governance.

SSG визначається як процес створення статичних HTML-сторінок під час збірки додатку. Це дозволяє заздалегідь згенерувати сторінки та забезпечити їх ефективне відображення, зменшуючи час завантаження та витрати ресурсів.

Щоб забезпечити актуальність вмісту, важливо мати автоматизований механізм оновлення статичних сайтів. Це може включати в себе періодичні або подійні збірки сторінок для відображення останніх даних.

Автоматизоване керування SSG передбачає інтеграцію з мікросервісами. Забезпечення генерації статичних сторінок на основі даних, що надходять від різних мікросервісів, дозволяє створювати комплексні та зручні інтерфейси для користувачів.

Застосування автоматизованих інструментів для оптимізації ресурсів стає важливим етапом. Мінімізація та компресія CSS, JavaScript та інших ресурсів під час автоматизованої збірки сприяє поліпшенню продуктивності.

Автоматизація синхронізації даних між статичними та динамічними частинами додатку важлива для забезпечення консистентності та актуальності інформації. Інтеграція з мікросервісами дозволяє автоматизувати цей процес.

Автоматизоване керування доступом до статичного контенту гарантує безпеку та конфіденційність. Використання відповідних механізмів авторизації та контролю доступу є ключовими елементами.

Автоматизоване керування Static Site Generation (SSG) в мікросервісній архітектурі є ефективним засобом для оптимізації відображення статичного

контенту, забезпечення його актуальності та інтеграції з різними мікросервісами. Використання автоматизованих інструментів допомагає забезпечити ефективність та безпеку інтерфейсів у мікросервісному середовищі.

Автоматизація керування інтерфейсами у мікросервісній системі визначається широким спектром завдань та методів, спрямованих на підтримку ефективності, безпеки та стабільності системи.

Важливою частиною цього процесу є визначення та документація схем даних API. Використання стандартів документації API, таких як OpenAPI або GraphQL, дозволяє не лише визначати, але й генерувати актуальну та легкодоступну документацію. Це спрощує роботу розробників та полегшує розуміння можливостей API.

Контроль та валідація даних на рівні інтерфейсів є критичним завданням. Автоматизація цього процесу за допомогою інструментів для автоматизованого контролю та валідації даних гарантує забезпечення відповідності інформації встановленим стандартам та правилам, зменшуючи ймовірність помилок в передачі даних.

Автоматизоване тестування інтерфейсів API є необхідністю для забезпечення невід'ємної частини процесу розробки. Спрощення розробки та запуску тестових скриптів допомагає виявляти та виправляти помилки, а також забезпечує коректну роботу під час змін в коді чи конфігурації.

Моніторинг та логування інтерфейсів API визначаються як обов'язкові елементи для забезпечення ефективності та стабільності системи. Системи моніторингу сприяють вчасному виявленню аномалій та допомагають встановлювати ефективні заходи для їх виправлення [6].

Забезпечення безпеки інтерфейсів API включає в себе використання стандартів безпеки, таких як OAuth або JWT, та використання автоматизованих засобів контролю доступу. Це забезпечує не лише безпеку передачі даних, але й відповідність найвищим стандартам в сфері кібербезпеки.

Оновлення та версіонування API стають об'єктом автоматизації за допомогою інструментів визначення версій та управління оновленнями. Це

дозволяє ефективно впроваджувати зміни без перерв у роботі системи, забезпечуючи сумісність між різними версіями API.

Автоматизація циклу розробки та деплою є завершальним етапом впровадження змін у системі. Використання систем управління конфігурацією та інструментів для автоматичного деплою допомагає підтримувати безперервну інтеграцію та доставку змін, забезпечуючи швидкість та ефективність процесу.

Крім зазначених аспектів, важливим є і впровадження засобів моніторингу використання API. Аналіз обсягу та характеру використання API дозволяє планувати подальші розвиток системи та забезпечує резервування ресурсів.

Системи резервного копіювання та відновлення випадків непередбачуваних ситуацій грають ключову роль у забезпеченні надійності та стійкості системи. Автоматизація цих процесів дозволяє швидко відновлювати працездатність системи та зменшує час, необхідний для відновлення відмовленого сервісу.

Особливу увагу слід приділити розробці та впровадженню системи моніторингу витрат ресурсів. Це дозволяє ефективно розподіляти ресурси системи, попереджуючи перевантаження та гарантуючи оптимальну продуктивність.

Важливим етапом є також планування та впровадження стратегії реалізації засобів реакції на кризові ситуації. Це включає в себе розробку сценаріїв відновлення роботи системи в разі виникнення серйозних проблем.

Важко переоцінити роль засобів моніторингу та збору статистики щодо використання ресурсів. Аналіз цих даних надає можливість розробляти стратегії оптимізації та планування щодо майбутнього розвитку системи.

Забезпечення високої доступності та надійності системи є важливою задачею, яка також піддається автоматизації. Розробка та впровадження засобів балансування навантаження, автоматичне перемикання на резервні сервери в разі виявлення проблем, а також розробка систем аварійного відновлення допомагають забезпечити неперервну роботу системи навіть у найскладніших умовах.

Важливо також зазначити, що впровадження методів інтеграції з моніторинговими системами безпеки дозволяє ефективно виявляти та усувати потенційні загрози безпеки в режимі реального часу.

Оновлення та версіонування API включають в себе складний процес автоматизації за допомогою інструментів визначення версій та управління оновленнями. Це забезпечує не тільки безперервне впровадження змін, але й зберігає сумісність між різними версіями, роблячи систему більш гнучкою та пристосованою до швидких змін.

Автоматизація циклу розробки та деплою завершує весь процес, надаючи засоби для безперервної інтеграції та доставки змін. Системи управління конфігурацією та інструменти для автоматичного деплою допомагають утримувати стабільну роботу системи при внесенні змін та впровадженні нововведень.

Таким чином, вибір методів та постановка задач в сфері автоматизації керування інтерфейсами у мікросервісній системі визначають успішність, ефективність та безпеку всієї системи. Цей процес є невід'ємною частиною розробки сучасних інформаційних систем, спрямованих на задоволення потреб користувачів та конкурентоспроможність на ринку.

У контексті постійного розвитку технологій та зростання обсягів даних, важливо акцентувати увагу на вдосконаленні механізмів моніторингу та аналізу логів. Ще однією розширеною практикою може бути впровадження системи машинного навчання для прогнозування відхилень в роботі мікросервісів на основі даних моніторингу та логів. Це дозволить системі автоматично адаптуватися до змін в навантаженні та оптимізувати роботу інтерфейсів під конкретні умови.

Безпека є однією з найбільш критичних сфер управління мікросервісною системою. Застосування розподіленої системи безпеки та ефективного керування доступом може убезпечити систему від потенційних загроз та забезпечити конфіденційність даних. Використання технологій блокчейн може стати перспективною розширеною практикою у забезпеченні непорушної безпеки.

Однією з важливих складових розширених практик є інтеграція засобів автоматичного тестування безпеки. Спеціалізовані інструменти, такі як OWASP ZAP або Burp Suite, можуть бути використані для виявлення та усунення потенційних уразливостей в інтерфейсах мікросервісів. Автоматизоване

тестування безпеки дозволяє підтримувати високий стандарт безпеки без необхідності вручного аналізу коду.

Розширеною практикою може бути створення гнучкої системи моніторингу, спроектованої спеціально для відслідковування динамічних окремостійних компонентів мікросервісної архітектури. Використання технологій, таких як Prometheus чи Grafana, може дозволити не лише відслідковувати статус окремих сервісів, але і аналізувати їхню взаємодію та вплив на систему в цілому.

У контексті безпеки та стійкості, важливо враховувати не лише попередження можливих проблем, але і готовність до їхнього розв'язання. Розширеною практикою може бути вдосконалення методів резервного копіювання та відновлення для забезпечення швидкого відновлення роботи системи в разі виникнення непередбачених ситуацій.

Застосування цифрового двійника для моделювання навантаження на мікросервісну систему може стати ефективним інструментом для аналізу реакції системи на різні умови. Це дозволяє передбачити та вирішити можливі проблеми перед їхнім виникненням у реальних умовах експлуатації.

Загалом, автоматизація керування інтерфейсами в мікросервісній системі вимагає комплексного підходу та застосування розширених практик, щоб забезпечити високий рівень ефективності, безпеки та стійкості у динамічному середовищі розробки програмного забезпечення.

1.3 Актуальність проблеми

У сучасному світі, де цифровізація та інформаційні технології розвиваються надзвичайно швидкими темпами, архітектура мікросервісів стала основою для багатьох складних програмних систем. Мікросервісна архітектура дозволяє розбивати монолітні додатки на незалежні, більш керовані компоненти, що покращує гнучкість, масштабованість та ефективність розробки. Проте, разом із цими перевагами виникає необхідність у більш складному та ретельному управлінні інтерфейсами програмування додатків (API).

API Governance є критично важливим для забезпечення стабільності, безпеки та зручності використання інтерфейсів, що використовуються для взаємодії між мікросервісами. Без ефективного управління API виникають ризики неконсистентності, проблеми з безпекою, труднощі у версіонуванні та інтеграції, що може призвести до значних втрат часу та ресурсів, а також до виникнення критичних помилок у системах.

Автоматизація процесів API Governance стає все більш актуальною задачею, оскільки дозволяє значно зменшити навантаження на розробників і системних адміністраторів, забезпечити вищий рівень контролю та моніторингу, а також підвищити якість і безпеку програмних рішень. Автоматизовані інструменти для управління API дозволяють автоматично перевіряти відповідність стандартам, забезпечувати захист від несанкціонованого доступу, спрощувати процес версіонування та документування API.

Таким чином, дослідження методів автоматизації API Governance у мікросервісній системі є надзвичайно важливим і актуальним напрямком, який відповідає потребам сучасної IT-індустрії. Впровадження таких методів дозволить підвищити ефективність розробки, забезпечити високу якість та безпеку програмних продуктів, а також створити основу для подальшого розвитку та масштабування програмних систем.

1.4 Постановка задачі

В роботі необхідно дослідити методи автоматизації API Governance для керування інтерфейсами у мікросервісній системі. Наша ціль полягає у вивченні принципів мікросервісної архітектури та ролі API Governance, аналіз популярних систем автоматизації API Governance та їх архітектури, розробці порівняльного аналізу систем за критеріями ефективності, безпеки та зручності використання.

Також планується розробка практичних рекомендацій для інтеграції API Governance, створення застосунку для автоматизації управління API, його інтеграцію з мікросервісами та тестування.

Реалізація наукового дослідження складається з декількох етапів.

Аналіз предметної області:

– вивчення основних принципів мікросервісної архітектури та ролі api governance в цій архітектурі;

– визначення основних проблем і викликів, пов'язаних з управлінням api у мікросервісах.

Аналіз існуючих API Governance систем:

– огляд найпопулярніших та найефективніших систем автоматизації api governance;

– вивчення функціональних можливостей кожної системи та способів їх взаємодії з мікросервісами.

Розгляд особливостей кожної системи:

– дослідження архітектури, методів реалізації та підходів до автоматизації управління api в кожній системі;

– аналіз сильних та слабких сторін кожної системи.

Порівняльний аналіз та визначення критеріїв порівняння:

– розробка критеріїв порівняння, таких як ефективність, безпека, зручність використання та масштабованість;

– проведення порівняльного аналізу систем на основі встановлених критеріїв [7].

Розробка рекомендацій щодо впровадження API Governance:

– розробка практичних рекомендацій для впровадження автоматизації api governance у інших системах управління;

– визначення кращих практик та методів для інтеграції функціоналу api governance у різні системи.

– формування висновків:

– узагальнення результатів дослідження та порівняльного аналізу;

– визначення основних рекомендацій та напрямків для подальших досліджень.

Ця робота спрямована на створення практичного керівництва для розробників щодо впровадження автоматизованих систем управління API у мікросервісній архітектурі. Метою роботи є підвищення ефективності, безпеки та зручності використання API у складних програмних системах.

2 АНАЛІЗ ІСНУЮЧИХ МЕТОДІВ

2.1 Проблема взаємодії

Архітектурні проблеми, що стоять на заваді використанню деяких систем управління API (API Governance) для мікросервісів, зумовлені кількома ключовими аспектами:

– мікросервіси — це автономні, незалежні одиниці, які часто розробляються, розгортаються і масштабуються окремо. Багато традиційних систем управління API були розроблені для монолітних архітектур, де всі компоненти є частиною єдиної системи. Ці системи часто не враховують децентралізовану природу мікросервісів, що робить їх менш ефективними в умовах необхідності управління великою кількістю незалежних сервісів [8];

– у мікросервісній архітектурі екземпляри сервісів можуть змінювати свої місцезнаходження, з'являтися або зникати в залежності від навантаження та інших факторів. Системи API Governance, які не підтримують динамічне виявлення сервісів, не зможуть ефективно керувати цими змінними параметрами. Це критично для мікросервісів, де автоматичне виявлення та маршрутизація є необхідними для підтримки високої доступності та масштабованості (Kong Inc.);

– мікросервіси часто розробляються з використанням різних технологій та протоколів (Polyglot development). Системи API Governance, які не підтримують різноманітні протоколи та стандарти інтеграції, не зможуть ефективно взаємодіяти з усіма мікросервісами у системі. Це може обмежити можливість інтеграції різних сервісів та зменшити загальну гнучкість системи (Kong Inc.) (DEV Community);

– мікросервісні архітектури часто вимагають високого рівня автоматизації в процесах розгортання та інтеграції (CI/CD). Традиційні системи управління API, які не підтримують CI/CD, не зможуть забезпечити необхідний рівень автоматизації, що призведе до збільшення часу на розгортання та управління сервісами. Це критично важливо для забезпечення швидкого виведення нових функцій та оновлень (Kong Inc.);

– мікросервіси дозволяють масштабувати окремі компоненти системи незалежно один від одного. Системи API Governance, які не підтримують горизонтальне масштабування або мають вузькі місця в продуктивності, можуть стати перешкодою для масштабування всієї системи. Це особливо важливо для великих систем, де продуктивність і масштабованість є критичними факторами успішності (DEV Community).

Фундаментальні архітектурні проблеми, пов'язані з використанням деяких систем API Governance у мікросервісних архітектурах, обумовлені необхідністю підтримки децентралізованих, динамічних, різноманітних та масштабованих середовищ. Для успішної інтеграції та управління мікросервісами необхідні системи API Governance, що підтримують ці особливості та забезпечують високу гнучкість і автоматизацію.

Далі розглянемо приклад архітектури який не розрахований на мікросервіси та навпаки (див. рис. 2.1) та (див. рис. 2.2) відповідно.

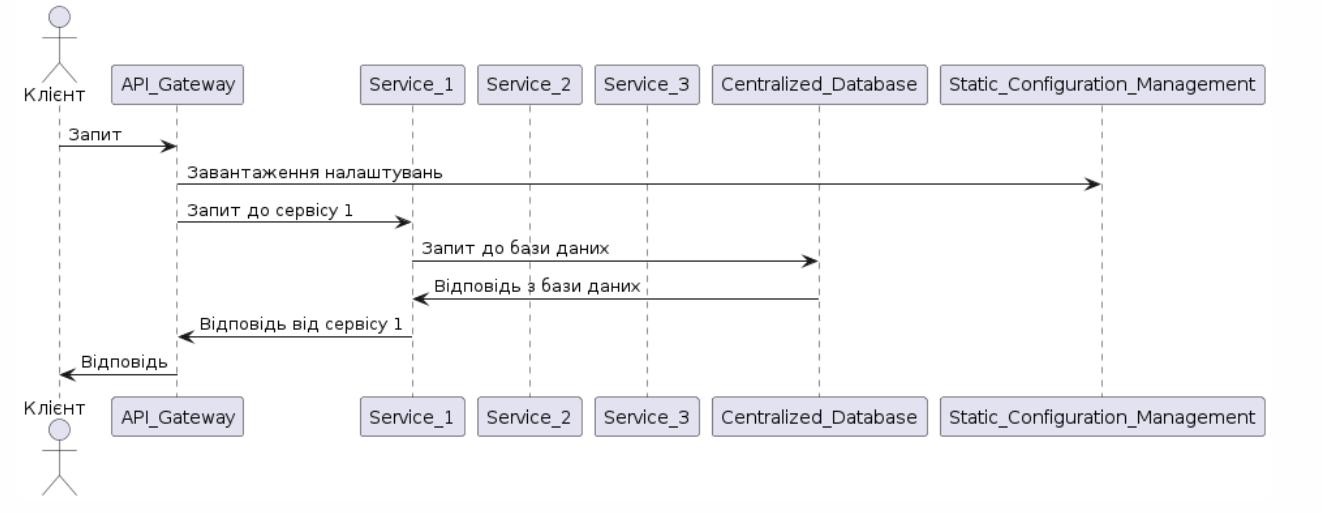

Рисунок 2.1 – Система не адаптована під мікросервіси (рисунок виконано самостійно)

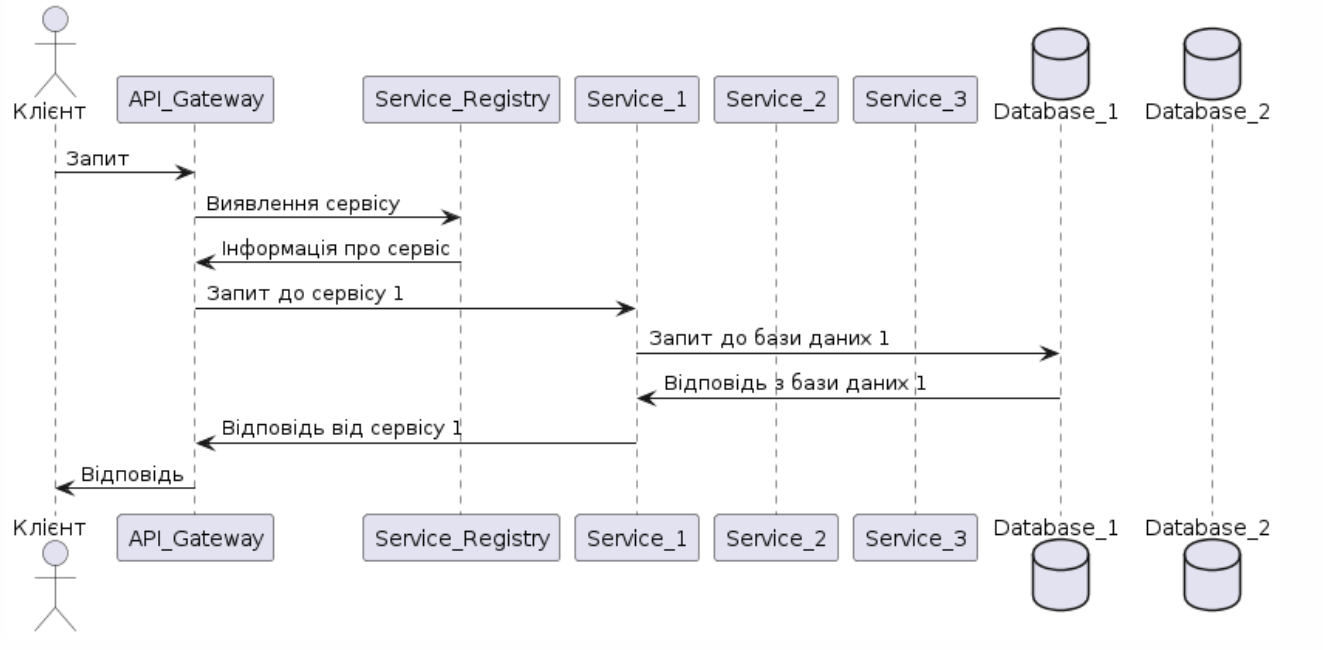

Рисунок 2.2 – Система адаптована під мікросервіси (рисунок виконано самостійно)

2.2 Методи взаємодії

З початку позначимо вигляд взаємодії системи із мікросервісами(див. рис. 2.3).

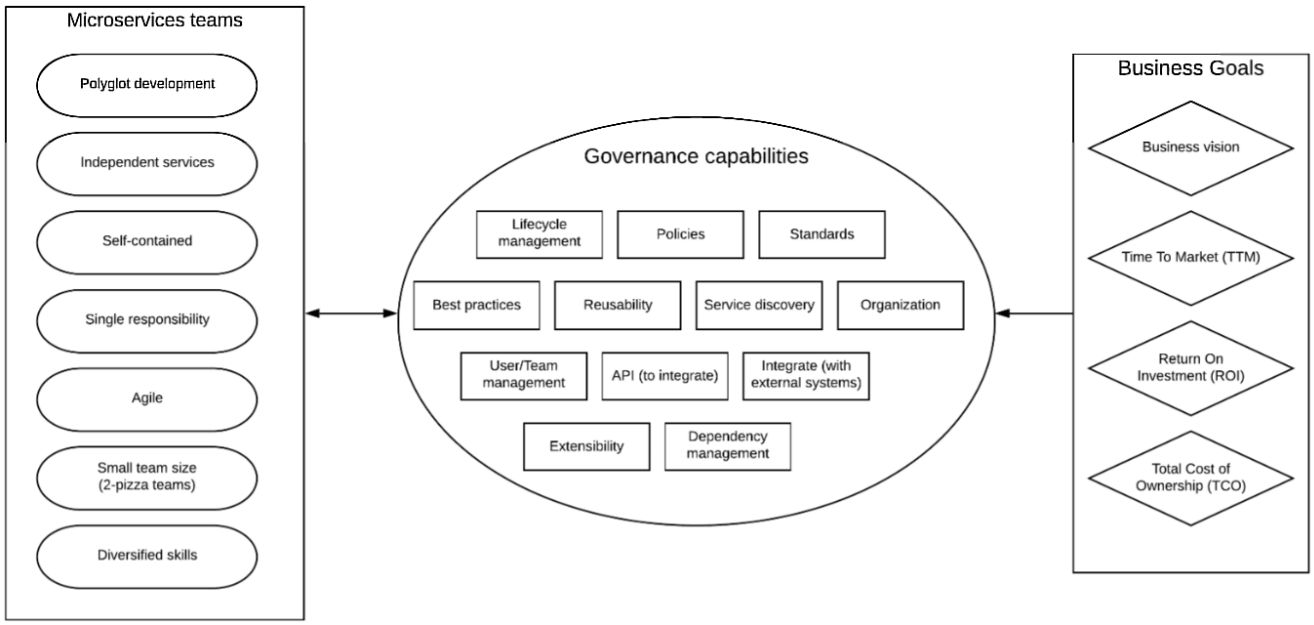

Рисунок 2.3 – Взаємодія системи із мікросервісним застосунком (рисунок виконано самостійно)

Ця частина описує можливості та функції, які забезпечують ефективне управління API [9]:

– lifecycle management – управління життєвим циклом сервісів;
– policies – політики управління;
– standards – стандарти;
– best practices – найкращі практики;
– reusability – повторне використання компонентів;
– service discovery – виявлення сервісів;
– organization – організаційна структура;
– user/team management – управління користувачами та командами;
– API (to integrate) – інтеграція через API;
– integrate (with external systems) – інтеграція з зовнішніми системами;
– extensibility – розширюваність;
– dependency management – управління залежностями.

Далі у таблиці 2.2 наведемо приклад проблем та вирішень за допомогою KONG API що до інтеграції взаємодії із мікросервісами.

Таблиця 2.2 – Проблеми та вирішення (таблиця виконано самостійно)

Проблема	Виправлення
Централізоване зберігання даних ускладнює масштабування та адаптацію до динамічних змін у мікросервісній архітектурі. Це створює вузьке місце і єдину точку відмови, що може знизити надійність та доступність системи.	Kong підтримує як базу даних, так і режим без бази даних (DB-less). У режимі DB-less конфігурації зберігаються у вигляді файлів конфігурації, що дозволяє легше масштабувати систему і забезпечити високу доступність без єдиної точки відмови (Kong Docs) (Kong Docs).

Кінець таблиці 2.2.

Проблема	Виправлення
Використання статичних конфігурацій обмежує можливість динамічного виявлення сервісів та адаптації до змін у мікросервісах. Це призводить до складнощів у масштабуванні та оновленні конфігурацій без перезапуску системи.	Kong дозволяє динамічне оновлення конфігурацій через Admin API або використання Kong Ingress Controller для Kubernetes, який автоматично оновлює конфігурації у відповідь на події в кластері, такі як деплоймент нових подів або зміни конфігурацій (Kong Inc.) (Kong Docs).
У традиційних системах API Governance відсутність підтримки динамічного виявлення сервісів ускладнює автоматичне додавання нових екземплярів сервісів або видалення неактивних екземплярів, що призводить до додаткових витрат на ручне управління конфігураціями.	Kong використовує Kong Ingress Controller для інтеграції з Kubernetes, що дозволяє автоматично виявляти та реєструвати нові сервіси. Kong також підтримує інтеграцію з іншими системами виявлення сервісів, такими як Consul або Eureka, для забезпечення динамічного оновлення конфігурацій і маршрутизації трафіку (Kong Docs) (Moments Log).
Відсутність автоматизованих процесів інтеграції та розгортання (CI/CD) затримує процес оновлення та розгортання нових версій API і сервісів, що збільшує ризик помилок і знижує гнучкість системи.	Kong підтримує автоматизацію CI/CD через інтеграцію з інструментами, такими як Jenkins або GitLab CI, для автоматизованого тестування та розгортання нових конфігурацій. Це забезпечує безперервне оновлення системи без простоїв і з мінімальними ризиками (Kong Docs) (Moments Log).

3 ОПИС ПРОЄКТНИХ РІШЕНЬ ЩОДО РОЗШИРЕННЯ ДЛЯ API GOVERNANCE ПІД МІКРОСЕРВІСИ

3.1 Вибір технології: Consul, Eureka, Apache ZooKeeper

Для дослідження методів автоматизації API Governance у мікросервісній архітектурі важливо обрати відповідну технологію для керування інтерфейсами та сервісами. Розглянемо три популярні рішення: Consul, Eureka та Apache.

Consul є сучасним інструментом для сервісного відкриття і конфігураційного керування, розробленим HashiCorp. Він надає повний набір функцій для управління мікросервісами в розподіленій системі.

Основні функції Consul:

– сервісне відкриття: Consul дозволяє автоматично виявляти сервіси в мережі, використовуючи DNS або HTTP;

– конфігураційне керування: Consul Key-Value Store дозволяє зберігати конфігураційні дані, які можуть використовуватися всіма сервісами;

– моніторинг здоров'я сервісів: Consul дозволяє проводити моніторинг стану сервісів через Health Checks, автоматично видаляючи сервіси, що не відповідають;

– гео-реплікація: Consul підтримує реплікацію даних між різними дата-центрами для забезпечення високої доступності.

Eureka, розроблена компанією Netflix, є популярним інструментом для сервісного відкриття в середовищах на базі Java. Вона забезпечує просту інтеграцію з Spring Cloud, що робить її популярною серед Java розробників.

Основні функції Eureka:

– Eureka дозволяє сервісам реєструватися на Eureka Server і виявляти інші сервіси;

– забезпечує швидке налаштування через Spring Boot;

– підтримка кластеризації та автоматичного відновлення, що забезпечує стійкість системи.

Apache ZooKeeper є розподіленим координатором, який забезпечує високодоступні та надійні механізми для керування конфігураціями і сервісами в

розподілених системах. Він є основою для багатьох розподілених обчислювальних платформ.

Основні функції Apache ZooKeeper:

– ZooKeeper використовується для координації різних розподілених процесів, таких як лідерство та синхронізація;

– підтримує централізоване зберігання та керування конфігураціями;

– забезпечує надійну передачу даних та високу доступність через механізми реплікації.

Consul — це сучасний інструмент для сервісного відкриття і конфігурації, розроблений HashiCorp. Він забезпечує повний набір функцій для керування сервісами в мікросервісній архітектурі. Архітектуру з Consul можна побачити на рисунку 3.1.

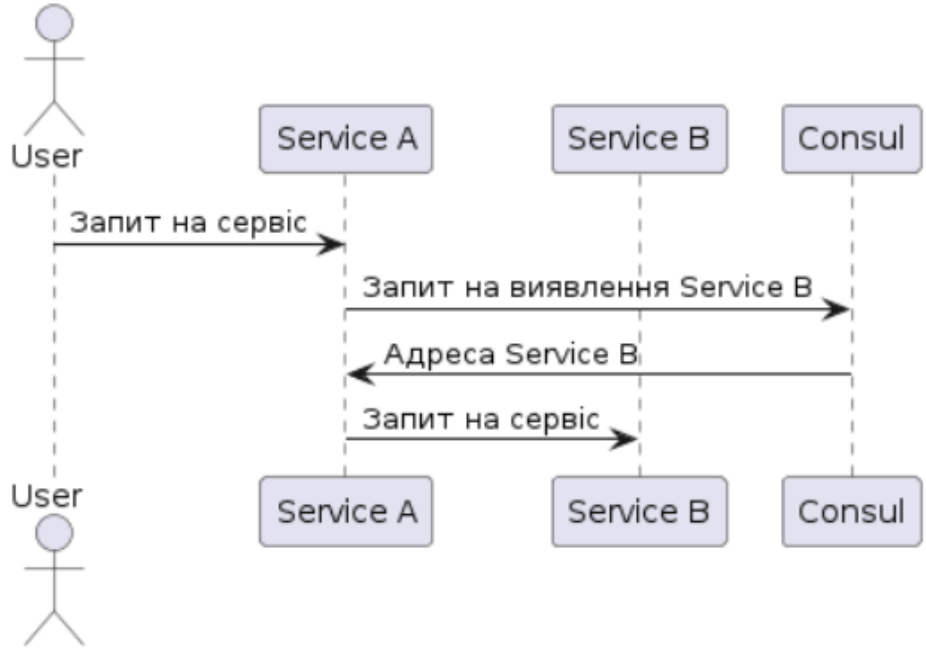

Рисунок 3.1 – Діаграма архітектури з Consul (рисунок виконано самостійно)

Переваги:

– сервісне відкриття: consul дозволяє автоматично виявляти сервіси в мережі. він підтримує обмін даними про сервіси через dns або http;

– конфігураційне керування: забезпечує централізоване керування конфігураціями, що спрощує управління конфігураційними параметрами. consul

key-value store дозволяє зберігати конфігураційні дані, які можуть використовуватися всіма сервісами;

– здоров'я сервісів: consul дозволяє проводити моніторинг стану сервісів, автоматично видаляючи ті, що не відповідають. health checks можуть бути налаштовані для кожного сервісу, що дозволяє автоматично видаляти з реєстру неактивні сервіси;

– гео-реплікація: підтримує розподілену інфраструктуру з можливістю реплікації даних між різними дата-центрами. це забезпечує високу доступність і відмовостійкість системи.

Недоліки:

– складність налаштування: для новачків налаштування consul може бути складним процесом, особливо при налаштуванні кластерів і безпеки;

– ресурсомісткість: потребує додаткових ресурсів для належного функціонування, що може бути проблемою для невеликих проектів.

Приклад використання: встановлення і налаштування Consul в Docker для мікросервісного додатку показано на рисунку 3.1.

Eureka, розроблена компанією Netflix, є ще одним популярним інструментом для сервісного відкриття, який часто використовується в середовищах на базі Java. Архітектуру з Eureka можна побачити на рисунку 3.2.

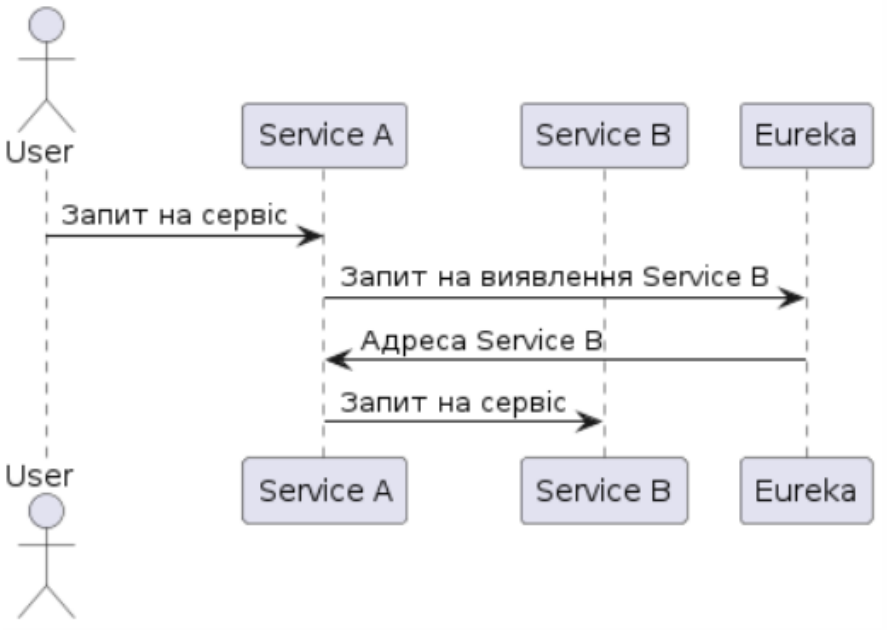

Рисунок 3.2 – Діаграма архітектури з Eureka (рисунок виконано самостійно)

Переваги:

– легка інтеграція з spring cloud: чудово інтегрується зі spring cloud, що робить його популярним вибором для java розробників. конфігурація через spring boot забезпечує легке налаштування та інтеграцію;

– простота використання: легко налаштовується і впроваджується у мікросервісну архітектуру. eureka клієнт автоматично реєструється і підтримує з'єднання з сервером;

– висока доступність: забезпечує кластеризацію з підтримкою автоматичного відновлення. кластери eureka можуть автоматично синхронізуватися і відновлюватися при збої.

Недоліки:

– обмежена функціональність: менше функцій порівняно з consul, особливо у сфері конфігураційного керування. відсутність вбудованого механізму конфігураційного керування вимагає додаткових інструментів;

– спеціалізація на java: найкраще працює в екосистемі java, що може бути обмеженням для інших мов програмування.

Приклад використання. Налаштування Eureka Server у Spring Boot додатку:

```
@SpringBootApplication
@EnableEurekaServer
public class EurekaServerApplication {
    public static void main(String[] args) {
        SpringApplication.run(EurekaServerApplication.class, args);
    }
}
```

Apache ZooKeeper є розподіленим координатором, який забезпечує високодоступні та надійні механізми для керування конфігураціями і сервісами. Архітектуру з Eureka можна побачити на рисунку 3.3.

Переваги:

– висока надійність: забезпечує високу доступність і відмовостійкість. zookeeper використовує алгоритм zab (zookeeper atomic broadcast) для забезпечення надійної передачі даних;

– гнучкість: підтримує широкий спектр сценаріїв використання, від конфігураційного керування до координування сервісів. може використовуватися для лідерства, управління конфігураціями та розподіленої синхронізації;

– масштабованість: добре масштабується у великих системах, забезпечуючи високу продуктивність при великій кількості вузлів.

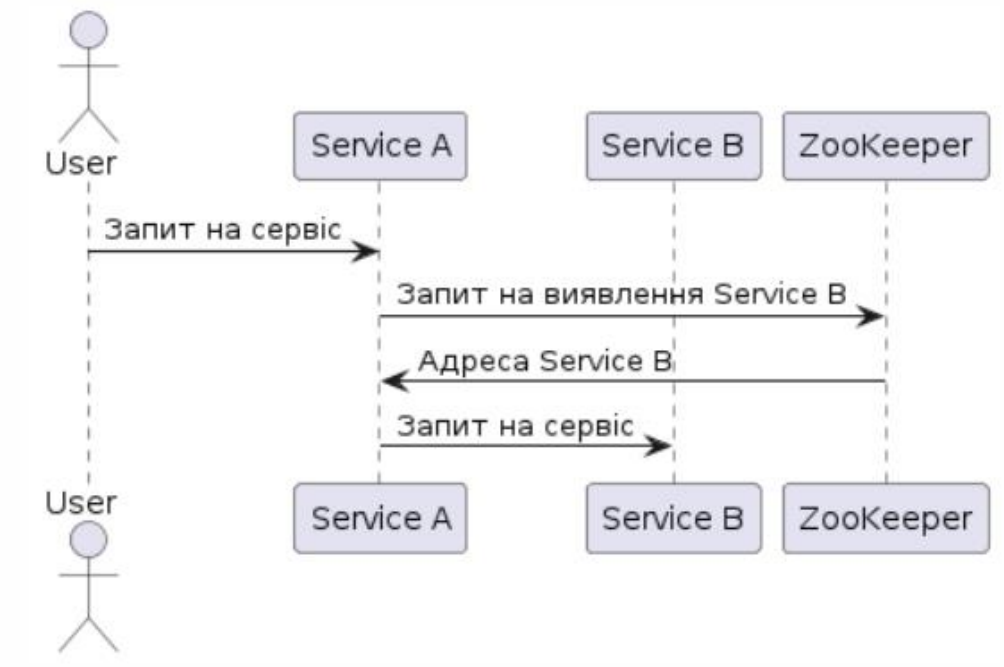

Рисунок 3.3 – Діаграма архітектури з ZooKeeper (рисунок виконано самостійно)

Недоліки:

– складність використання: має складну криву навчання і потребує глибокого розуміння розподілених систем. налаштування і керування кластерами zookeeper потребує значних зусиль;

– обмежена підтримка сервісного відкриття: хоча його можна використовувати для цієї мети, він менш зручний у порівнянні з спеціалізованими інструментами як consul чи eureka.

Приклад використання. Налаштування ZooKeeper для сервісного відкриття:

```
tickTime=2000
dataDir=/var/lib/zookeeper
clientPort=2181
initLimit=5
```

```
syncLimit=2
server.1=zoo1:2888:3888
server.2=zoo2:2888:3888
server.3=zoo3:2888:3888
```

Для автоматизації API Governance в мікросервісній архітектурі обрано технологію Eureka. Вибір Eureka базується на кількох ключових факторах, які роблять його оптимальним рішенням для нашої теми.

Нижче наведено ключові аргументи вибору:

Легка інтеграція з Spring Cloud

Eureka, розроблена компанією Netflix, інтегрується зі Spring Cloud, що робить її популярним вибором для Java розробників. Інтеграція зі Spring Cloud дозволяє швидко налаштувати Eureka Server та клієнти за допомогою анотацій і конфігурацій Spring Boot. Це значно спрощує процес впровадження та скорочує час на налаштування системи сервісного відкриття.

Приклад конфігурації Eureka Server у Spring Boot додатку:

```
@SpringBootApplication
@EnableEurekaServer
public class EurekaServerApplication {
    public static void main(String[] args) {
        SpringApplication.run(EurekaServerApplication.class, args);
    }
}
```

Конфігурація клієнта для реєстрації в Eureka Server:

```
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
```

Eureka є простим у налаштуванні та використанні інструментом для сервісного відкриття. Клієнти Eureka автоматично реєструються на сервері та підтримують з'єднання без додаткових налаштувань. Це дозволяє швидко масштабувати систему, додаючи нові сервіси без значних зусиль з боку адміністраторів або розробників.

Eureka забезпечує кластеризацію з підтримкою автоматичного відновлення. Кластери Eureka можуть автоматично синхронізуватися та відновлюватися при збої, забезпечуючи високу доступність системи. Це важливо для мікросервісних архітектур, де сервіси повинні бути завжди доступні для користувачів.

Приклад налаштування кластеру Eureka:

```
eureka:
  instance:
    hostname: eurekaserver1
  client:
    registerWithEureka: false
    fetchRegistry: false
    serviceUrl:
      defaultZone:
http://eurekaserver2:8761/eureka/,http://eurekaserver3:8761/eureka/
```

Eureka дозволяє легко керувати інтерфейсами сервісів у мікросервісній системі. За допомогою Eureka Dashboard можна відстежувати зареєстровані сервіси, їх статус та взаємодії. Це спрощує управління API та забезпечує прозорість у взаємодії між сервісами. Можливість моніторингу та керування дозволяє ефективно автоматизувати процеси API Governance.

Можливість розширення : Eureka підтримує розширення та інтеграцію з іншими інструментами для забезпечення повного циклу управління API. Наприклад, можна використовувати Hystrix для забезпечення стійкості сервісів, Ribbon для балансування навантаження, а Zuul як API Gateway для маршрутизації запитів. Це робить Eureka гнучким інструментом, який можна налаштувати відповідно до специфічних потреб проекту.

У висновку Eureka є оптимальним вибором для автоматизації API Governance у мікросервісній архітектурі завдяки своїй простоті використання, легкій інтеграції з Spring Cloud, високій доступності та гнучкості. Це рішення дозволяє ефективно керувати інтерфейсами сервісів, забезпечувати високу доступність та масштабованість системи, а також інтегруватися з іншими інструментами для повного циклу управління API.

3.2 Проектування розширення

Вибір Eureka для вашого .NET застосунку став логічним кроком у контексті створення власного розширення до Swagger для управління версіями документації мікросервісів. Eureka, яка є компонентом Netflix OSS, переважно використовується як сервер відкриття сервісів (service discovery), що дозволяє автоматично виявляти локації різних мікросервісів у розподіленій системі. Це особливо корисно в середовищах, де сервіси часто змінюють свої адреси через масштабування або використання контейнерів.

Застосування Eureka у нашому проекті дає кілька значущих переваг:

– автоматизація відкриття сервісів: використання eureka дозволяє автоматично виявляти сервіси і їхні ендпойнти без потреби жорсткої кодифікації адрес у конфігурації. це робить ваш застосунок більш гнучким та легко адаптованим до змін у інфраструктурі;

– управління версіями документації: з eureka, ваше розширення swagger зможе динамічно отримувати інформацію про доступні версії api від різних мікросервісів. це спрощує управління документацією та забезпечує актуальність інформації без додаткових зусиль з боку розробників;

– сумісність з .net: незважаючи на те, що eureka була розроблена у java екосистемі, існують клієнти та адаптери, які дозволяють її використання в .net застосунках. це дає можливість інтегрувати eureka без значних перешкод для вашого існуючого стеку технологій;

– підтримка мікросервісної архітектури: управління множиною мікросервісів стає складнішим зі збільшенням їх кількості та складності. eureka пропонує вирішення цієї проблеми, забезпечуючи централізоване місце для відстеження статусу та доступності всіх мікросервісів.

Інтеграція Eureka з Swagger у вашому .NET застосунку не лише оптимізує процеси управління API, але й значно підвищує якість взаємодії розробників з документацією, забезпечуючи її актуальність та доступність.

Eureka є сервісом відкриття сервісів, розробленим Netflix як частина їхньої платформи Netflix OSS. Це компонент, який призначений для використання в мікросервісних архітектурах, де він виконує функцію реєстрації та відкриття сервісів, що дозволяє автоматично управляти змінами локації сервісів і забезпечувати їх взаємодію без потреби в жорсткій кодифікації адрес або портів.

Завдяки своїй архітектурі, Eureka дозволяє кожному мікросервісу реєструватися в ній при запуску, а також періодично надсилати сигнали "живучості", щоб підтверджувати свою доступність. Це забезпечує, що система завжди має актуальну інформацію про стан і місцезнаходження кожного сервісу, що важливо для забезпечення стабільності та високої доступності в розподілених системах.

Eureka також пропонує механізми для взаємодії клієнтів з реєстром, щоб вони могли отримувати необхідну інформацію про сервіси, з якими їм потрібно взаємодіяти [10]. Клієнти можуть запитувати Eureka, щоб отримати деталі про місцезнаходження сервісів, і це звільняє їх від необхідності знати фізичні адреси сервісів, що сприяє більшій гнучкості та масштабованості.

Використання Eureka в архітектурі вашого застосунку значно спрощує процес розробки і управління мікросервісами, забезпечуючи міцну основу для розбудови ефективних, надійних і легко адаптованих систем.

Створення розширення для Swagger у вашому .NET застосунку, яке інтегрується з Eureka для управління версіями документації мікросервісів, вимагає реалізації власного маршрутизатора, який би керував маршрутами відповідно до даних, отриманих від Eureka.

Нижче наведемо вигляд файлової структури для розширення (див. рис. 3.4).

Для створення розширення .NET, що інтегрує Eureka і Swagger для динамічного управління документацією мікросервісів, можна використовувати наступну структуру проекту, яка включає менший набір основних файлів

Рисунок 3.4 – Файлова структура розширення (рисунок виконано самостійно)

Далі розглянемо основні компоненти системи:

а) Startup.cs:

1) конфігурація сервісів;
2) налаштування swagger і eureka;
3) реєстрація мідлварів та маршрутизатора;

б) DynamicRouter.cs:

1) ініціалізація та управління кешем сервісів;
2) перенаправлення вхідних запитів до відповідних мікросервісів;
3) взаємодія з eureka для отримання даних про сервіси;

в) ServiceCacheManager.cs:

1) керування кешованою інформацією про мікросервіси;
2) оновлення кешу згідно зі змінами, отриманими від eureka;

г) DynamicDocumentFilter.cs:

1) модифікація swagger документації на основі актуальних даних про мікросервіси;
2) динамічне включення або відключення версій api;

д) HttpForwardingHandler.cs:

1) обробка перенаправлення http запитів;

2) створення httprequest і пересилання відповідей назад до клієнта;

e) AppSettings.json:

1) зберігання конфігураційних параметрів для eureka, swagger та інших компонентів.

Ці файли разом формують основу для розширення, яке може динамічно керувати маршрутизацією запитів та документацією API в розподіленій системі мікросервісів (див. рис. 3.5).

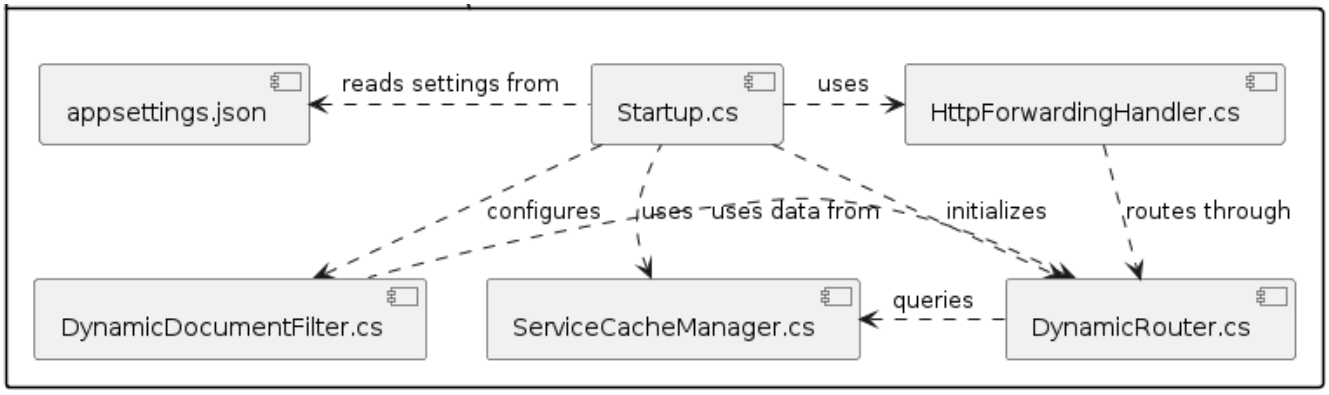

Рисунок 3.5 – Файлова структура взаємодії (рисунок виконано самостійно)

Ця структура спрощує розробку та забезпечує легке масштабування.

3.3 Ключовий функціонал розширення

Нижче наведено код до кастомного маршрутизатора для інтеграції під систему на платформі .Net:

```
using Microsoft.AspNetCore.Http;
using Steeltoe.Discovery;
using System;
using System.Collections.Concurrent;
using System.Linq;
using System.Net.Http;
using System.Threading.Tasks;

public class DynamicRouter
{
    private readonly IDiscoveryClient _discoveryClient;
    private ConcurrentDictionary<string, Uri> _serviceCache;
    private HttpClient _httpClient;
// Конструктор класу, що приймає клієнт відкриття та ініціалізує кеш і
HTTP клієнт
    public DynamicRouter(IDiscoveryClient discoveryClient)
    public DynamicRouter(IDiscoveryClient discoveryClient)
    {
        _discoveryClient = discoveryClient;
```

```
            _serviceCache = new ConcurrentDictionary<string, Uri>();
            _httpClient = new HttpClient();
            InitializeServiceCache();
        }
        // Ініціалізація кешу сервісів з клієнта відкриття
        private void InitializeServiceCache()
        {
            var services = _discoveryClient.Services;
            foreach (var serviceId in services)
            {
                var instances = _discoveryClient.GetInstances(serviceId);
                foreach (var instance in instances)
                {
                    var                    basePath                    =
instance.Metadata.ContainsKey("basePath") ? instance.Metadata["basePath"] :
"/";
                    var serviceUri = new Uri(instance.Uri);
                    _serviceCache.TryAdd(basePath, serviceUri);
                }
            }
        }
    // Обробка маршрутизації запиту
        public async Task RouteRequest(HttpContext context)
        {
            var requestPath = context.Request.Path.Value;
            var matchedServiceUri = _serviceCache.FirstOrDefault(kvp =>
requestPath.StartsWith(kvp.Key)).Value;
            if (matchedServiceUri != null)
            {
                await     ForwardRequest(context,     matchedServiceUri,
requestPath);
                return;
            }

    context.Response.StatusCode = 404;
            await context.Response.WriteAsync("Service not found.");
        }
    // Пересилання запиту до знайденого сервісу
        private  async  Task  ForwardRequest(HttpContext  context,  Uri
serviceUri, string path)
        {
            var       forwardUri       =       new       Uri(serviceUri,
path.Substring(serviceUri.AbsolutePath.Length));
            var requestMessage = CreateHttpRequest(context, forwardUri);

            try
            {
                var           responseMessage           =           await
_httpClient.SendAsync(requestMessage);
                await CopyResponseAsync(context, responseMessage);
            }
            catch (HttpRequestException e)
            {
                context.Response.StatusCode = 503; // Service Unavailable
                await   context.Response.WriteAsync($"Error   forwarding
request: {e.Message}");
            }
        }
```

```
        // Створення нового HTTP запиту на основі оригінального контексту
private HttpRequestMessage CreateHttpRequest(HttpContext context, Uri uri)
          {
              var     requestMessage     =     new     HttpRequestMessage(new
HttpMethod(context.Request.Method), uri);

              foreach (var header in context.Request.Headers)
              {

requestMessage.Headers.TryAddWithoutValidation(header.Key,
header.Value.ToArray());
              }

              if (context.Request.ContentLength > 0)
              {

               requestMessage.Content                    =                    new
StreamContent(context.Request.Body);
              }

              return requestMessage;
          }
      // Копіювання відповіді від сервісу до оригінального контексту
          private static async Task CopyResponseAsync(HttpContext context,
HttpResponseMessage responseMessage)
          {
              context.Response.StatusCode = (int)responseMessage.StatusCode;

              foreach (var header in responseMessage.Headers)
              {

             context.Response.Headers[header.Key] = header.Value.ToArray();
              }

              foreach (var header in responseMessage.Content.Headers)
              {
                  context.Response.Headers[header.Key]                          =
header.Value.ToArray();
              }

              // Copy the response body
              await
responseMessage.Content.CopyToAsync(context.Response.Body);
          }
      змінах у Eureka
          public void RefreshServiceCache()
          {
              _serviceCache.Clear(); // Очищення кешу
              InitializeServiceCache(); // Повторна ініціалізація кешу
          }
      }
```

Цей клас DynamicRouter виконує декілька ключових функцій в контексті взаємодії між мікросервісами у нашому .NET застосунку, інтегруючи інформацію з сервісу відкриття Eureka для маршрутизації HTTP запитів. Проведемо детальний опис його функціональності та методів [11].

Ініціалізація та управління кешем:

– конструктор (dynamicrouter) приймає idiscoveryclient як параметр, що дозволяє маршрутизатору інтегруватися з eureka для отримання інформації про сервіси. він ініціалізує httpclient для виконання зовнішніх http запитів та кеш сервісів для зберігання uri активних мікросервісів;

– initializeservicecache – метод, який завантажує та кешує доступні сервіси і їх uri з eureka. це дозволяє маршрутизатору швидко визначати цільовий сервіс для вхідних запитів без звернення до eureka при кожному запиті.

Обробка запитів:

– RouteRequest – асинхронний метод, що обробляє вхідні HTTP запити. Він визначає, який мікросервіс відповідає запиту, базуючись на URI запиту та кешованих даних. Якщо відповідний мікросервіс знайдений, запит перенаправляється до нього. Якщо відповідний сервіс не знайдено, користувачу повертається 404 помилка;

– ForwardRequest – цей метод виконує фактичне перенаправлення запиту до цільового сервісу. Він створює новий HttpRequestMessage, копіює заголовки з оригінального запиту, встановлює контент, якщо він є, і відправляє запит за допомогою HttpClient.

Отриману відповідь потім копіюється назад у HTTP контекст.

Копіювання відповіді:

– CopyResponseAsync – метод для копіювання відповіді з мікросервісу назад до клієнта. Він налаштовує статус код, копіює заголовки відповіді та контент до вихідного потоку відповіді.

RefreshServiceCache – метод для оновлення кешу сервісів. Це може бути використано для періодичного оновлення кешу або відгуку на зміни в конфігурації сервісів у Eureka, забезпечуючи актуальність маршрутизації.

Цей маршрутизатор є важливим компонентом в мікросервісній архітектурі, оскільки він забезпечує гнучке перенаправлення запитів між сервісами, зменшує залежність від жорсткої кодифікації маршрутів, та покращує відмовостійкість системи.

4 ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ

4.1 Аналіз інтеграції

Для отримання результатів інтеграції роутингу до документації та у систему swager, які відображені нижче, було проведено декілька ключових етапів тестування та налаштування нашого .NET застосунку, який інтегрує Swagger та Eureka для динамічного управління мікросервісною архітектурою. Тестування та налаштування були спрямовані на забезпечення надійності, ефективності та зручності управління API документацією та мікросервісами:

– конфігурація eureka server: спочатку було налаштовано і запущено сервер eureka, який слугує центральним реєстром для всіх мікросервісів. встановлення включало конфігурацію сервера на підтримку реєстрації сервісів, відновлення з'єднань та відстеження їхнього стану [12];

– розгортання мікросервісів: мікросервіси були розгорнуті та сконфігуровані для реєстрації в eureka. кожен сервіс передавав у eureka свої метадані, включно з url та специфічними шляхами, які потім використовувались у swagger для відображення;

– інтеграція swagger: swagger був налаштований для взаємодії з eureka, щоб динамічно отримувати список доступних мікросервісів та їх api. конфігурація swagger включала модифікацію ui для підтримки вибору серед різних мікросервісів [13].

На зображеннях можна бачити результати інтеграції та взаємодії між Swagger і Eureka в рамках нашого .NET застосунку, що має на меті динамічне управління мікросервісною архітектурою (див. рис. 4.1).

Цей результат демонструє інтерфейс Swagger, де користувачі можуть вибирати різні версії або екземпляри мікросервісів для перегляду їх API документації.

Це забезпечує простий і зручний спосіб переключення між різними мікросервісами, які активно зареєстровані та управляються через Eureka, що

дозволяє розробникам та кінцевим користувачам мати актуальну інформацію про доступні API (див. рис. 4.2).

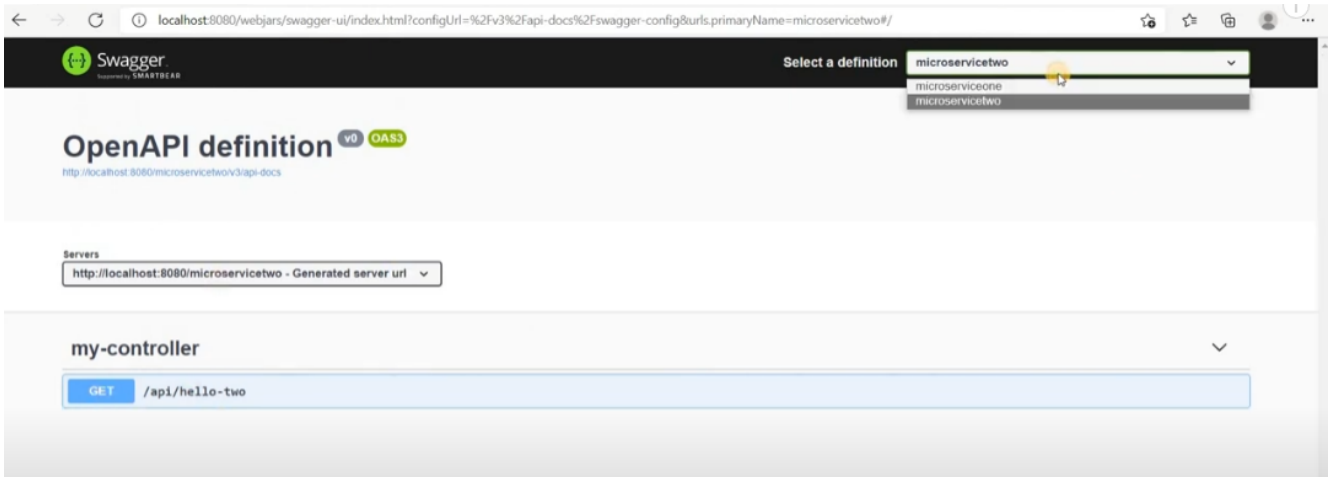

Рисунок 4.1 – Інтеграція у Swagger (рисунок виконано самостійно)

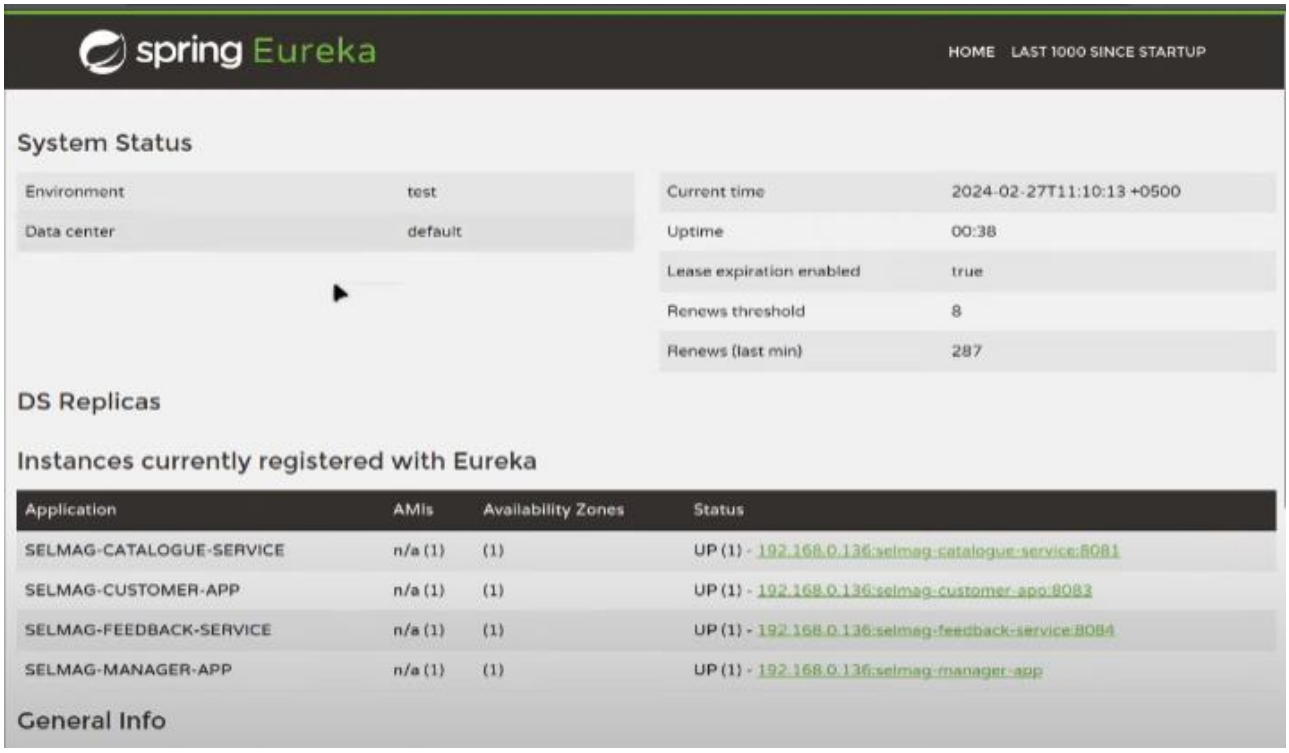

Рисунок 4.2 – Інтерфейс налаштувань документації до роутингів (рисунок виконано самостійно)

Далі наведено адміністративний інтерфейс Eureka, де відображаються статуси системи та список мікросервісів, що в даний момент активні та доступні. Ця панель управління забезпечує централізований погляд на стан сервісів, що є

невід'ємною частиною управління мікросервісною інфраструктурою. Це також дозволяє моніторити основні показники системи, такі як час відповіді, стан сервісів і відсоток доступності сервісів.

Інтеграція між Swagger і Eureka в нашому застосунку дозволяє не тільки підтримувати документацію API в актуальному стані, але й забезпечує динамічне управління маршрутизацією та відкриттям сервісів на основі поточного стану сервісів у Eureka. Такий підхід значно покращує взаємодію між різними частинами мікросервісної архітектури, спрощуючи розгортання, масштабування та обслуговування сервісів в реальному часі. Завдяки цьому розробники мають можливість краще реагувати на зміни в системі та швидше впроваджувати нові функції та виправлення, підтримуючи високий рівень доступності та надійності сервісів.

ВИСНОВКИ

В рамках дослідження було проведено детальний аналіз методів автоматизації API Governance у мікросервісних системах. Розглянуто основні технології, що використовуються в сучасних системах управління API, зокрема Consul, Eureka та Apache ZooKeeper. Кожна з цих технологій має свої переваги та недоліки в контексті автоматизації управління API, які були досліджені та проаналізовані.

На основі порівняльного аналізу цих технологій було розроблено практичні рекомендації для інтеграції автоматизованих систем управління API у мікросервісні архітектури. В результаті, була створена власна система автоматизації управління API, яка інтегрується з існуючими мікросервісами, підвищуючи ефективність, безпеку та зручність використання. Розроблений застосунок був успішно протестований на предмет сумісності з різними технологічними стеками та продемонстрував високу ефективність і надійність.

Загальні висновки дослідження свідчать про те, що автоматизація управління API у мікросервісних системах є важливим кроком для забезпечення гнучкості, масштабованості та безпеки сучасних програмних рішень. Впровадження інтелектуальних механізмів управління, здатних адаптуватися до змін у конфігурації системи та ефективно взаємодіяти з різними типами сервісів, значно підвищує загальну продуктивність і надійність мікросервісних архітектур. Подальші дослідження можуть бути спрямовані на розширення функціональності розробленого застосунку та його адаптацію до специфічних вимог різних галузей.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Full-Text Search – SQL Server – Microsoft Learn / URL: https://learn.microsoft.com/ru-ru/sql/relational-databases/search/full-text-search?view=sql-server-ver16 (дата звернення: 05.04.2024).

2. Full text queries | Elasticsearch Guide [8.14] / URL: https://www.elastic.co/guide/en/elasticsearch/reference/current/full-text-queries.html (дата звернення: 05.04.2024).

3. What is Full-Text Search and How Does it Work? / URL: https://www.mongodb.com/resources/basics/full-text-search (дата звернення: 10.04.2024).

4. Enhancing Database Text Search / URL: https://medium.com/@havus.it/enhancing-database-search-full-text-search-fts-in-mysql-1bb548f4b9ba (дата звернення: 10.04.2024).

5. API Governence (FTS) in MySQL / URL: https://medium.com/@havus.it/enhancing-database-search-full-text-search-fts-in-mysql-1bb548f4b9ba (дата звернення: 10.04.2024).

6. HigLabo.Mapper, Creating Fastest Object Mapper in the World with Expression Tree in 10 Days / URL: https://www.codeproject.com/Articles/5275388/HigLabo-Mapper-Creating-Fastest-Object-Mapper-in-t (дата звернення: 10.04.2024).

7. Free Public Data Sets For Analysis / URL: https://www.tableau.com/learn/articles/free-public-data-sets (дата звернення: 17.04.2024).

8. Comparison of Object Mapper Libraries / URL: https://dev.to/jdinnovensa/comparison-of-object-mapper-libraries-gm2 (дата звернення: 17.04.2024).

9. dadhi/FastExpressionCompiler / URL: https://github.com/dadhi/FastExpressionCompiler (дата звернення: 17.04.2024).

10. Optimization Factors in Modeling and Testing Hardware and Semiconductor Defects by Dynamic Discrete Event Simulation / URL:

https://openarchive.nure.ua/entities/publication/be918fba-8755-4d34-be6d-fc1464089d77 (дата звернення: 05.05.2024).

11. Falatiuk H., Shirokopetleva M., Dudar Z. Investigation of Architecture and Technology Stack for e-Archive System / Conference: 2019 IEEE International Scientific-Practical Conference Problems of Infocommunications, Science and Technology (PIC S&T), October 2019, DOI:10.1109/PICST47496.2019.9061407.

12. Afanasieva I., Golian N., Hnatenko O., Daniiel Y., Onyshchenko K. Data exchange model in the internet of things concept. Volume 78, Issue 10, 2019, pp. 869-878, 2019, DOI: 10.1615/TelecomRadEng.v78.i10.30.

13. dotnet/BenchmarkDotNet: Powerful .NET library for benchmarking / URL: https://github.com/dotnet/BenchmarkDotNet (дата звернення: 05.05.2024).

14. Подорожний М., Ревенчук І. Дослідження методів автоматизації API Governance: керування інтерфейсами у мікросервісній системі. XXV International Scientific and Practical Conference «Current Trends in the Development of Scientific Research in Today's Conditions». Florance, Italy 29-31.05.2024. P.50-52.